Ethereum ZK Rollup Framework

Technical Litepaper

April 2026

Eddy Boughioul, Evan Kereiakes, Martin Ondejka

Zeko brings programmable privacy, recursive ZK proofs, and sovereign rollup deployment to Ethereum - utilizing o1js, the TypeScript library for building advanced ZK smart contracts.

1 Executive Summary

Zeko is proposing a zero-knowledge (ZK) sovereign rollup framework that lets any team deploy a purpose-built L2, L3, or arbitrarily deep rollup stack with programmable privacy - settling directly to Ethereum L1 using recursive ZK proofs.[1, 2] A canonical Zeko L2 would serve developers who need shared infrastructure and composability. Both offerings are powered by o1js - a ZK domain-specific language built by o1Labs that compiles application logic to arithmetic circuits without requiring developers to leave the TypeScript ecosystem.[3, 4]

This proposal creates a Zeko rollup instance and infrastructure native to Ethereum. The design principle that makes the Ethereum path credible is architectural: the proof pipeline - Kimchi/Pickles proving, ZK VM execution, Groth16 wrapping - converts Zeko's native proofs into a form that a Solidity verifier contract can check cheaply on Ethereum L1, without any trust assumptions.

This litepaper describes the full technical architecture, the Ethereum settlement pipeline, the ETH and ERC-20 bridge design, the programmable privacy model, cost economics, and the developer experience.

Key Properties at a Glance

Property	**Zeko on Ethereum**
Proof System	Kimchi/Pickles (PLONKish) → Groth16 on BN254
Settlement Layer	Ethereum L1
Data Availability	EIP-4844 blobs
Language	TypeScript / o1js - no new language required
Bridge Assets	Native ETH + ERC-20 tokens (trustless, ZK-proven)
Privacy Model	Programmable: client-side or server-side proving, private inputs never on-chain
Finality	Cryptographic - no challenge window, no fraud proofs
Speed	No blocks - instant transaction inclusion by sequencer
Recursion	Native via Pickles - constant-size proof regardless of batch depth

2 Zeko Account and Transaction Architecture

Zeko does not execute contracts on-chain. Instead, it verifies proofs that attest that the execution was performed correctly off-chain. A transaction in Zeko does not represent a sequence of instructions, but a set of state transitions that must be validated against the current state.

2.1 The Account Model

An account is defined by two elements: a public address and a tokenId, where the tokenId is a field. The account identifier is the combination of these two fields.[5] Each account contains several important fields:

- balance

- nonce
- timing
- permissions
- elements
- action state
- verification key

2.1.1 The Action State

The action state is a native primitive in Zeko used to append concurrent updates to a sequence of actions emitted by a ZK application. It is constructed as a sequential hash-based commitment, where each new action updates the state as follows:

$$\text{actionState}_{n+1} = \text{Poseidon}(\text{actionState}_n, \text{hash}(\text{action}_n))$$

This produces a single value that commits to the entire ordered sequence of actions. The structure is append-only and deterministic: given the same sequence of actions, the resulting state is always identical. In Zeko, this mechanism is used to represent cross-domain events, such as deposits coming from Ethereum. By reconstructing the sequence of actions, it is possible to prove that a specific event has been included in the system.

Because this construction relies on iterative Poseidon hashing, it is not efficient to verify directly on Ethereum. It will be more efficient to do this in a dedicated circuit and commit its state to the bridge.

2.1.2 The Verification Key

Zeko smart contracts are executed off-chain, and only the contract's verification key is stored on-chain. The contract itself is compiled and executed off-chain, locally in the user's browser or machine to maximize privacy, or this can be done on a server for faster proof generation. The resulting proof is then submitted to Zeko, where the network verifies that the account update and proof are valid with respect to the stored verification key.

2.2 Transactions and Batching

Transactions in Zeko are composed of account updates. Each account update describes a modification applied to an account, along with the conditions under which this modification is valid. A transaction can contain multiple account updates, and all of them must be valid for the transaction to succeed. Each account update includes: i) target account, ii) applied state changes, iii) preconditions, iv) authorization. Preconditions define the expected state of the account before the update is applied. If the current state does not match these conditions, the transaction fails. This model ensures that state transitions are deterministic and verifiable without re-executing the underlying logic.

Zeko processes transactions off-chain, validates and sequences them on-chain, bundles them into batches, and then commits a cryptographic proof of each batch's validity to the L1 chain. The sequencer is a key component in this process, acting to gather transactions and apply them to the current state. It batches the transactions for efficient settlement to the L1, then proves the validity of these batches using zero-knowledge proofs. A root SNARK proof representing the batch is created and committed to the L1 ZK app's account state.

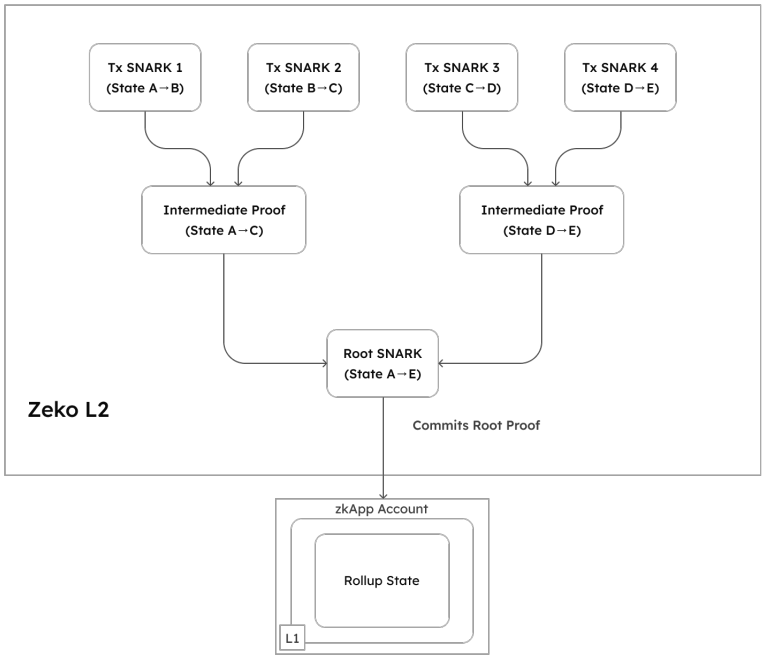

Figure 1: Zeko ZK rollup architecture: sequencer, batch proving, L1 commitment

This process leverages transaction SNARKs, which prove the transition from one ledger state to another. These proofs are composed into a binary tree structure. The root SNARK of this tree serves as a proof for the entire batch. The constant size of this proof is how Zeko compresses the computation of validating a batch of transactions, significantly increasing transaction throughput.

3 Proof System & Ethereum Settlement Pipeline

3.1 Kimchi / Pickles - The Native Proving Stack

Zeko's proving system is built on Kimchi, a PLONKish argument system developed by o1Labs and implemented in Rust.[6] Kimchi requires no trusted setup ceremony, which removes a significant trust assumption common in other SNARK systems. The recursion layer, Pickles, is implemented in OCaml and compiled to JavaScript via `Js_of_ocaml`. Pickles enables native recursive proof composition: a proof can take other proofs as inputs, compressing arbitrarily deep computation

chains into a single constant-size output. This is the foundation of Zeko's batching architecture.

For proving mode, o1js currently uses a WASM prover supporting both browser-based client-side proving and Node.js server-side proving. o1Labs is actively building a Native Prover using direct Rust FFI for Node.js, which will materially improve server-side proving throughput for large ZK app circuits with no changes required to application code.

3.2 Settlement Pipeline - From Kimchi Proof to Ethereum

The five-stage pipeline converts a Zeko Kimchi/Pickles proof into a form verifiable by a Solidity smart contract, while ensuring consistency between Ethereum-native commitments and Zeko-native commitments. A key challenge in this design is that different environments rely on different cryptographic primitives:

- Ethereum uses Keccak, SHA-256, and KZG commitments.
- Zeko uses Poseidon for efficient proving inside ZK circuits.

These primitives are not interchangeable: Keccak, SHA-256, and KZG operations are expensive inside Zeko proving circuits, while Poseidon is expensive to verify on Ethereum. As a result, it is not practical to directly recompute Ethereum commitments inside Zeko, or Zeko commitments inside Ethereum. Instead, the system relies on dedicated proofs that reconcile both representations from the same underlying data. The full Zeko proving and settlement pipeline can be broken down as follows:

Pipeline Stage
Developer - TypeScript / o1js
o1js Circuit Compiler (dry-run → constraint system → arithmetic circuit)
Kimchi Prover (Rust → WASM / Native) + Pickles Recursion (OCaml → JS)
Kimchi / Pickles Proof (PLONKish - no trusted setup)
Zeko Sequencer (batch + recursive proof composition via Pickles)
Rust Kimchi Verifier (o1Labs, in development) - native Rust verification of Kimchi/Pickles proof
ZK VM (SP1) - proves verifier ran correctly → STARK proof
Groth16 Wrap - STARK → compact 200-byte SNARK on BN254
Ethereum L1 - Solidity Groth16 Verifier Contract (250–300k gas units, per batch)

Step 1 - Kimchi/Pickles proof generation:
The Zeko sequencer aggregates transactions into a batch and generates a PLONKish batch proof via Kimchi with full Pickles recursion. No trusted setup.

Step 2 - Rust Kimchi verifier:
The Kimchi verifier is implemented in pure Rust by o1Labs. This binary verifies the Kimchi/Pickles proof natively outside of a browser or JS runtime.

Step 3 - ZK VM execution:
The Rust verifier runs inside a SP1 (ZK VM).[7] SP1 enables standard Rust support, thus facilitating the integration of Mina Rust. Its precompilations can also be used for hashing functions such as Keccak and SHA256.

The ZK VM proves that the Rust verifier executed correctly and outputs a Groth16 proof and the corresponding Solidity verification method. In addition to verifying the execution proof, the ZK VM is also used to generate auxiliary proofs that bind Ethereum and Zeko data representations:

- A bridge proof, which reconstructs both the Ethereum bridge state (using Keccak) and the Zeko action state (using Poseidon) from the same sequence of deposits.
- A data availability proof, which shows that the data used by Zeko is the same data as the blob submitted on Ethereum.

These proofs ensure that all commitments, Ethereum-side and Zeko-side, are derived from the same underlying data without requiring either environment to recompute the other's cryptographic primitives.

The ZK VM's STARK proof is wrapped into a compact Groth16 SNARK on the BN254 curve. The resulting proof is approximately 200 bytes - small and inexpensive to verify on-chain.

Step 4 - Solidity verifier contract:
The Groth16 proof and public inputs are submitted to a verifier contract on Ethereum L1. Verification costs approximately 250–300k gas per batch, a fixed cost amortized across all transactions in the batch. The on-chain settlement transaction contains:

- the Groth16 proof
- the public inputs (previous state root, new state root, batch hash, batch index)
- the blob versioned hash(es) referencing EIP-4844 data
- the bridge state commitments
- the Poseidon commitment of the blob data

On successful verification, the contract i) verifies that the current on-chain state matches the expected previous state, ii) verifies the proof, iii) updates the canonical Zeko state root, iv) stores the new bridge state and data availability commitments. If verification fails, the batch is rejected and the state remains unchanged.

Each step in the pipeline is transparent and deterministic. The Groth16 proof is constant-size (200 bytes) regardless of how many transactions or recursive proof layers are batched inside it. This is the key property that makes per-transaction settlement cost decrease as throughput increases. The total settlement cost is expected to be around 350k–400k gas units.

3.3 Data Availability via EIP-4844 Blobs

To reconstruct Zeko's state, we do not need the entire transaction - only the account update information (state diffs stripped of Pickles proofs), along with any additional information required by the bridge. This information is published via EIP-4844 blob transactions, which are approximately 10x cheaper than equivalent calldata.[8]

Blobs carry the full execution data necessary to reconstruct Zeko state independently, enabling permissionless verification and independent indexers. Typical batches require 1–3 blobs depending on transaction volume.

4 Full Implementation on Ethereum

For Zeko to function as a Layer 2 on Ethereum, we need to verify its state changes on that chain. Rather than attempting to directly verify a Zeko Kimchi/Pickles proof on Ethereum, Zeko introduces an approach based on reconstructing and verifying state transitions - by validating ZK App commands, reconstructing account state, and enforcing preconditions within dedicated circuits.

To maintain code consistency between Zeko and Ethereum, the goal is to validate the transaction originally intended for Zeko so that it can also be validated on Ethereum. To do this, an account state similar to Zeko's is reconstructed in the settlement circuit.

4.1 Verify the ZK App Command

The ZK App command is the transaction originally intended for Zeko. It includes the preconditions, the proof, and the new state. The circuit validates this command in the context of the reconstructed Ethereum settlement. The data required to reconstruct the app state are state root, blobHash, and actionState. For an implementation-oriented Rust reference, see [9].

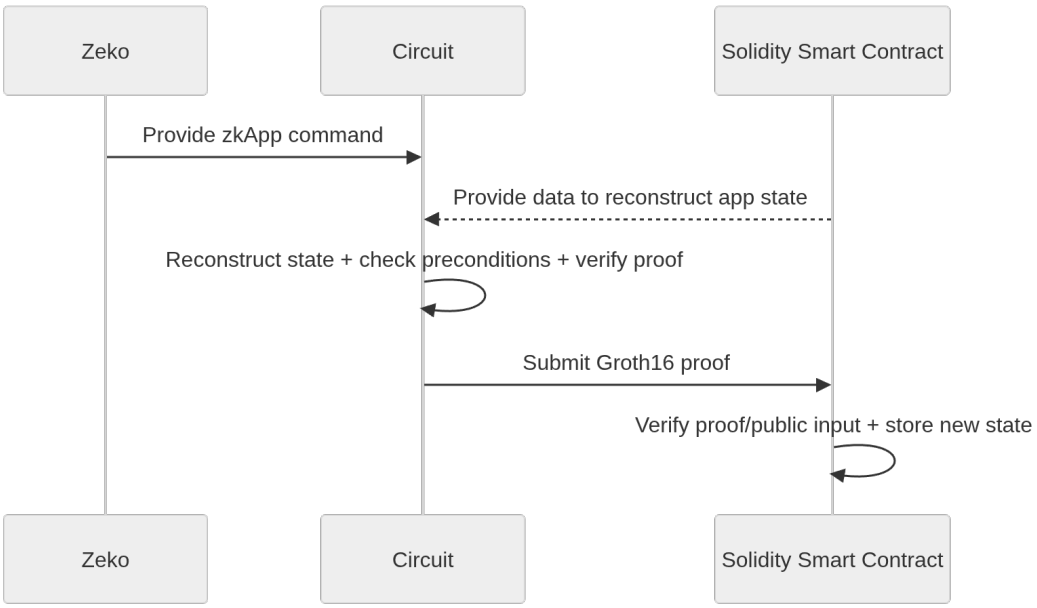

Figure 2: ZK app command verification flow for Ethereum settlement

4.2 Bridge Validation

The bridge is the primary trust boundary between Ethereum L1 and Zeko L2. Its design ensures that no participant can create or release assets on either side without a corresponding cryptographic proof. Initially, the bridge will support native ETH and ERC-20 tokens.

Security invariant: funds are minted on Zeko only when locked on Ethereum; funds are released on Ethereum only when a valid ZK proof confirms the withdrawal is included in a finalized Zeko state root.

The bridge is trustless because the sequencer cannot mint ETH deposits on Zeko without a corresponding lock event on Ethereum. The sequencer cannot release ETH on Ethereum without a valid Groth16 proof of the withdrawal being included in a finalized state root. All enforcement is in Ethereum smart contracts, not in Zeko operator behavior. And if the sequencer goes offline, withdrawals remain possible via the last finalized state root - users are never locked in.

4.2.1 Ethereum → Zeko Deposits (L1 → L2)

Deposit flow steps:

- A user sends a transaction to the L1 Bridge smart contract on Ethereum.
- The contract locks ETH or ERC-20 tokens and emits a Deposit event with sender, amount, token address, and a unique deposit nonce.
- Zeko nodes monitor Ethereum for deposit events via an L1 listener.
- The deposit event is transformed into an L2 deposit transaction and submitted to the Zeko sequencer.
- The deposit transaction is executed on Zeko: ETH or tokens are minted on L2 and balances are updated in the Zeko state tree.
- The deposit is final on Zeko once the corresponding batch is settled on Ethereum.

Deposit Info: Ethereum block confirmation is approximately 1–2 minutes, followed by Zeko inclusion in the next sequencer batch. Total end-to-end finalized Ethereum-to-Zeko deposit time is approximately 5–10 minutes. The deposit cost is estimated to be around 80k gas units for ETH and 120k gas units for an ERC20.

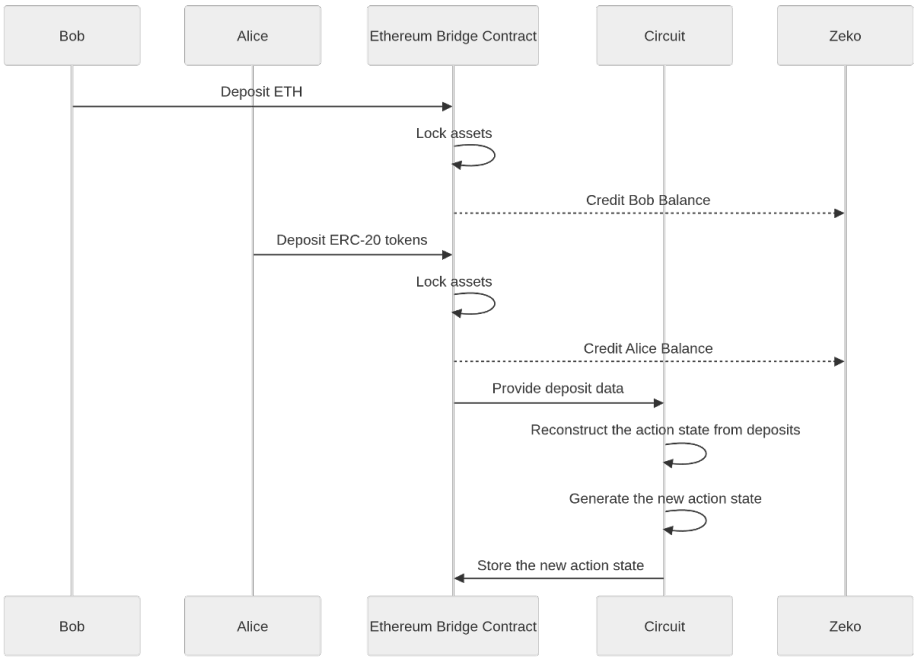

Figure 3: Bridge validation: reconstructing both the Ethereum bridge state and the Zeko action state inside a circuit in order to witness cross-chain events

The circuit takes as input an initial Ethereum state, an initial action state, and a sequence of deposits. It then recomputes the two resulting states by applying the corresponding transition function to the same data. On the Ethereum side, the state is updated by hashing the previous state together with the hash of each new deposit. On the Zeko side, the action state is updated using the Zeko action state transition function.

This ensures that the same sequence of deposits is used to derive both the Ethereum bridge state and the Zeko action state. As a result, Ethereum does not need to verify each deposit individually. It only needs to verify the proof and check that the current on-chain state matches the last generated Ethereum state before accepting the new state.

A simplified specification is:

```
function computeState(initialEthereumState, initialActionState, deposits):
    ethereumState = initialEthereumState
    actionState = initialActionState
    for deposit in deposits:
        depositHash = hashDeposit(deposit)
        ethereumState = Keccak(ethereumState, depositHash)
        actionState = Poseidon(actionState, Poseidon(deposit))
    return (ethereumState, actionState)

function verifyBridge(
    proof,
    initialEthereumState,
    initialActionState,
    expectedEthereumState,
    expectedActionState
```

```
):
    assert(lastGeneratedEthereumState == initialEthereumState)
    assert(expectedEthereumState == onChainEthereumState)
    verifyProof(
        proof,
        initialEthereumState,
        initialActionState,
        expectedEthereumState,
        expectedActionState
    )
    lastGeneratedEthereumState = expectedEthereumState
    lastGeneratedActionState = expectedActionState
```

4.2.2 Zeko → Ethereum Withdrawals (L2 → L1)

Withdrawal flow:

- A user submits a withdrawal transaction on Zeko, burning ETH or tokens and creating a withdrawal record in the Zeko state.
- The withdrawal is included in a sequencer batch.
- Zeko generates a ZK proof covering batch execution and the updated state root containing the withdrawal record.
- The batch settles on Ethereum: proof verified, new state root finalized in the Settlement contract.
- The user submits a withdrawal claim to the L1 Bridge contract, providing withdrawal data and a Merkle proof of inclusion in the finalized state root.
- The bridge contract verifies that the state root is finalized, the withdrawal has not been previously claimed, and the Merkle proof is valid.
- The contract releases ETH or ERC-20 tokens to the user.

Withdrawal Info: User waits for batch settlement on Ethereum (batch time + L1 finality). No challenge period needed - this is a ZK rollup, not an optimistic rollup. Total time is approximately 1 hour. The withdrawal gas cost is around 140k gas units for ETH and 180k gas units for an ERC20 token.

To make a withdrawal, the user provides proof that the withdrawal is recorded in the L2 state. The state managing withdrawals is a Merkle tree using a hashing algorithm optimized for Solidity, with its root stored in the settlement contract.

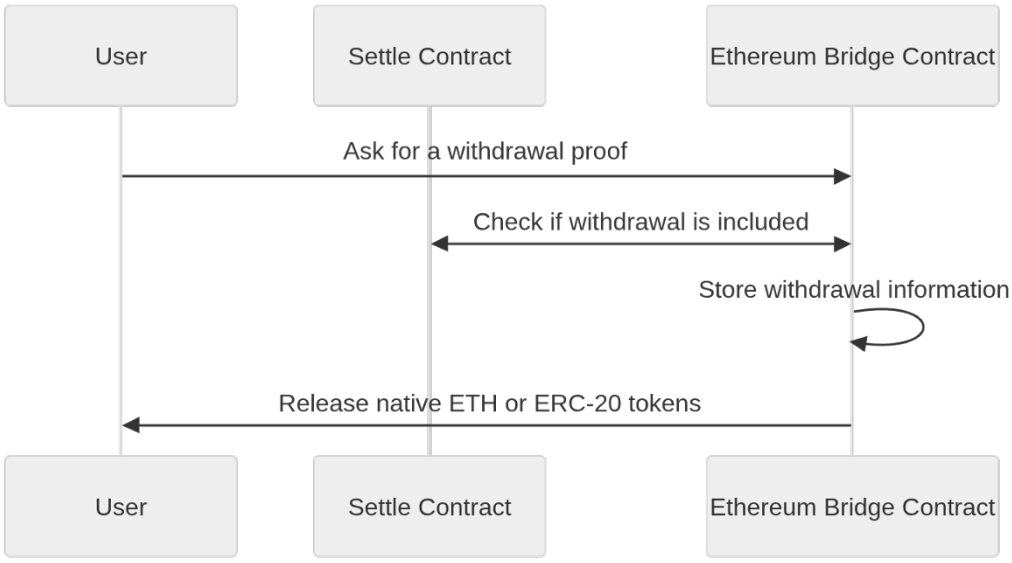

Figure 4: Withdrawal Merkle proof verification on L1

4.2.3 Data Structures

Structure	Purpose
Deposit queue (L1 → L2)	Ordered list of deposit events from L1, consumed by the Zeko sequencer
Withdrawal tree (L2 → L1)	Merkle tree of all withdrawal records; root committed in each settled state
State root (L1 contract)	Canonical Zeko state root stored on Ethereum, updated after each verified batch
Deposit / withdrawal nonce	Monotonically increasing nonce per user to prevent replay attacks
Nullifier set	Set of spent withdrawal IDs stored in the bridge contract to prevent double-spend

4.3 Blob Validation: EIP-4844

To verify that the blob data used in the proof is correct, we must prove that the same underlying data is referenced both on Ethereum and inside the Zeko proof, without any trust assumption on the sequencer. The prover circuit acts as the bridge between the EIP-4844 data availability layer and Zeko's execution layer.[8] On Ethereum, a blob is referenced through its versioned hash, computed from the blob's KZG commitment and accessible on-chain via the `BLOBHASH` opcode. On the Zeko side, the prover takes the raw blob data, reconstructs its field-element representation, and binds these values inside the proof through a Merkle commitment. This Merkle root is the circuit-side commitment to the blob contents.

Following Method 1 from Dankrad's note, the circuit does not verify the full KZG proof directly.[10] Instead, it proves correct computation over the blob data itself, while the on-chain DA verification checks that the same evaluation is valid under Ethereum's EIP-4844 commitment path. In

other words, the circuit proves correctness with respect to the Merkle-committed blob values, and Ethereum verifies that the same blob is also valid with respect to the KZG commitment and opening proof.

The challenge point must be derived identically on both sides so that the value proven inside Zeko is the same value opened against Ethereum's blob commitment. The circuit therefore computes an evaluation point from the blob commitment and Merkle root, evaluates the interpolation polynomial defined by the blob values at that point, and exposes the result as a public value. The DA verifier then recomputes the same evaluation point and asks the EIP-4844 point-evaluation precompile to verify that the KZG commitment opens to the same value.

A simplified specification is:

```
function circuit(merkleRoot, kzgCommitment, publicY,
    leafValues[4096], merkleProofs[4096], zkPrivateInputs):
    # 1. Rebuild / verify Merkle commitment to the blob data
    for i in 0..4095:
        assert verifyMerkleLeaf(
            root = merkleRoot,
            index = i,
            leaf = leafValues[i],
            proof = merkleProofs[i])

    # 2. Derive the challenge point shared with L1
    x = hashToField(kzgCommitment, merkleRoot)

    # 3. Evaluate the interpolation polynomial f at x
    #    where f(omega^i) = leafValues[i]
    y = barycentricEvaluateOutsideDomain(
        values = leafValues,
        point = x)

    # 4. Bind the circuit to that evaluation
    assert y == publicY

    # 5. Continue with the rollup/state-transition logic using leafValues
    nextState = executeRollupLogic(leafValues, zkPrivateInputs)
    return nextState

function verifyBatch(proof, blobIndex, kzgCommitment,
    kzgProof, merkleRoot, publicY, publicInputs):
    # 1. Check that this commitment is the commitment of the blob
    expectedVersionedHash = kzgToVersionedHash(kzgCommitment)
    onchainVersionedHash = blobhash(blobIndex)
    assert expectedVersionedHash == onchainVersionedHash

    # 2. Recompute the same challenge point as the circuit
    x = hashToField(kzgCommitment, merkleRoot)

    # 3. Ask the EIP-4844 precompile to verify f(x) = publicY
    assert pointEvaluationPrecompile(
        versionedHash = onchainVersionedHash,
        z = x,
        y = publicY,
```

```
        commitment = kzgCommitment,
        proof = kzgProof)

    # 4. Verify the zk proof whose public inputs include:
    #    - merkleRoot
    #    - kzgCommitment
    #    - publicY
    #    - other rollup public inputs
    assert verifyZkProof(proof,
        publicInputs = {
            merkleRoot,
            kzgCommitment,
            publicY,
            publicInputs})

    return true
```

This ensures that the blob published on Ethereum and the value used inside the Zeko execution proof are both derived from the same raw data. The circuit proves correct blob processing under a Merkle commitment, while the DA verifier checks that the same evaluation matches Ethereum's EIP-4844 KZG commitment path. Together, these checks close the consistency chain from raw blob data to finalized L2 state, without relying on sequencer honesty.

5 Programmable Privacy and o1js

Privacy in Zeko is the default execution model. When a developer writes a ZK app method in o1js, private inputs to that method never appear on-chain, inside the proof, or in the batch data. They are used solely to generate circuit constraints. The proof attests that the computation was performed correctly without revealing what inputs drove it. Zeko and o1js have two proving modes, client-side and server-side, available to developers and users depending on the UX, level of privacy, or specific part of the stack where privacy is needed.[3, 4]

Mode	**Client-side Proving**	**Server-side Proving**
Where computation runs	User's device (browser or local Node.js)	Dedicated prover server
Privacy level	Strongest - private inputs never leave the user's device	Application data hidden from the public and from Ethereum
Performance	Slower - constrained by user device	Scalable - parallelizable across proving infra

In o1js and Zeko, the developer chooses where proving happens for all logic - a flexibility that matters for AI agent workflows as well as institutional finance and data applications where user devices are not a reliable proving environment. Teams deploying dedicated Zeko rollup instances can enable additional privacy layers via:

- Encrypted data availability
- Encrypted inputs and outputs

These layers are configurable per application and per user requirement, giving enterprise-grade deployments full-stack privacy from the protocol through to the application logic and to user data.

5.1 o1js - TypeScript as a ZK DSL

o1js is both a ZK SDK for production ZK app development and a ZK DSL embedded in TypeScript. On the surface it is pure TypeScript. Underneath, the cryptography is handled by Kimchi (Rust, compiled to WASM) and Pickles (OCaml, compiled to JavaScript). Developers write provable logic as normal TypeScript class methods annotated with `@method` - the compiler handles circuit generation automatically.[3, 4, 6]

Because o1js is TypeScript, all major AI coding assistants have extensive training coverage, making ZK development tractable for teams without a dedicated cryptography expert. There are millions of TypeScript developers worldwide who can onboard to o1js without learning a new language. Additional standardized scaffold templates, agent skills, tooling (SDK, CLI, etc.) and use-case demos are available in the Zeko and o1js GitHub and docs to bootstrap developer productivity, reduce boilerplate and enable consistent team-wide application and rollup development.[1, 2, 3, 4, 11, 12]

6 Conclusion

Zeko extends the o1js paradigm beyond its native environment by bringing its zero-knowledge execution model to Ethereum as sovereign Layer 2 zero-knowledge rollup infrastructure. By reusing the o1js account model, transaction semantics, and proof system, Zeko enables developers to build private ZK rollups and apps with familiar TypeScript tools, while benefiting from Ethereum's security and ecosystem.

Rather than attempting to directly verify Zeko proofs on Ethereum, Zeko introduces an approach based on reconstructing and verifying state transitions. By validating ZK app commands, reconstructing account state, and enforcing preconditions within dedicated circuits, Zeko ensures that only valid state updates can be committed on L1.

The bridge design guarantees correctness through the reconstruction of the action state and nonce-based replay protection, preventing unbacked minting and maintaining a strict 1:1 relationship between L1 assets and their L2 representation. Blob validation under EIP-4844 enables efficient data availability while ensuring that the committed L2 state is fully consistent with the data included in proofs.

Together, these components form a coherent architecture where correctness is enforced through verifiable computation rather than trust assumptions. This technical litepaper demonstrates that it is possible to preserve Zeko's cryptographic guarantees while leveraging Ethereum as a settlement layer. As the ecosystem evolves, Zeko aims to expand its scalable, secure, and developer-friendly environment for private, ZK-native applications on Ethereum - bridging two of the most advanced paradigms in blockchain design.

Zeko + o1js: composable ZK proofs at application scale,
settled with cryptographic finality on Ethereum.

References & Resources

[1] *Zeko Documentation*. URL: https://docs.zeko.io/.

[2] *Zeko GitHub*. URL: https://github.com/zeko-labs.

[3] *o1js Documentation*. URL: https://docs.o1labs.org/o1js.

[4] *o1-labs/o1js*. URL: https://github.com/o1-labs/o1js.

[5] *MIP-0004 zkApps*. URL: https://github.com/MinaProtocol/MIPs/blob/main/MIPS/mip-0004-zkapps.md.

[6] *o1-labs/proof-systems (Kimchi)*. URL: https://github.com/o1-labs/proof-systems.

[7] *SP1 Documentation*. URL: https://docs.succinct.xyz/.

[8] Vitalik Buterin et al. *EIP-4844: Shard Blob Transactions*. Ethereum Improvement Proposal. 2022. URL: https://eips.ethereum.org/EIPS/eip-4844.

[9] *ZK app command verification in Rust*. URL: https://github.com/youtpout/mina-rust/blob/e39b136a1a05317f09d6c6c0df7086f789f10c17/crates/verification-test/tests/test_apply.rs#L72.

[10] Dankrad Feist. *KZG commitments in proofs*. URL: https://notes.ethereum.org/@dankrad/kzg_commitments_in_proofs.

[11] *o1js npm package*. URL: https://www.npmjs.com/package/o1js.

[12] *ZK app CLI*. URL: https://github.com/o1-labs/zkapp-cli.