

# Hindsight litepaper

Confidential USDC payments on Solana whose amounts stay sealed after Q-Day: what the mechanism is, what one payment costs, and who sees what.

October 2026      15 min read

---

## Contents

The problem

How Hindsight works

What one payment costs and who sees it

Roles

Fees

\$HIND

Components

What exists today

Hindsight provides confidential USDC payments on Solana. Each amount is a Pedersen commitment on chain, and its opening reaches only the payee and the pool's auditor, sealed in X-Wing envelopes (X25519 + ML-KEM-768). This paper covers the mechanism, the cost of one payment, what each party can see, and what ships.

## The problem

Every validator and archival RPC node keeps a full copy of the ledger. An adversary can record every ciphertext today at the cost of disk space and decrypt it once a cryptographically relevant quantum computer exists. Most retail payment amounts lose their value within weeks. Salaries, supplier prices in multi-year contracts, treasury positions, legal settlements and advisory fees stay sensitive for a decade or longer. The EU's coordinated roadmap requires data that must stay confidential for ten years or more to be protected against quantum attack by the end of 2030. The draft NIST IR 8547 disallows all quantum-vulnerable public-key algorithms after 2035.

Token-2022 confidential transfers hide amounts with twisted ElGamal. A ciphertext is a Pedersen commitment  $C = xG + rH$  plus a decryption handle  $D = rP$ , where the public key is  $P = s^{-1}H$ . The key holder computes  $C - sD = xG$  and then solves a small discrete log to recover  $x$ . That last step is why transfer amounts are capped at 48 bits and split into 16-bit and 32-bit halves.  $P$  is stored on chain, so Shor's algorithm can recover  $s$  from it, and the handle then gives up  $x$  through a search a classical computer finishes easily. Each commitment carries three handles: source, destination and auditor. A single discrete log on a mint's auditor key therefore reveals every transfer amount that mint has ever carried.

## How Hindsight works

Hindsight keeps the Pedersen commitment, which is perfectly hiding, and the classical range proof. It drops the decryption handle. The opening of each amount goes to the payee and the auditor inside post-quantum envelopes, and nobody is credited with a commitment they cannot open. Confidentiality has to last indefinitely, so it rests on perfect hiding and ML-KEM. Soundness only has to hold when a transaction is verified, so it rests on classical range proofs and Ed25519 and moves with Solana's own post-quantum migration.

### Commitments

Amounts are unsigned 64-bit USDC base units. Commitments live in Ristretto255.  $G$  is the Ristretto basepoint, and  $H$  is the Ristretto map of SHA3-512 applied to the compressed  $G$ , the same generators `solana-zk-sdk` uses. Each participant holds one `ConfAccount` per pool, at seeds `["acct", pool, owner]`, and its `balance_commitment` field holds  $C_{\text{bal}}$ . The program updates  $C_{\text{bal}}$  through the `curve25519` syscalls (add 521 CU, multiply 2,208 CU). It holds no secrets and decrypts nothing.

A payment of  $v$  is a note with a fresh random blinding  $r$ . Writing  $b'$  for the payer's remaining balance:

$$C_{\text{note}} = vG + rH, \quad C'_{\text{bal}} = C_{\text{bal}} - C_{\text{note}}, \quad v, b' \in [0, 2^{64})$$

`transfer` requires the instruction just before it to be the native `VerifyBatchedRangeProofU128`, with commitments exactly  $[C_{\text{note}}, C_{\text{bal}} - C_{\text{note}}]$ , bit lengths  $[64, 64]$  and unused context slots zero. Otherwise it fails with `ProofContextMismatch`. `deposit` sets  $C_{\text{bal}} \leftarrow C_{\text{bal}} + aG$  for a public amount  $a$ . `withdraw` requires `VerifyBatchedRangeProofU64` on  $C_{\text{bal}} - aG$  before the vault pays out. No discrete log is ever solved, so amounts are full 64-bit values with no split.

`open_account` starts a balance at the public commitment to zero  $r_0H$ , where  $r_0$  is the SHA-256 of `hindsight:initial-balance-blinding:v1` `pool` `owner` `key epoch` `retry counter`. This keeps a zero balance provable after a full withdrawal. Each balance change must name the current `expected_sequence`, so a change built on an outdated sequence fails with `StaleSequence`. Only the owner can change  $C_{\text{bal}}$ . Each change also replaces the account's 68-byte self-record, which is the balance and its blinding under AES-256-GCM with a key that stays in the owner's client.

## Envelopes

The payer builds a 72-byte plaintext from the opening  $(v, r)$  and a 32-byte memo. It seals the plaintext twice, once to the payee's X-Wing key and once to the auditor's. Each key is a 1,216-byte encapsulation key in a `KeyRecord` account (seeds `["kem", owner, epoch]`), written by `register_key`. Encapsulation yields a shared secret  $ss$  and a 1,120-byte ciphertext  $ct$ . The AES-256-GCM key and nonce are derived from them, with `dom` = `hindsight:envelope:aes256gcm:key-nonce:v0:`:

$$k \parallel n = \text{HKDF-SHA256}(ss; \text{salt} = \text{SHA256}(pk \parallel ct); \text{info} = \text{dom} \parallel \text{AAD}), \quad |k| = 32, |n| :$$

$$\text{AAD} = \text{hindsight:envelope:v0:} \parallel \text{role} \parallel \text{pool} \parallel \text{sender} \parallel \text{payee} \parallel \text{seq} \parallel \text{epoch} \parallel C_{\text{note}}$$

The role byte is 0 for the payee and 1 for the auditor. Both envelopes name the payee's wallet, and each names the epoch of the key it was sealed to. Every envelope has the same layout:

| Offset | Length | Field                                      |
|--------|--------|--------------------------------------------|
| 0      | 1,120  | X-Wing ciphertext (ML-KEM-768 + X25519)    |
| 1,120  | 8      | amount $v$ , u64 little-endian (encrypted) |
| 1,128  | 32     | blinding $r$ (encrypted)                   |
| 1,160  | 32     | memo, padded (encrypted)                   |
| 1,192  | 16     | AES-256-GCM tag                            |

The envelopes travel in the instruction data of `transfer`, which rejects any other length with `InvalidEnvelopeLength`. The note stores  $\text{SHA256}(\text{env}_{\text{payee}} \parallel \text{env}_{\text{auditor}})$  in `envelope_hash`, so any copy can be checked against the chain.

## The note inbox

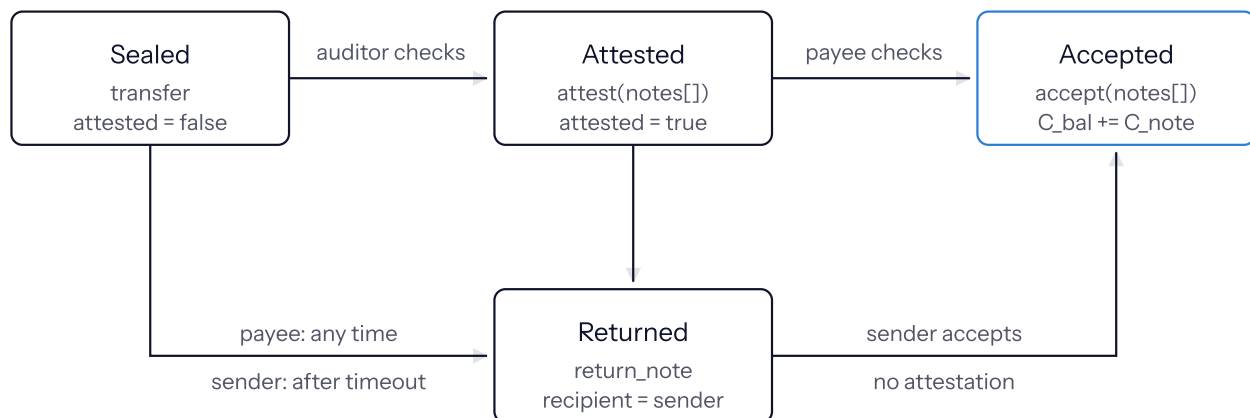
`transfer` creates a `Note` at seeds `["note", pool, sender, seq]`. The note holds the sender and recipient, the original recipient,  $C_{\text{note}}$ , the envelope hash, the creation slot, both key epochs, and the flags `attested` and `returned`. Incoming value waits in notes until the payee accepts it. The payee's client is notified, fetches the envelope pair, checks the pair's hash against `envelope_hash`, decrypts its envelope and checks

$$vG + rH \stackrel{?}{=} C_{\text{note}}.$$

Only after that check does the client call `accept`. `accept` adds  $\sum_i C_{\text{note},i}$  to the payee's balance, closes the notes and refunds their rent to the original senders. Envelopes are not proven correct, but no payee accepts what it cannot open, so a malformed envelope costs only its sender. The payee calls `return_note`, which sets the recipient back to the sender, and the sender, who already holds the opening, accepts the note into its own balance. A sender can also return an unaccepted note once `return_timeout_slots` have passed since the note's creation slot.

## The auditor's attestation

Each pool names one auditor (`Pool.auditor`) and its key epoch (`auditor_kem_epoch`). The auditor service does not trust the relay or the indexer. It fetches the confirmed `Note`, `Pool` and `KeyRecord` accounts itself, checks their owners, addresses, epochs and the ordered envelope hash, decrypts its envelope and checks the same equation the payee checks. Notes that pass go into batched `attest` calls, which set `attested = true`. A note that fails to open or match is quarantined and never attested. When the pool sets `attest_required`, `accept` fails with `AttestationRequired` for any unattested note, so every amount credited in that pool has been read by the auditor. A returned note is the exception, since its sender accepts it back without attestation.



The life of a note. `transfer` creates it sealed, `attest` marks it as read by the auditor, and `accept` adds it to the payee's balance and closes it. A note that does not open goes back to its sender, who accepts it without attestation.

On honest flows, the openings of all balances and outstanding notes sum to the public principal, and the vault holds at least that amount:

$$\sum_j b_j + \sum_{i \in \text{notes}} v_i = \text{Pool.principal} \leq \text{vault.amount}$$

`deposit` and `withdraw` move the principal together with the tokens. `transfer`, `accept` and `return_note` leave it unchanged. Both native features the design depends on are live on mainnet: the ZK ElGamal Proof program, re-enabled at epoch 982 (slot 424,224,000), and v1 transactions of up to 4,096 bytes, live since epoch 1035 (2026-09-15).

## What one payment costs and who sees it

### Bytes and compute

A transfer is one v1 transaction, and the payer is its only signer and fee payer. The figures here come from a measured run on an Agave 4.3.0 validator with the program deployed and native proof verification on. In that run a transfer took 4,024 of 4,096 bytes and 226,570 CU against a 300,000 CU limit, and 200,000 of those CU are the U128 proof verification. The rows are the design budget, and the last one is the measured remainder.

| Part                                                                     | Bytes        |
|--------------------------------------------------------------------------|--------------|
| Signature, header and config                                             | 90           |
| 9 account keys and blockhash                                             | 320          |
| Range-proof instruction: 736-byte proof, 264-byte context, discriminator | 1,001        |
| <code>transfer</code> fields: commitment, sequence, self-record          | 110          |
| Payee envelope                                                           | 1,208        |
| Auditor envelope                                                         | 1,208        |
| Compute budget and the rest (measured remainder)                         | 87           |
| <b>Measured total</b>                                                    | <b>4,024</b> |

The same run measured every instruction a payment passes through:

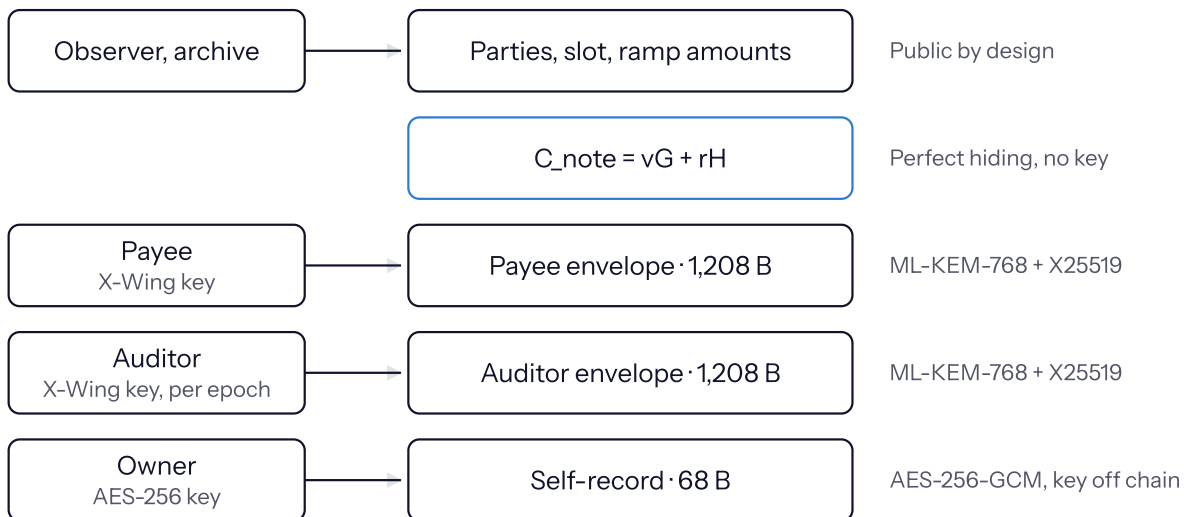
| Instruction                    | Bytes | CU            |
|--------------------------------|-------|---------------|
| <code>register_key</code>      | 1,485 | 7,043         |
| <code>open_account</code>      | 432   | 18,708–21,708 |
| <code>deposit</code>           | 473   | 27,235        |
| <code>transfer</code>          | 4,024 | 226,570       |
| <code>attest</code> , one note | 257   | 6,148         |
| <code>accept</code> , one note | 399   | 9,532         |
| <code>return_note</code>       | 257   | 6,122         |
| <code>withdraw</code>          | 1,479 | 140,575       |

`register_key` and `open_account` run once per participant. Attestation cost is shared across a batch, and a batch of four notes measured 356 bytes and 15,267 CU. The withdrawal figure includes the 111,000 CU U64 proof verification. The client built the transfer's range proof in 283 ms with a WASM build of `solana-zk-sdk` 7.0.1, and its witnesses never left the client. A note is a 235-byte account whose rent the sender pays and gets back when the note closes.

## What the parties see

|                                | Chain observer                            | Archive after Q-Day                                       | Payee                                                     | Auditor                                                 | Sender                        |
|--------------------------------|-------------------------------------------|-----------------------------------------------------------|-----------------------------------------------------------|---------------------------------------------------------|-------------------------------|
| Transfer amount $v$            | No: $C_{\text{note}}$ is perfectly hiding | No: hiding is unconditional; envelopes rest on ML-KEM-768 | Yes, from its envelope, checked against $C_{\text{note}}$ | Yes, from its envelope, checked before attest           | Yes, it chose $v$             |
| Memo                           | No                                        | No                                                        | Yes                                                       | Yes                                                     | Yes                           |
| Confidential balances          | No                                        | No                                                        | Its own, from its self-record                             | Every note amount; with the public ramps, every balance | Its own, from its self-record |
| Deposit and withdrawal amounts | Yes                                       | Yes                                                       | Yes                                                       | Yes                                                     | Yes                           |
| Who paid whom, and when        | Yes                                       | Yes                                                       | Yes                                                       | Yes                                                     | Yes                           |

A commitment hides  $v$  information-theoretically. For every other value  $v'$  there is a blinding  $r'$  with  $v'G + r'H = C_{\text{note}}$ , so no amount of computation can pick out the real one. X-Wing keeps its shared secret safe while either X25519 or ML-KEM-768 holds. After Q-Day, ML-KEM-768 is the part that holds. Self-records are AES-256-GCM under client keys that are never put on chain. Parties, timing and ramp amounts are public by design, as they are in Token-2022.



What each party can open for one payment. No key opens the commitment. Each envelope opens with exactly one X-Wing key, and the self-record opens with its owner's AES key. Whoever recovers  $\$(v, r)\$$  checks it against  $\$C_{\text{note}}\$$ .

## Roles

### Payer

A payer is an employer, buyer or treasury. It holds three keys: an Ed25519 wallet, an X-Wing keypair generated from fresh entropy (never from a wallet signature), and an AES-256 self-record key. Onboarding is `register_key` and then `open_account`. `deposit` moves a public amount of USDC into the vault, within the pool's per-deposit `deposit_cap` and total `pool_cap`. To pay, the client reads the payee's ConfAccount and KeyRecord and the auditor's KeyRecord from chain. It commits the amount, proves it in a Worker and seals both envelopes, then uploads the opaque pair to the relay and submits one signed v1 transaction. Transfers to frozen accounts fail with `AccountFrozen`. An account sends its transfers one at a time, so a payer that needs parallel throughput uses accounts under several of its wallets as lanes. Returned notes come back to the payer's balance.

### Payee

A payee is an employee or supplier. It onboards the same way and then works from its inbox. It opens and checks each note locally, waits for attestation where the pool requires it, and accepts several notes in one `accept`. A note that does not open goes back with `return_note`. `withdraw` pays a public amount to any USDC token account other than the vault, against a U64 proof on the remaining balance.

### Auditor

The auditor is one accountable party per pool, set by `init_pool`. Its service runs in its own trust domain with its own process, database and origin, and the hosted API never receives its ledger, openings or keys.

The per-epoch X-Wing keys and the Ed25519 attestation key sit behind the `AuditorHsm` custody interface. Each signed batch is persisted before submission, so a restart reconciles instead of duplicating. Operators sign in with their wallets into scoped roles. Managers create CSV and JSON ledger exports and selective disclosures for one `ConfAccount`, which are encrypted at rest and kept for an operator-set retention period. The pool's freeze authority is a separate signer that can `freeze` and `thaw` accounts.

## Payroll platform

A payroll platform runs `@hindsight/sdk/payroll` in the employer's browser or a process the employer controls, so CSVs, salaries, openings and keys never reach a hosted service. `parsePayrollCsv` reads `payee,amount` with optional `row_id` and `memo`, an explicit unit (`usdc` or `base_units`) and memos of up to 32 UTF-8 bytes. `planPayroll` spreads rows across lanes round-robin. `preparePayroll` first checks every Pool, `ConfAccount` and `KeyRecord` the batch touches against raw chain bytes. Only then does it generate the notes, envelopes, intermediate commitments and self-records. `executePayroll` runs lanes concurrently and each lane in order. It journals every signed transaction, encrypted, before submitting it, so a lost RPC response is retried with the same bytes. The employer's wallets sign every lane. The hosted progress service stores only an encrypted-plan digest and note references.

## Fees

Transfer amounts are hidden, so a fee proportional to a transfer would reveal the amount. Fees therefore sit where amounts are already public, or they are flat. `init_pool` writes each pool's policy once, and that policy carries two fee fields:

| Policy field                   | Type              | Applies to                                                                |
|--------------------------------|-------------------|---------------------------------------------------------------------------|
| <code>ramp_fee_bps</code>      | u16, basis points | <code>deposit</code> and <code>withdraw</code> , whose amounts are public |
| <code>flat_transfer_fee</code> | u64, per note     | each transfer note, the same for every note                               |

Fees are paid in USDC. A payer reads a pool's fees from its Pool account before depositing. `init_pool` requires every policy field, failing with `ConfigurationRequired` if one is missing, and no instruction changes a policy afterwards. The policy also holds `attest_required`, `return_timeout_slots`, `deposit_cap`, `pool_cap`, `auditor_rotation_interval_seconds`, `admin_multisig_threshold` and `admin_timelock_seconds`. The ramp fee is 5 basis points,  $f = \lfloor a \cdot 5/10^4 \rfloor$ , rounded down so the participant keeps the rounding: a deposit of  $a$  credits  $a - f$ , and a withdrawal of  $a$  pays out  $a - f$ . The flat transfer fee is zero, so a note carries exactly its value. The ramp fee is the only fee.

Rent deposits are refundable and are not fees. A Note's deposit returns to its sender on `accept`, and an owner whose balance is zero takes back its `ConfAccount` and `KeyRecord` deposits with `close_account` and `close_key_record`.

# \$HIND

The protocol's share is the whole ramp fee. `deposit` and `withdraw` pay it in the same instruction into one account: the associated token account of the program address `["protocol"]`, which no key signs for. Those fees exist to buy \$HIND, the project's token, on the market and burn it, and the program does both in one transaction.

`claim_buyback` moves an exact amount  $a$  from the fee account to the keeper's USDC account. One or more swap instructions, Jupiter's on mainnet, spend all of it on \$HIND delivered to the protocol's own token account. `complete_buyback`, the final instruction, requires the keeper's USDC balance back where it started and at least the claim's minimum output in the protocol's account, then burns everything that arrived and adds it to `total_burned`. The minimum is at least  $\lceil a \cdot n/d \rceil$  for a ratio  $n/d$  fixed once in the BuybackConfig account (seeds `["buyback"]`), together with the \$HIND mint and the one keeper allowed to crank. If any check fails, the whole transaction fails, claim included. No instruction withdraws or sweeps the fees.

---

|        |                      |
|--------|----------------------|
| Supply | 1,000,000,000, fixed |
|--------|----------------------|

---

|        |                                                |
|--------|------------------------------------------------|
| Launch | all of it in one public launch, on equal terms |
|--------|------------------------------------------------|

---

|           |                                                                                     |
|-----------|-------------------------------------------------------------------------------------|
| Held back | nothing: no presale, team or investor allocation, airdrop, reserved pool or vesting |
|-----------|-------------------------------------------------------------------------------------|

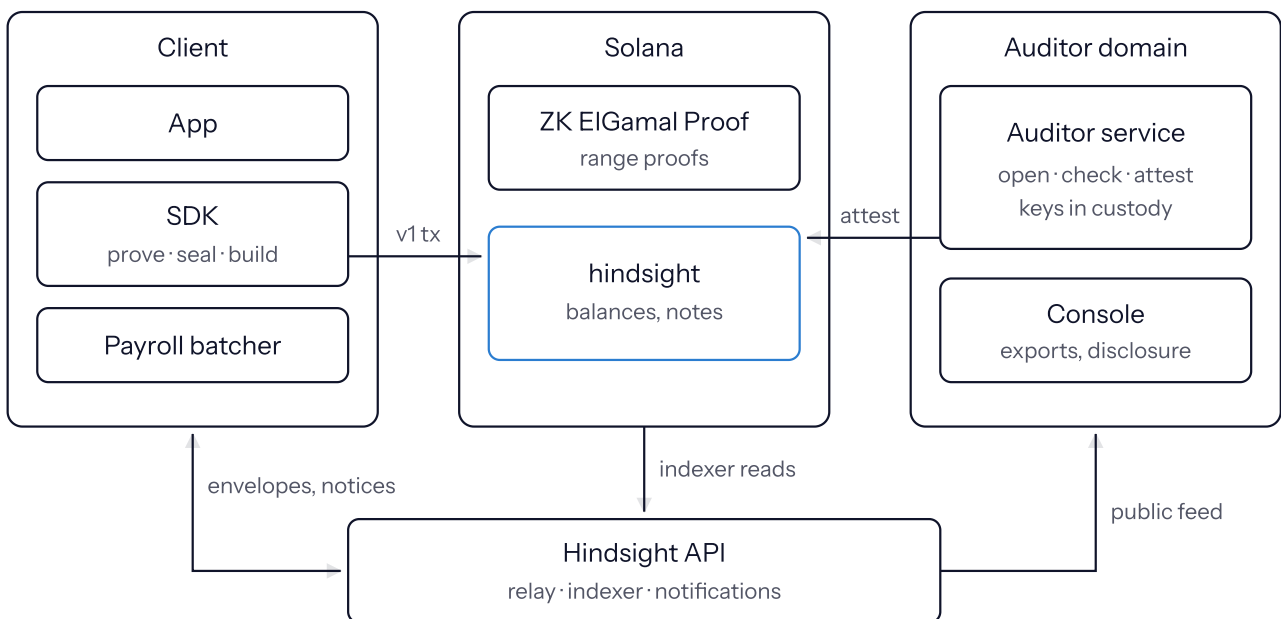
---

No payment needs \$HIND. Payers and payees hold only USDC; the token is where the pool's use goes.

## Components

- **Program.** `hindsight` is built on Anchor 0.31.1 and has seventeen instructions: `init_pool`, `register_key`, `open_account`, `deposit`, `transfer`, `attest`, `accept`, `return_note`, `withdraw`, `close_account`, `close_key_record`, `freeze`, `thaw`, `initialize_rent_recovery`, `configure_buyback`, `claim_buyback` and `complete_buyback`. Its accounts are Pool (238 bytes), KeyRecord (1,265 bytes), ConfAccount (190 bytes), Note (235 bytes), BuybackConfig (209 bytes) and RentRecoveryRegistry (21 bytes plus 32 per protected auditor key), plus a 165-byte classic SPL Token vault. Creating a pool or the BuybackConfig requires the program's upgrade authority, so no other signer can claim a mint's canonical pool or set the buyback's terms.
- **SDK.** `@hindsight/sdk` and `@hindsight/contract` 0.15.1 provide:
  - commitments on Solana's generators
  - X-Wing envelopes through `@noble/post-quantum` 0.7.1, checked against the draft's published vectors
  - U128 and U64 range proofs from a hash-pinned WASM build of `solana-zk-sdk` 7.0.1
  - self-record and opening arithmetic
  - v1 transaction builders

- wallet signing that verifies the returned wire bytes
- a typed client for the API's 51 calls
- **Relay.** Stores envelope pairs by hash. It accepts only exact 1,208-byte envelopes whose ordered SHA-256 matches, and only from wallet sessions. Clients check every envelope against `envelope_hash`, so the relay is not trusted.
- **Indexer.** Projects finalized Pool, ConfAccount, KeyRecord and Note state and streams notifications for each wallet ( `GET /v1/events` ). It signs nothing, cannot decrypt, and stores no confidential amount, opening or memo.
- **Auditor service and console.** The attestation worker described above, plus 14 console calls for queues, history, metrics, exports and disclosures.
- **Payroll batcher.** The payer-side `@hindsight/sdk/payroll` module and the hosted progress service.
- **Buyback crank.** `npm run buyback`, a scheduled job that holds the keeper key. It quotes an exact-input route on Jupiter ( `lite-api.jup.ag/swap/v1` ) whose minimum meets the on-chain floor and submits the claim, the swap and the burn as one transaction. On localnet and devnet a co-signing stand-in market takes Jupiter's place.
- **The app.** `/app` has Account, Inbox, Send, Payroll and Withdraw pages that run every flow in the browser through the SDK.



The components and what passes between them. Clients submit signed v1 transactions to Solana directly and exchange only opaque envelopes and public references with the API. The auditor reads the API's public feed, re-checks each note against chain state and attests on chain. Keys, openings and payroll files stay in the client and in the auditor domain.

## What exists today

The program, services and app are built and running end to end. The program deploys to devnet and mainnet-beta. A full run on an Agave 4.3.0 validator had all 16 of its v1 transactions accepted. It covered honest envelopes opened by both recipients, attestation, acceptance, a full withdrawal, and a garbage-envelope note that the payee returned and the sender accepted back, and it checked solvency after every transition. A separate run of the auditor service attested six honest notes in batches of four and two and left a garbage note unattested. A 50-payee payroll also ran in two lanes of 25. Every payment in it was one 4,024-byte transaction and all 50 notes were accepted, and a deliberately lost RPC response caused no duplicate payment.