The Fizk Protocol
White Paper

A crypto-backed confidential stablecoin

GitHub

June 2025

Abstract

Stablecoin supply has grown from ~$2 billion (2019) to >$200 billion (2024), yet most transactions still expose user balances and spending patterns to the public ledger. That transparency creates real-world risks such as targeted hacks, “wrench attacks,” corporate espionage, and other forms of exploitation.

Fizk introduces a confidential stablecoin payment layer. An intent-based, end-to-end zk-proof system delivers high throughput while preserving full cryptographic security guarantees. Governance-controlled audit keys and blacklist functions let the protocol balance privacy with FATF-style oversight mechanisms.

The protocol innovates on a few key areas:

1) **End-to-end zk proofs:** Every state transition is proven by a Pickles/o1js zk circuit, then folded into a single Mina-verifiable proof.
2) **High-throughput intent layer:** User intents are ordered on Sui’s fast finality chain, letting Fizk batch thousands of transfers per block while still inheriting Mina security.
3) **Auditable confidentiality:** Every transaction is encrypted under an on-chain audit key, allowing governance to produce a transparent flow-of-funds report without breaking user privacy.

A decentralised, confidential yet audit-ready payment layer restores cash-like privacy and offers the transparency hooks regulators increasingly request. By eliminating the long-standing privacy-vs-compliance trade-off, Fizk provides a dependable settlement layer for payments and DeFi liquidity across Web3.

Introduction

Financial privacy, both individually and commercially, is a fundamental feature of today's digital world. The importance of financial privacy can be observed ubiquitously throughout the traditional financial sphere. Financial institutions go to extreme lengths to implement privacy preserving features to safeguard their customers' data, with individuals and businesses increasingly concerned with the handling and protection of their data as leaks can have devastating consequences.

Without privacy, individuals would reveal sensitive information with each transaction. Motivated actors would be able to trivially ascertain how much money someone made, what charities they donated to, how much debt they were in. This information could be used to discriminate, coerce, or blackmail. Furthermore, businesses depend on the assurance of financial privacy. If all their financial activities were exposed, rivals and malicious actors could study, anticipate, and front-run their moves. Such transparency would open the door to market manipulation, stifle innovation, and hand unfair advantages to competitors.

Governments around the world explicitly categorise data privacy as a fundamental constitutional right, an umbrella that includes financial information. To that end, they have strictly enforced regulatory requirements and have dished out severe punishments and fines for the mishandling of private data.

While most blockchain platforms provide some forms of privacy in the form of pseudo-anonymity, the distributed ledger that underpins them necessitates complete transparency of balances and transactions. While some privacy focused coins have provided a safe haven for those wishing to achieve desired levels of confidentiality, this is not the case when it comes to stablecoins.

Stablecoins have ballooned in market capitalisation as they provide significant improvements to traditional fiat value exchange with fast and cheap payments, remittances and provide a low-friction crypto volatility hedge. They are the preferred value vehicle in DeFi and on exchanges, generating more that $36 trillion in on-chain settlement volume in the past year alone. This extraordinary growth has been achieved despite the lack of one of the cornerstone features required for true financial sovereignty. Privacy.

Fizk zkRollup

Fizk is a confidential rollup that brings privacy to stablecoin payments. Built on a recursive zk architecture and anchored to Mina, it allows users to deposit popular stablecoins like USDC and USDT into escrow contracts, where they are wrapped into zkUSDC or zkUSDT inside the rollup. These wrapped tokens take the form of encrypted UTXO notes — indistinguishable, unlinkable, and fully confidential — enabling users to send and receive funds without exposing their balances, transaction history, or addresses.

At its core, the Fizk protocol is a series of zk circuits that enforce the rules of state transitions of the `FizkRollupState`. These zk circuits are written as zkPrograms in o1js. o1js is a typescript library for writing zk applications and is foundationally built upon a Kimchi + Pickles proof system which features efficient verification and, uniquely, recursive proving.
Recursive proving allows Fizk to break application logic into a series of nested zkPrograms that all rollup into one succinct proof type that governs the state. Furthermore, it enables merge proofs. As long as we can prove adjacent and consecutive state transitions, we can recursively merge these into one single proof that can represent any number of arbitrary transitions.

***Notation**: Let S_i be the represented FizkRollupState after the i-th transaction t_i.*
We write $\pi_{a,b}$ for a Kimchi proof that evolves from S_a to S_b, and
$\mathrm{Merge}(\pi_{a,b},\ \pi_{b,c})$ *for Pickles' recursive gadget that produces a single proof of $S_a \to S_c$.*

$$\begin{aligned}
\pi_{0,1} &= \mathrm{Prove}\left(S_0 \xrightarrow{t_1} S_1\right) \\
\pi_{1,2} &= \mathrm{Prove}\left(S_1 \xrightarrow{t_2} S_2\right) \\
\pi_{0,2} &= \mathrm{Merge}\left(\pi_{0,1},\ \pi_{1,2}\right) \\
\pi_{0,6} &= \mathrm{Merge}\Big(\pi_{0,2},\ \mathrm{Merge}\left(\pi_{2,4},\ \pi_{4,6}\right)\Big)
\end{aligned}$$

Fizk utilises this powerful feature to roll up application state transitions into a single succinct proof that can be settled to the Mina blockchain for a complete state finality guarantee.

Recursive proofs can be verified in $\mathcal{O}(1)$ time, the settlement on Mina results in a highly decentralised, ubiquitous, and cheap verification of the rollup state on the cryptographic level.

Intent Based Proof System

An intent is a zk proof that proves a user can execute a specific state transition, such as the transfer for zkUSDC without revealing the secret witnesses behind that action. Publishing the intent rather than the raw data means that the network learns nothing about amounts or keys in the confidential system. The “prove-then-rollup” pattern lets users generate transactions privately on client-side devices before they are verified and executed in the rollup, keeping every balance confidential yet every transition deterministic. A transfer-intent is just the simplest concrete case of this general structure, however the same structure underpins every interaction with the protocol, such as deposits, redemptions, and governance votes.

Fizk preserves privacy while maintaining a public auditable state machine with this two-layer proof architecture.

Layer 1 - Intent proofs: A wallet generates an intent proof locally, using its private inputs. For a transfer, the proof commits to the note’s plaintext and includes a Schnorr signature made with the note’s spending key. Without exposing those values, the proof shows that

- The note commitment being spent belongs to the prover.
- The prover controls the corresponding spending key.
- The total value of the inputs equals the total value of the outputs plus any fee.

Only the succinct proof and the encrypted new notes are published.

Layer 2 - Rollup proof: The global roll-up zkProgram consumes the ordered set of intent proofs for a block and applies each one to the `FizkRollupState`. For every intent the circuit

1. Verifies the intent proof.
2. Performs the associated assertions and operations on the state inside the proof.

3. Updates the `FizkRollupState` in the proof's state transition.

After all intents have been processed, the circuit continues to merge adjacent state transitions to finally emit a single recursive proof that the state has moved from and to. Because the proofs are merged recursively, the roll-up fails if even one intent is invalid or sequenced out of order.

This layered approach allows users to construct proofs in parallel while the roll-up maintains a deterministic, verifiable state machine with low on-chain cost.

$$\pi_{0,N} = \mathrm{Merge}(\pi_{0,1},\ \pi_{1,2},\ \pi_{2,3},\ \ldots,\ \pi_{N-1,N})$$

This approach ensures:

- **Concurrency**: Users can create and prove intents in parallel, since they all simply assert ownership over notes and authorisation over actions.
- **Safety**: Double spends, invalid vault interactions, or note mismatches are prevented at the rollup layer by applying changes to the maps, with assertions checked against current state.
- **Determinism**: The rollup logic remains deterministic because state transitions are processed sequentially using the verified, ordered intent proofs.

Key Rollup Components

Decentralised Shared Sequencer (Sui)

Fizk's rollup only works if every intent proof is processed in a deterministic order. Rather than using a centralised sequencer or inventing a bespoke consensus layer, the protocol piggy-back on Sui's validator set, utilising the performant L1 chain as a coordination layer throughout the intent lifecycle. There are a number of key benefits:

- End to end visibility on intent flow throughout its lifecycle.
- A high level data availability reference layer for `FizkRollupState`.
- Optimistic state for the `FizkRollupState`.

- Effective sequencing of intents so the orchestrator can precisely roll the state up into block proofs.
- The entire intent stream lives on Sui, so users and auditors can trustlessly reconstruct and verify the rollup.

Trusted Execution Agents (TEA)

The Fizk protocol operates four types of Trusted Execution Agents for managing intent flow and communication between various chains the rollup operates on.

Each Fizk TEA runs inside an AWS Nitro Enclave. Because every state transition is sealed by an end-to-end zk proof, a malicious validator cannot inject an invalid update onto Mina; attested enclaves therefore let Fizk rely on a small, high-performance validator pool without sacrificing security.

When an enclave boots, AWS hashes every stage of the launch into a set of SHA-384 platform-configuration registers (PCR-0...PCR-3). The enclave then generates an in-RAM key-pair K_{enc}, and AWS issues a short X.509 attestation document that embeds the PCRs and the public key, signed by the Nitro root CA. Using reproducible builds for the agents, anyone holding that root certificate can verify the signature and know that messages signed with K_{enc} came from exactly that code and no other.

The protocol maintains the hash of the different validator types $\mathrm{Root}_{\mathrm{EC}} = H\big(\mathrm{PCR}_0 \,\|\, \dots \,\|\, \mathrm{PCR}_3\big)$ alongside the associated K_{enc} inside the global rollup state S_k.

The PCR’s that correspond the enclave image file are registered on Sui by governance and once the TEA is operational, the attestation is verified on-chain against those PCR’s and the K_{enc} is recorded, so that specific interactions have to be signed by that specified key.

IntentSponsor

The entry point to the Fizk rollup is the TEA-IntentSponsor. Users generate their intent proofs locally and send them to the IntentSponsor accompanied by any new Notes that they have created. The IntentSponsor analyses the proof to ensure that the note the user sent is part of the notes that were committed to inside of the proof. Once verified, the IntentSponsor will encrypt the notes twice, once with the viewing key of the payment address and again with the protocol’s Audit Key. The IntentSponsor will then upload the proof and accompanying encrypted notes to the DA layer. Once the proof is uploaded, the IntentSponsor will create the intent object on Sui ready for sequencing and validation by the Validator

Validator

The TEA-Validator is Fisk's state-keeper and block manager. It watches for NewIntent events from Sui. As it receives an event it downloads and verifies the proof from DA, performing the associated map / state operation in RAM, finally it submits a transaction on Sui, verifying the intent and moving the object into a block.

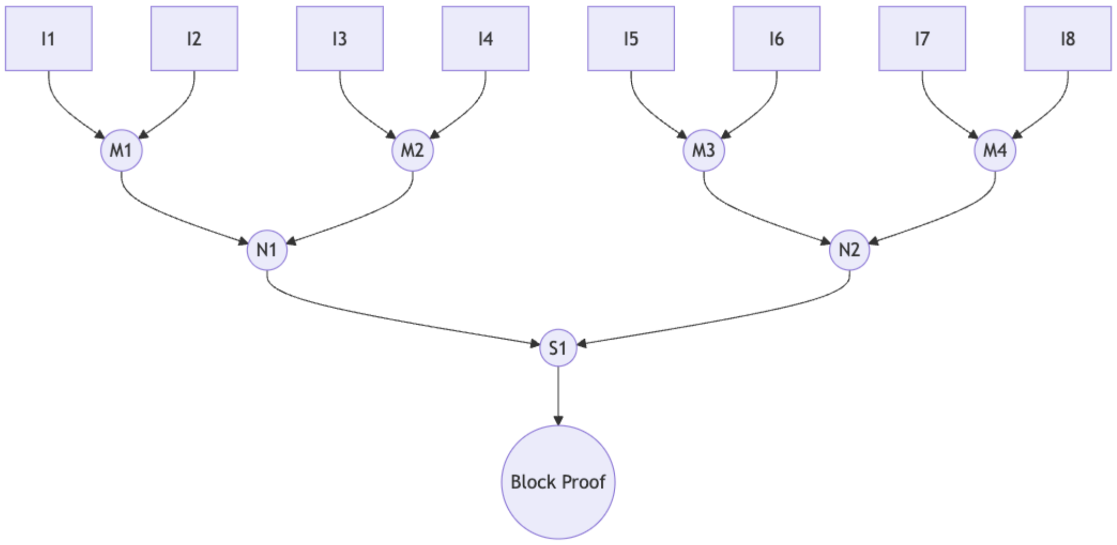
Once an intent is verified the state can be treated as optimistic for use inside the rollup.

After the block window has passed, the validator will generate the block transition proof which does the following:

- Computes the block state hash & updates the historical state tree.
- Progresses the state to the new block number

The block blob is then generated and uploaded to DA alongside the block transition proof. Every 500 blocks a checkpoint blob which archives the rollup state to DA. The Validator will then finalise the block in Sui, closing out the block and creating a new block object ready to receive new intents.

Orchestrator



The TEA-Orchestrator's primary duty is to coordinate the rolling up of the block proof. The Orchestrator will watch for the block finalised event from Sui to signal that a block is ready for rollup. Once the event has been received it will take the following actions:

1) Fetches the intent proofs m from DA and hydrates the pruned map data for each intent, re-executing the operations in memory to obtain the exact state path.

   Notation: *Let P_i represent the pruned map slice, generated for the live map of generic map $\mathcal{M}$ having applied the set of assertions and updates $\mathcal{A}_i$, inside the intent i.*

$$P_i = \mathrm{Prune}(\mathcal{M}_{i-1}, \mathcal{A}_i),\ (S_{i-1}, P_i) \xrightarrow{\pi_{i-1,i}} S_i$$

   Because the orchestrator supplies each worker with exactly the map it needs, every sub-proof still hashes to the same global state root, and the recursive composition is guaranteed to verify.

2) Passes each intent proof plus its pruned frontier to a worker in the Fizk Proving Network to generate the leaf layer of proofs for the block proof tree

3) When the m leaf proofs return the orchestrator continues the orchestration of workers to merge them in a binary tree of depth $\log 2m$. A complete block requires $2m - 1$ total proofs, but only $\log 2m$ synchronous merge rounds which

means the time-cost grows logarithmically while computational work is fully parallel in each round.

4) The final proof π_{block} and it's state is settled to Mina.

Observer(s)

Fizk's rollup is closed by design, so every fact about the outside world, other-chain events and the passage of time, must be imported by a trusted yet verifiable process. That task falls to the Observer TEA, maintaining core functions:

- **Bridge sentinel:** The enclave watches the target chains (Sui, Mina, USDC issuer) for deposit or withdrawal events. When a transaction has the required confirmations it will sign a payload for the user, enabling the corresponding mint or burn.
- **Settlement watcher:** After the Orchestrator settles a block proof to Mina, the Observer verifies the Mina receipt and settles the block on Sui. Until this flag is true the bridge remains locked, preventing fractional reserves.

Because the Observer influences escrow withdrawals, Fizk runs it as a quorum of enclaves. A threshold of signatures is required for any observation to become valid, so the compromise or key leak of a single Observer cannot prematurely unlock the bridge.

Data Availability (Walrus)

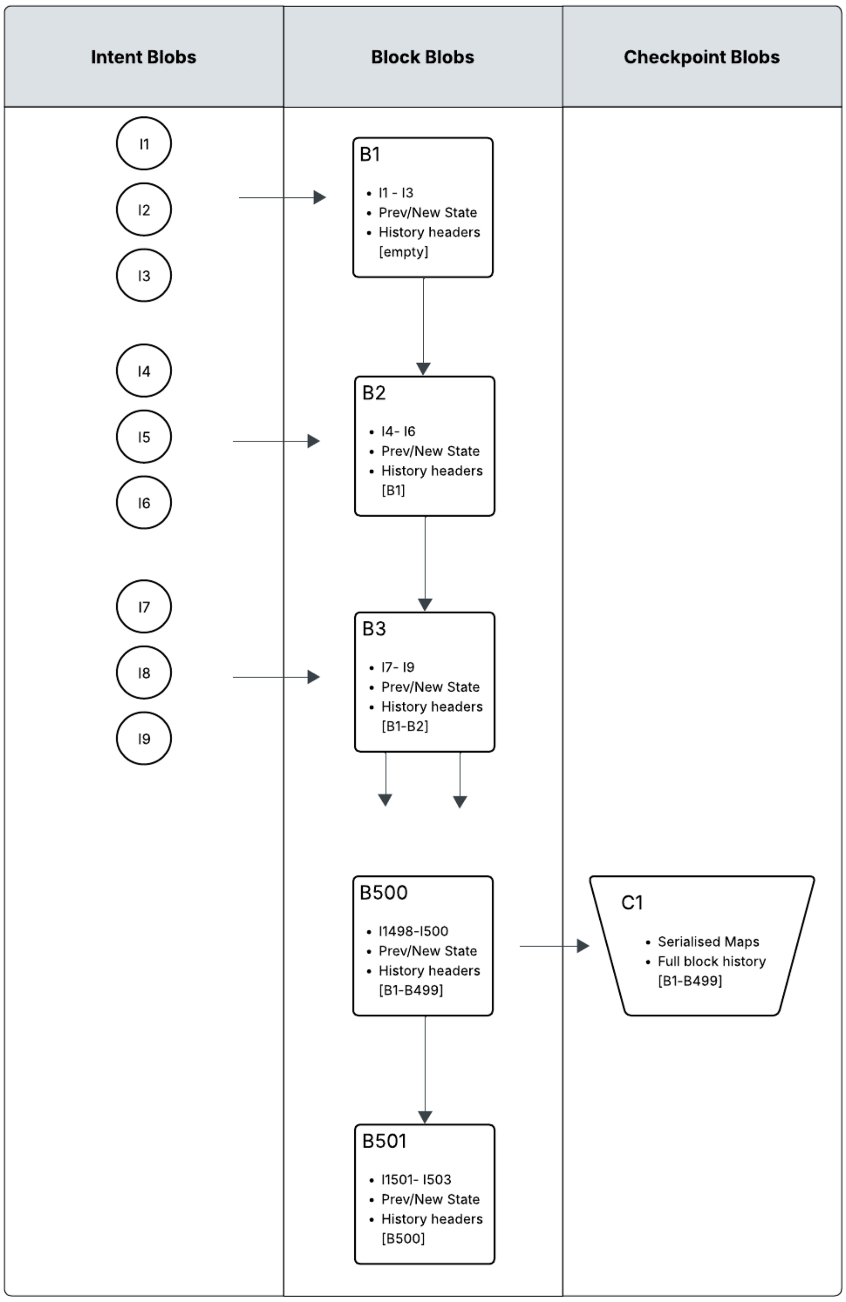
Fizk requires a performant, highly available and decentralised DA layer to function. Fizk needs to store, persist, and retrieve a whole array of data . It would be too expensive and uneconomical to store this data on Sui, and so we utilise an adjacent service designed for this purpose.

Walrus is Mysten Labs' decentralized blob-storage network purpose-built for Sui. Instead of full replication (≈ 25 × storage cost) or classic Reed–Solomon schemes that require downloading the entire file to heal a single lost slice, Walrus uses *Red Stuff*, a two-dimensional erasure code that keeps only ≈ 4.5 × redundant data yet can "self-heal" a missing shard with bandwidth proportional to the loss itself $O(|\mathrm{blob}|/n)$. Storage commitments, availability proofs, and committee-rotation messages are anchored in Sui smart-contracts, so any party can audit who is obligated to hold which chunk and for how long. The network tolerates ⅓ Byzantine storage nodes, supports asynchronous challenge/response (no timing-assumptions for proofs-of-storage), and rolls through epoch changes without interrupting reads or writes. For Fizk this means:

- Cheap, verifiable persistence for intent proofs, encrypted notes, archived serialised maps, and full block blobs;
- Retrieval guarantees strong enough for recursive proof reconstruction even if a third of the DA nodes vanish;
- Seamless integration with the same Sui event stream we already use for intent sequencing.

In short, Walrus gives the rollup a cost-effective, censorship-resistant data layer that scales with the application while keeping on-chain storage on Sui and Mina minimal.

Blob design and lifecycle



Intent Blobs

When a user submits an intent to the TEA-IntentSponsor they provide the zk-proof π_{intent} and any encrypted note payloads. The Sponsor packages those artefacts into a single *intent blob*, uploads it to Walrus, and returns the resulting blob-ID (a 32-byte content address). The user therefore needs no SUI or WAL balance, the upload fee is paid by the Sponsor and later reimbursed in zkUSD.

The intent transaction published on Sui carries that blob-ID. The TEA-Validator watches the stream, fetches each blob, checks that π_{intent} verifies and, if successful, admits the intent to the block.

During block closure the validator concatenates every verified intent blob plus the per-intent note ciphertexts into a single block blob, appends the block-transition proof, and stores the bundle in Walrus. Because the Validator has absorbed all intent data, the Sponsor can immediately recycle the space occupied by the original blobs: it pre-buys a fixed storage quota at the start of each epoch and reuses those bytes as intents roll forward, paying only the WAL storage fee once per epoch instead of per user action.

Block Blobs

As intents are processed into blocks by the Validator, it constructs the representation of the block into a JSON object. When the block is finalised, the blob is uploaded to Walrus and included in the block object in Sui.

Because each block blob contains both the pre-state and post-state roots and every intent delta, the entire rollup can be replayed from genesis using nothing more than Sui events plus the Walrus blobs they reference.

The block blob contains:

- **Block Data**: This holds the exact state transition: the block number, the preceding `FizkRollupState`, and hash, and the fully processed intent list with their associated map/state operations
- **BlockMetadata:** Links the most recent block into the historical chain. It stores the blob-id of the most-recent checkpoint, the height of that checkpoint and a rolling array of block headers created since then.

A block header is a lightweight summary of the block:

- Block Number
- Previous and New `FizkRollupState` state hashes
- Intent count
- Block proof
- Block blob id in walrus

As each new block is processed and uploaded, the chain of block headers grow linking to each block blob id inside of Walrus so the entire chain can be reconstructed from each subsequent block.

Checkpoint Blobs

Every five hundred blocks the Validator creates a checkpoint blob.

The blob bundles:

- the entire serialised snapshots of all the `IndexedMerkleMaps` `in the FizkRollupState`
- the full list of block data from the previous checkpoint and the delta since

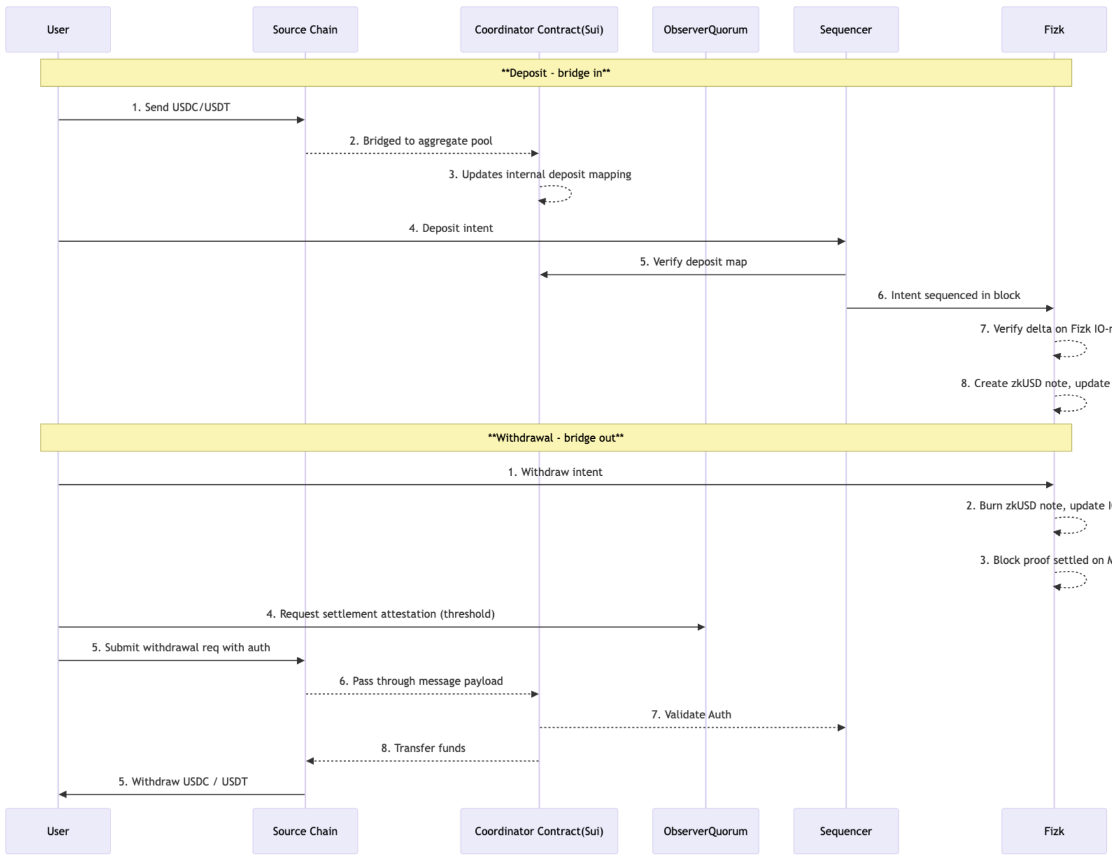
After uploading the file to Walrus the Validator writes its blob-ID into a single Checkpoint object on Sui.

The Sync routine:

1. Fetch the Checkpoint object, download the associated blob, and re-hydrate the maps to recreate the state at height k.
2. Read the current block object on Sui to learn the latest block-height k_{tip} and the blob-ID of the most recent block.
3. Starting with the oldest blob-ID listed after the checkpoint, walk the header chain forward, downloading each block-blob and replaying its map/state operations.
4. After replaying to k_{tip} the Validator holds the exact live roots published in FizkRollupState.

Because every checkpoint archives the headers accumulated since the prior one, each block-blob contains at most ~500 header entries. Older headers are retired from the per-block blobs once they migrate into a checkpoint, and the Walrus space they occupied can be reassigned. This rolling design keeps storage costs bounded while still letting any party reconstruct the entire rollup history from genesis with deterministic integrity checks.

Bridging



User
Source Chain
Coordinator Contract(Sui)
ObserverQuorum
Sequencer
Fizk
Deposit - bridge in
1. Send USDC/USDT
2. Bridged to aggregate pool
3. Updates internal deposit mapping
4. Deposit intent
5. Verify deposit map
6. Intent sequenced in block
7. Verify delta on Fizk IO-map
8. Create zkUSD note, update IO-map
Withdrawal - bridge out
1. Withdraw intent
2. Burn zkUSD note, update IO-map
3. Block proof settled on Mina
4. Request settlement attestation (threshold)
5. Submit withdrawal req with auth
6. Pass through message payload
7. Validate Auth
8. Transfer funds
5. Withdraw USDC / USDT

Fizk supports trust-minimized deposits and withdrawals of USDC and USDT through a cross-chain architecture anchored by Wormhole. Assets are routed into an aggregate liquidity pool on the Coordinator Contract deployed on Sui, allowing users to enter and exit the rollup from any supported source chain. This design ensures that stablecoins inside Fizk (zkUSDC, zkUSDT) remain fully fungible and portable, regardless of their origin chain.

The bridging process is coordinated via a two-phase model: intent-side proof generation and sequencing, and settlement-side attestation and release. A dedicated I/O Indexed Merkle map in the rollup state tracks accumulated deposits and withdrawals for each Fizk address, ensuring idempotent accounting and cross-chain consistency.

USDC Support

Fizk integrates both Wormhole and Circle's Cross-Chain Transfer Protocol (CCTP) to facilitate USDC bridging. CCTP allows native burn-and-mint operations across supported chains, making USDC fully fungible across its multichain footprint. This enables Fizk to maintain a single canonical USDC pool on Sui, while offering withdrawal endpoints to any supported chain — providing unified liquidity and seamless UX for users.

USDT Support

Unlike USDC, USDT lacks a native cross-chain burn-and-mint mechanism, making cross-chain fungibility infeasible. As a result, USDT deposits are only supported from Ethereum L1, where liquidity is deepest and canonical. Withdrawals are limited to Ethereum, preserving solvency and preventing mismatches. Future integration of DeFi liquidity routing (e.g., on-chain swap layers) may expand USDT support in a trust-minimized way.

Deposit Flow

1. **Initiation**: A user initiates a deposit by sending USDC/USDT to the Wormhole bridge on their source chain, along with a payload containing their Fizk Address.
2. **Routing**: Wormhole relays the deposit to the Coordinator Contract on Sui, which holds the aggregate stablecoin pool.
3. **Mapping**: The Coordinator updates the internal deposit ledger by incrementing the user's accumulated balance, keyed by their Fizk Address.
4. **Intent Submission**: The user generates and submits a `Deposit` intent to the Sequencer. This intent references their target deposit amount.
5. **Verification**: The Sequencer queries the Coordinator's on-chain deposit mapping and verifies that the user's claimed deposit amount matches the recorded balance.
6. **Rollup Execution**: If valid, the intent is sequenced into a block. During proving, the rollup verifies the delta against the internal I/O map, and creates a corresponding zkUSDC/zkUSDT note, updating the I/O map accordingly.

This flow ensures atomicity between deposit verification and note issuance, while enforcing cross-chain correctness through a deterministic, auditable process.

Withdrawal Flow

1. **Burn & Record**: The user initiates a withdrawal by submitting a `Withdraw` intent. This burns one or more zkUSDC/zkUSDT notes and updates the I/O map with their accumulated withdrawal total.
2. **Finality**: Once the block is proven and settled to Mina, the I/O map root becomes immutable and globally verifiable.
3. **Attestation**: The user signs a withdrawal request payload (including their Fizk Address and destination address) and submits it to the Observer Quorum. Upon reaching threshold, the Observers return a signed attestation over the I/O map state.
4. **Release Request**: The user submits the attestation payload to the Wormhole contract on their desired destination chain, which forwards the request to the Coordinator on Sui.
5. **Validation & Execution**: The Coordinator invokes the Sequencer to verify the attestation and ensure the user has sufficient withdrawal credit. If valid, funds are released to the destination address on the user's chosen chain.

This model ensures that:

- Withdrawals are cryptographically tied to actual zk note burns.
- Final settlement cannot be faked or replayed across chains.
- Attestation-based releases are rate-limited and threshold-controlled.

By separating the sequencing layer from the final release mechanism, Fizk minimizes cross-chain trust assumptions while maintaining solvency, auditability, and exit safety.

Governance

Fizk is governed by a two-tier system that keeps decision-making agile while leaving ultimate authority with token holders.

General Assembly

Everyone who stakes the Fizk Network Token (FIZK) belongs to the General Assembly, and voting power is strictly proportional to the amount staked. Unstaking is permitted at any time, but a 5% exit fee on the staking position discourages short-term speculation and removes a small amount of supply, giving the token a mild deflationary bias. In return for locking their tokens, stakers share a portion of the transfer fees and gain the right to steer the protocol. After unstaking, there is a delay period before the users can withdraw their tokens, this is a defensive measure to prevent hostile proposals.

Council

From among themselves, Assembly members elect up to ten Council representatives. The Council carries out routine governance. It reviews improvement proposals, adjusts risk parameters, and signs off operational budgets. Council members are compensated from the FORP so they can devote time to the role.

Checks and balances

The Assembly can dismiss any councillor, veto any Council decision, or call a full re-election at will. Every Council-approved proposal enters a mandatory time-lock before execution, giving the Assembly a clear window to coordinate a veto if a decision appears misaligned with community interests.

A small compensated Council keeps the protocol responsive, while the staked community retains an unambiguous power of override, ensuring that Fizk remains both nimble and owner-controlled.

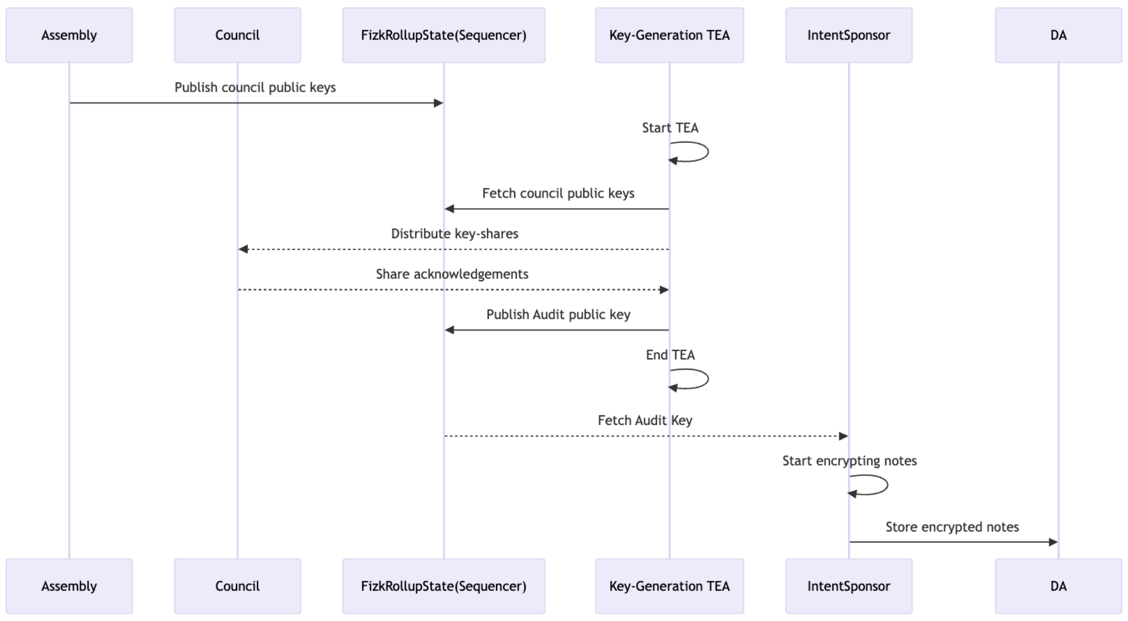
Audit

Fizk seeks to give users cash-like privacy while preventing the rollup from becoming a no-questions-asked mixer.
The protocol therefore supports selective auditability, every UTXO note is encrypted twice, once with the recipient's viewing key and once with an Audit Key controlled by governance.
If the Council detects, or is alerted to, suspicious activity (e.g., a large hack funnelling funds through Fizk), it can launch an Audit Report to trace flows and, if desired, blacklist addresses.

Audit Key

The Audit Key's public half is maintained in the `FizkRollupState`. The private half is split with Shamir secret-sharing among the Council members. A dedicated Audit Key service TEA spins up, ad-hoc, for key generation, rotation, or an audit session.

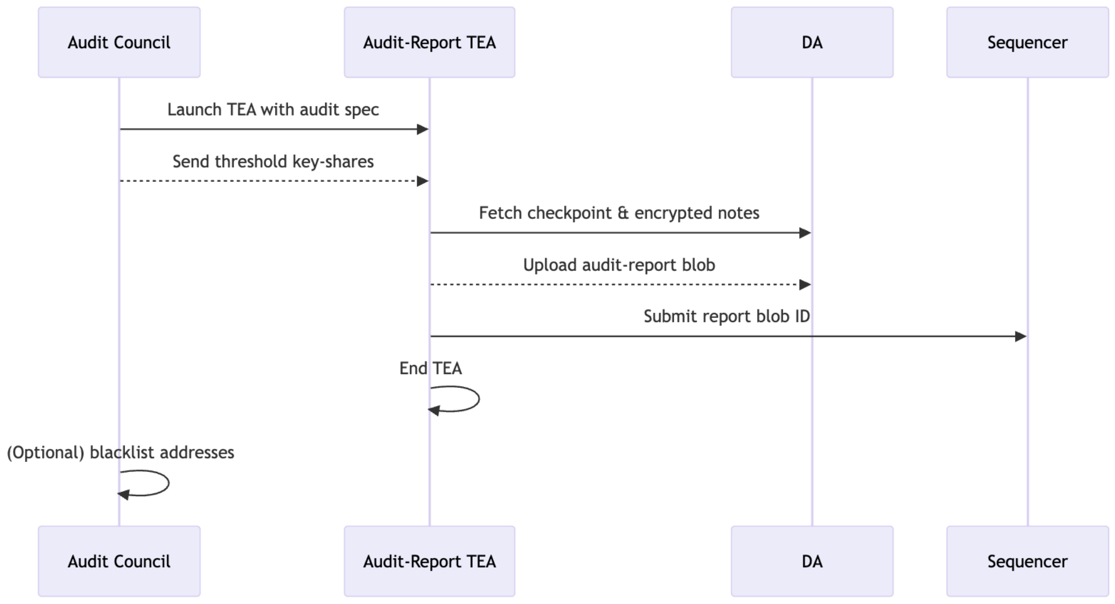
Key Generation



1. The Assembly elects the first Council and publishes their public keys to the rollup state.
2. A Key-Generation TEA launches, fetches those keys, creates a fresh Audit private key in RAM, and splits it into shares.

3. Each councillor signs an acknowledgement; once the threshold is met the TEA generates an intent proof and posts the Audit public key to the Sequencer.
4. Once the Audit Key is in state the IntentSponsor fetches the new key and begins encrypting the notes.

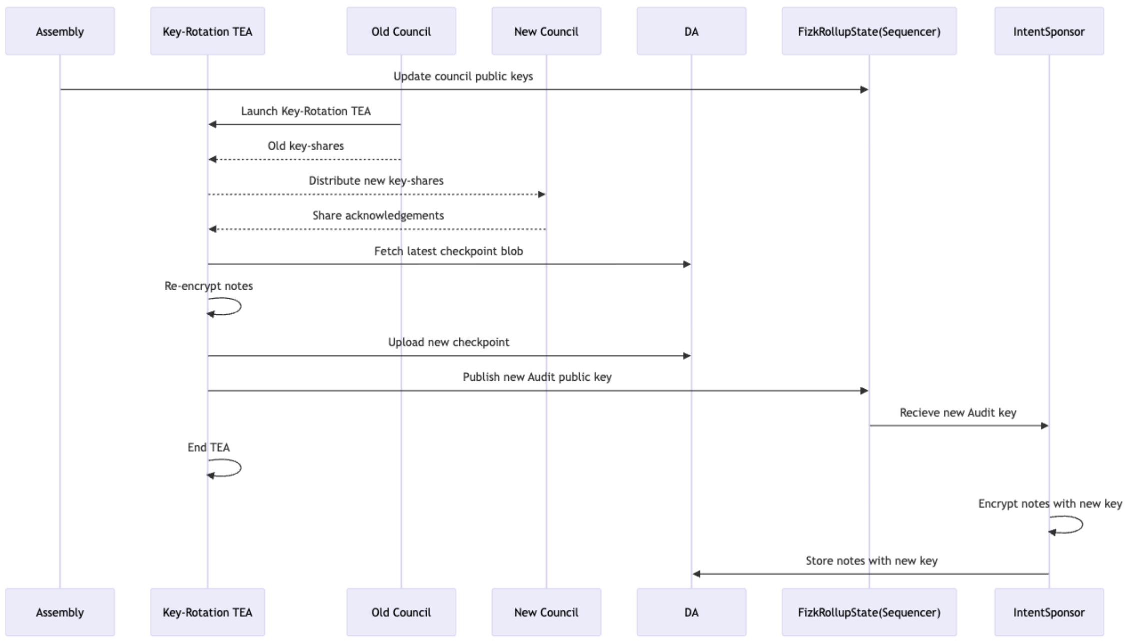
Audit Report



When an investigation is requested:

1. An Audit-Report TEA is launched with a spec: addresses or note commitments to trace and the time-window to inspect.
2. Councillors send their key-shares; upon reaching threshold the enclave fetches the latest checkpoint blob from Walrus.
3. The TEA decrypts only the notes in scope, reconstructs the flow graph, and writes a report blob back to Walrus, posting the blob id to the sequencer.
4. If warranted, the Council may immediately submit a blacklist proposal; it executes without delay but can be overturned later by a full Assembly vote

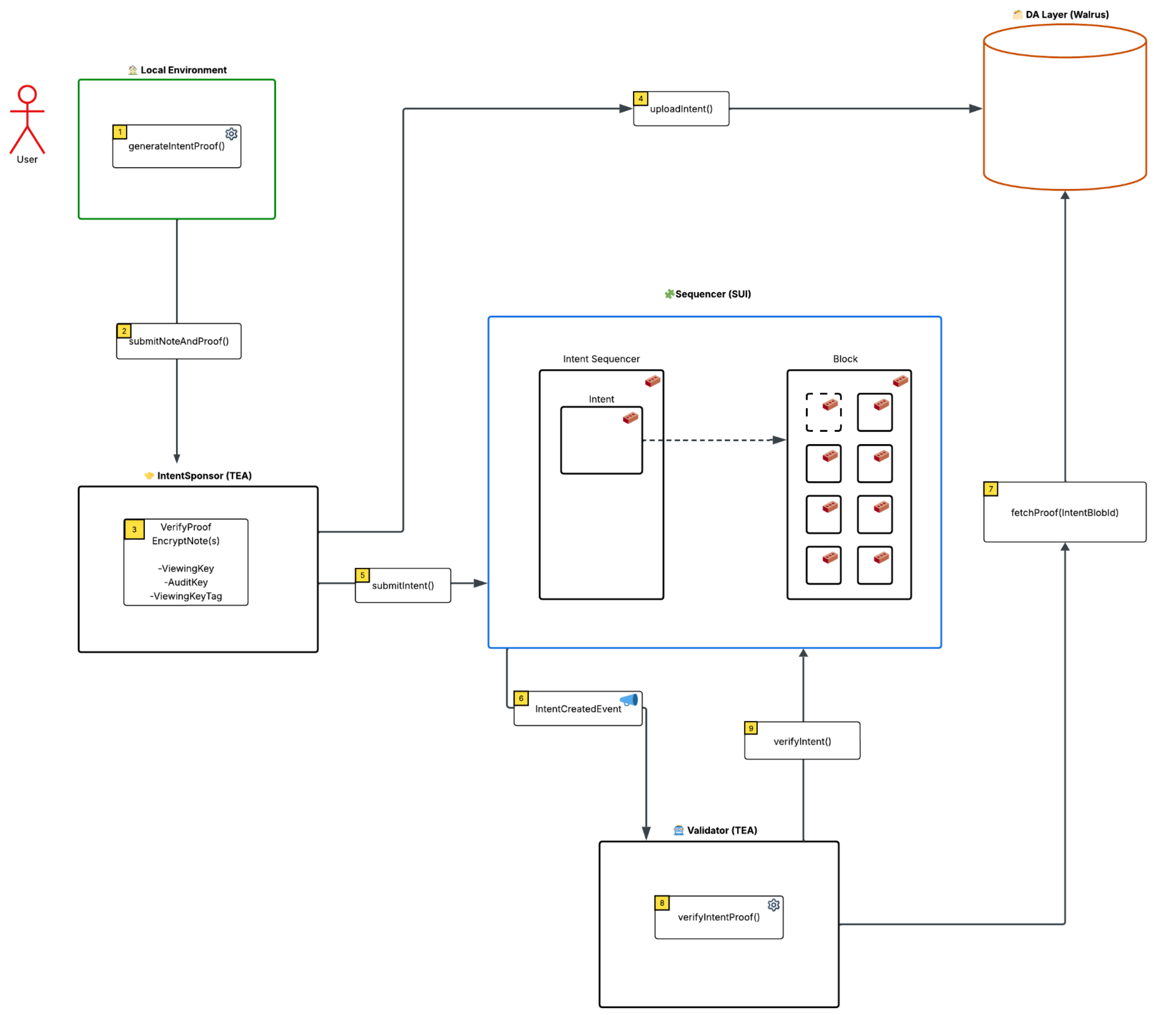
Key Rotation



The tricky part is effectively rotating the Audit Key at any point the council changes. The key rotation process must update each historical note since genesis:

1. After the Assembly seats a new Council, a Key-Rotation TEA starts.
2. Out-going councillors supply the old key-shares; in-coming councillors supply signatures to receive new shares. The enclave now holds both keys in memory.
3. It downloads the latest checkpoint blob from Walrus, and sequentially decrypts and re-encrypts every note with the new Audit Key, and constructs a new version of the same checkpoint blob, but with every note replaced with the new key.
4. When re-encryption catches up to the frontier, the TEA posts the new Audit Key to the sequencer. The IntentSponsor then switches to the new key, and the TEA shuts down.

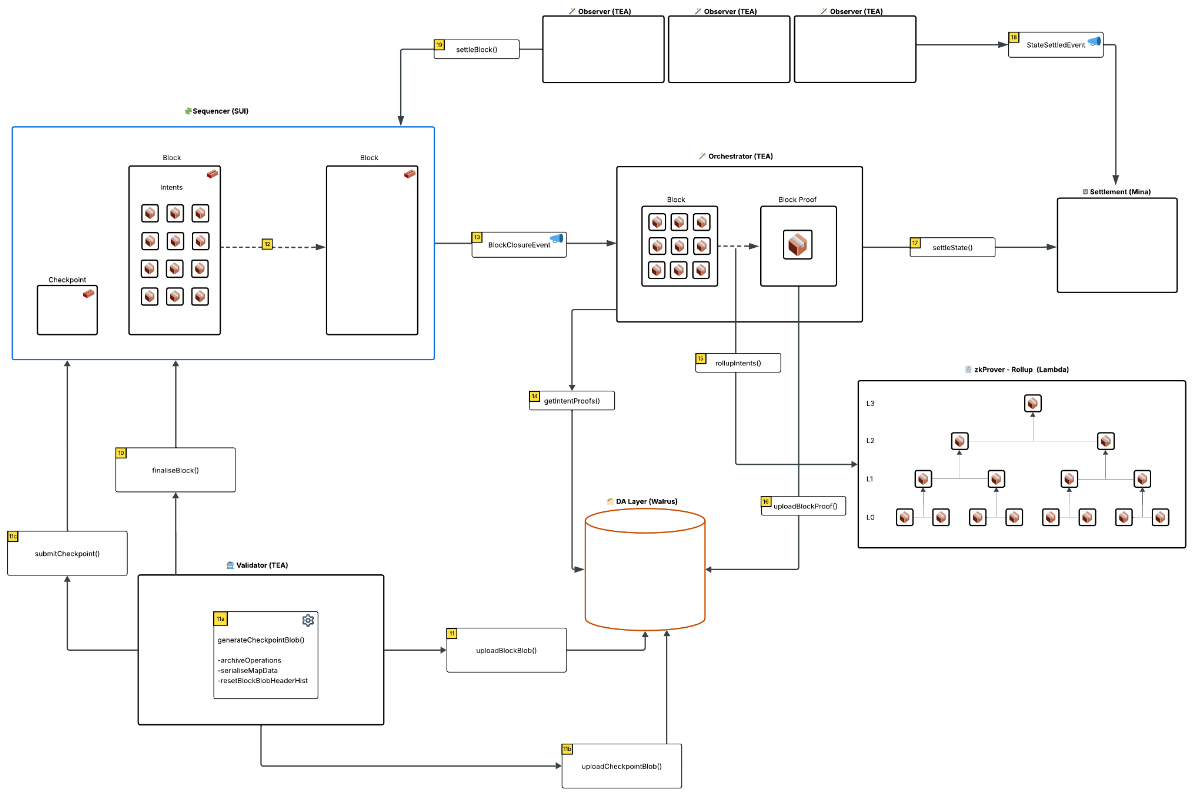
Intent Lifecycle



1) The user generates the intent proof client-side.
2) The user then submits the proofs alongside any accompanying notes to the IntentSponsor.
3) The IntentSponsor verifies any commitments against the provided proof and encrypts the notes with the viewing key and the audit key.
4) The IntentSponsor uploads the intent proof with the encrypted notes and viewing key tag to Walrus.
5) Then it submits the intent to the sequencer with the reference to the blob id on Walrus.
6) The sequencer emits an IntentCreatedEvent.
7) The Validator fetches the proof from Walrus.
8) The Validator verifies the proof and applies the map operations to its local in memory map.

9) The Validator verifies the intent in the sequencer, marking the optimistic state change and moving the intent object into a block.

Block Lifecycle



10) Once the block window has completed the Validator will create a block transition proof and submit a FinaliseBlock tx to the sequencer.

11) Checkpoint:

    a) If 500 block have passed since the last checkpoint, the Validator will generate a checkpoint blob which archives the block history and serialises the map data

    b) The checkpoint will be uploaded to Walrus

    c) The Validator will submit the new checkpoint reference to the sequencer

12) The new block object will be created on Sui, ready to receive new validated intents

13) As the older block is closed, a BlockClosureEvent will be published by the sequencer, which the Orchestrator will pick up

14) The Orchestrator will prepare the data necessary for generating the block proof. This will include pruned map state for each S_i and the intent proofs for each rollup tx
15) The Orchestrator will kick off the block proving through the zkProver. To begin with that will be a Lambda proving service, to be later offloaded to a decentralised proving network.
16) The block proof will be uploaded to DA
17) The Orchestrator will settle the proof to Mina
18) The Observers will watch for the StateSettleEvent from Mina and then each will submit the settleBlock tx to the sequencer
19) Once quorum has been reached the block will be marked as settled in the sequencer, finalising the state and unlocking funds for bridging.

FizkRollupState

The `FizkRollupState` is a complex data structure that encompasses all the state that makes up the Fizk Protocol. The state is made up of a series of merkle roots, protocol parameters and governance, enclave, and regulatory config.

Indexed Merkle Map

An `IndexedMerkleMap` is a key-value merkle storage primitive data structure provided by o1js and is used for storing data in the rollup. It provides several benefits over a sparse merkle tree:

- **Proof-friendly:** Look-ups and updates cost ~4–5 × fewer constraints than a sparse Merkle tree of equal height, so each operation stays cheap inside the circuit.
- **Serialisable:** The map can be written to disk as a breadth-first list of `(index, hash, height)` tuples. The protocol utilises this feature to archive map data in the data availability (DA) layer, and anyone can stream it back and reconstruct the exact same tree in $O(nodes)$ time, with no re-hashing required.
- **Prunable:** Any in memory tree operator can serve a tiny pruned tree containing just the leaf (or low-leaf) and its sibling hashes; a wallet downloads this $O(\log N)$ slice, recomputes the same and proves inclusion / exclusion without holding the full tree.

Inside `FizkRollupState` we keep only the map roots while the data persists in the DA layer. This design gives Fizk (i) low-cost proofs on chain, (ii) a compact, verifiable audit trail off chain, and (iii) efficient client-side proving speed on ordinary hardware.

ZkUsd State

ZkUsd Maps

The ZkUsd Maps $\mathcal{M}_{\text{zk}}$ are the fundamental merkle maps by which we track the transfers and balances of respective zkUSDC and zkUSDT in the rollup. The maps contain two core components, UTXO note commitments and nullifiers.

UTXO note commitments

Fizk adopts a Bitcoin-style `unspent-transaction-output` (UTXO) model. Each transaction consumes one or more existing notes and creates new, unspent outputs. A valid output is spendable exactly once enforced by the zkPrograms.

Notation: *Here H is Poseidon - a snark friendly hashing algorithm, a_i is the amount, PA_i is the payment address of the recipient, s_i is the secret, which the nullifier is eventually derived from, n_i is the nonce, a random 254 bit number that introduces entropy into the note. All C_i values form the keys that make the note commitments in the $\mathcal{M}_{\text{zk}}$.*

$$C_i = H(a_i \,\|\text{PA}_i|\, s_i \,\|\, n_i).$$

It is impossible to tell anything about a particular note only knowing the commitment. However the presence of a note in the tree can be asserted inside the proof by someone who knows the values that were used in hashing the note.

For the recipient to be able to spend a note, the pre-hashed data needs to be communicated to them. Therefore the note is encrypted with the recipient's viewing key and published alongside the transaction. Each user's wallet syncs with the network by reading the stream of note commitments, attempting to decrypt each one with their key, and if successful they know they have a spendable note.
This process is optimized by maintaining a short "view tag" alongside each encrypted note. The view tag is a 1 byte value derived from the recipient's viewing public key, allowing wallets to quickly filter notes before attempting decryption. Instead of decrypting every note in the network, a wallet only attempts to decrypt notes with a matching view tag, reducing the computational load by a factor of 256. This optimization maintains full privacy while dramatically improving wallet synchronization performance, especially as the network grows to millions of transactions.

A Fizk payment address PA_i consists of the viewing key vk_{pub} and the spending key sk_{pub}. The vk_{pub} is derived from the sk_{pub}, which is a 32 byte Schnorr key on the Pallas elliptic curve (part of the Pasta cycle).

In order to spend a note a user must sign with the corresponding spending key. This format allows users to share the viewing key with interested parties, who are then able to view the wallets balance and transfers, while retaining control over spending alone.

Nullifiers

When a note is spent, the spender generates a nullifier $\nu_i = H(r_i \,\|\, s_i)$, which we insert into $\mathcal{M}_{\mathrm{zk}}$. As ν_i depends on the secret of the note s_i and the r component of the spending signature r_i that is used when spending the note, it cannot be linked back to the original commitment, preserving privacy and irrevocably marking the note as spent to prevent double-spending.

ZkUsd I/O Map

To facilitate bridging of USDC and UDST into and out of the rollup the protocol maintains an input/output(I/O) map $\mathcal{M}_{\mathrm{iozk}}$. The $\mathcal{M}_{\mathrm{iozk}}$ tracks accumulated deposits and withdrawals, decorated by the destination chain.

zkUSD Transfer Fee

There is a flat fee that is paid on the transfer of zkUSD stables. It funds the operational expenses of the network and is directed to the Fizk Operational Reserve Pool(FORP). The fee is controlled by governance.

Enclave Config

For each operational TEA type, the protocol maintains the hash of the generated PCR's

$$\mathrm{Root}_{\mathrm{EC}} = H\big(\mathrm{PCR}_0 \,\|\, \dots \,\|\, \mathrm{PCR}_3\big)$$

paired with the K_{enc} of the enclave.

Fizk Network Token (FIZK)

The protocol uses two different maps for FIZK. The Fizk Network Token Map $\mathcal{M}_{\text{fnt}}$ and the corresponding I/O map, $\mathcal{M}_{\text{iofnt}}$. As FIZK doesn't need to be confidential, we track balances and transfers through a simple address-amount in the $\mathcal{M}_{\text{fnt}}$. The $\mathcal{M}_{\text{iofnt}}$ tracks accumulated deposits and withdrawals, decorated by destination chain, to enable bridging capabilities.

Governance

For the operational governance of the Fizk protocol, the `FizkRollupState` maintains the following state:

- **Fizk Operational Reserve Pool (FORP):** This is the treasury of the protocol which manages the balance of the reserve pool. Spendable by governance or the council.
- **Council Seats:** The list of keys associated with the sitting council, held as a merkle list.
- **Assembly Proposal Threshold:** A percentage required to reach quorum on governance proposals.
- **Assembly Proposal Veto Threshold:** The minimum assembly votes required to veto the execution of a proposal.
- **Proposal Execution Delay:** The period of time a passed proposal must wait before being executed - provides sufficient veto time for the assembly.
- **Proposal Snapshot Validity**: The period of time a proposal is valid for once created.
- **Mina Settlement Key:** The pub key that has the power to settle the state to mina, which will be the TEA-Orchestrator ephemeral key.
- **Global Gov Reward Index:** An index tracking mechanism for fees to FIZK stakers
- **Total Amount Staked:** Total FIZK staked - used in calculations for the reward index
- **Gov Tx Fee Percentage:** The percentage of the tx fee that is redirected to token stakers.

Governance also uses two maps - Proposal Map $\mathcal{M}_{\text{govpl}}$ and the Stake Map $\mathcal{M}_{\text{govstk}}$ to manage the governance proposals and staked FIZK respectively.

Regulatory State

The protocol stores the audit public key and a Blacklist which provides the capability to freeze funds following audit reports.

Block State

Finally, the protocol uses some metadata on the state for processing and validation purposes.

- **previousBlockClosureTimestamp:** The timestamp the previous block was closed
- **blockNumber:** The block height of the current block
- **intentSequence:** The intent number that is being processed
- **historicalStateMap:** The root of the historical state which is a rolling map of blockNumber to hashed protocol state, this unlocks the powerful capability of the historic state proofs which is important for governance snapshotting.

Defensibility and Trust Assumptions

Fizk leans on three mutually-reinforcing safety pillars:

- **Mathematical soundness**: every state transition is proven inside a recursive Kimchi → Pickles chain; if a proof verifies, the transition is valid by construction.
- **Hardware isolation**: all Trusted Execution Agents (TEAs) run in AWS Nitro enclaves whose PCR-anchored attestations are recorded on-chain.
- **Segregated duties**: each TEA type performs a single, auditable task, so a key leak in one enclave cannot silently corrupt the whole system.

Baseline Threat Model

- **Hypervisor honesty:** We accept the standard Nitro assumption that the underlying KVM hypervisor enforces memory and CPU isolation correctly.
- **Council quorum**: At least ⅔ of Council key-shares remain honest; this is the same threshold that already signs every Audit-Key operation.
- **Observer quorum:** Price-feeds and bridge attestations need independent TEA-Observer signatures.
- **Cryptography:** We rely on the unforgeability of the Kimchi proof system

No other component, including Sui validators or the Walrus storage nodes, need to be trusted for correctness, only for liveness.

The rollups canonical chain is

$$S_i \xrightarrow{\pi_{i,i+1}} S_{i+1} \cdots S_{k-1} \xrightarrow{\pi_{k-1,k}} S_k$$

And the only way to transition from S_i to S_{i+1} is by supplying a proof $\pi_{i,i+1}$ the Kimchi circuit's constraints hold.

Even if an attacker steals every secret key inside the IntentSponsor, Validator, and Orchestrator, they still cannot fabricate a proof that violates the constraint system. Mina will therefore reject the block, the council will observe the failure, recycling the TEAs and restoring from the last good block. No counterfeit notes and no fake vault deposits.

Compromised component	Immediate effect	Detection path	Resulting damage
IntentSponsor	Could publish malformed intent blobs	Validator would catch the invalid intent proof and reject the intent	Potential network interruption as keys rotated No state damage - the intent would be rejected by the validator.
Validator	Could accept a fake intent or build a fraudulent block blob	Orchestrator must merge intent proofs; an invalid delta makes the block proof fail.	Network interruption as Council rotates keys after detection. Intents must be replayed into the reset block
Orchestrator	Could try to settle an unprovable block on Mina	Mina settlement contract rejects settlement; Observers see the missing StateSettleEvent.	None; Council rotates the Orchestrator key, and orchestrator starts producing the valid block proof Interruption to bridging activities as settlement is delayed
Observer (single)	Could feed a bad price of bridge attestation	Requires signatures; a single leak cannot reach quorum. Bad data would be detected by interested parties	None; Council rotates the faulty observer key.
Observer quorum	Could lie about collateral prices or fake deposits.	Rollup monitored for suspicious activity ; Council can emergency-pause.	Extensive damage possible

Because duties are strictly compartmentalised, no single enclave compromise can steal money or falsify state; at worst it causes a denial-of-service that surfaces in a short time frame.

As all private keys live solely in the enclave RAM, breaking a key requires both an undisclosed hypervisor exploit and an application bug, an event that is extremely low probability.

Emergency Recovery & Governance Override

The council will retain the ability to recycle the keys of any of the TEAs on the sequencer. As soon as a compromise is detected the enclave can be rebooted and the keys recycled. Because proof settlement to Mina is the final check, the worst-case scenario is a temporary freeze while the Council reboots the network.

Conclusion

Fizk delivers a purpose-built confidential payment layer that closes the long-standing gap between privacy and regulatory accountability in stablecoins. By embedding every state transition inside a recursive Pickle/Kimchi proof chain and anchoring those proofs to Mina, the protocol gives mathematical finality to all balances while keeping transaction details confidential. A high-throughput intent layer on Sui, persisted by Walrus data availability and coordinated by Nitro-enclave Trusted Execution Agents, scales this cryptographic assurance to thousands of intents per block without introducing new trust assumptions.

Selective auditability is built into the protocol. Dual encryption of each note under user viewing keys and a Council-controlled Audit Key lets governance trace flows when necessary, yet prevents routine surveillance. The two-tier governance model of the Assembly and the Council couples day-to-day agility with a community veto, while emergency key-rotation paths and observer quorums limit the blast radius of any single compromise.

Through this architecture, Fizk offers more than just transactional privacy, it provides a dependable and scalable foundation for confidential payments in the digital economy.

By preserving user privacy without compromising regulatory auditability, Fizk paves the way for a new category of stablecoin infrastructure: one that restores financial privacy to web3, unlocks new use cases for businesses, and meets the compliance needs of institutions. Confidential payments are a necessity, and Fizk is building the infrastructure to make them standard.