

Hindsight whitepaper

Confidential USDC payments on Solana with post-quantum envelopes: the complete construction, from the commitments and the program's accounts to the services that run it.

October 2026 247 min read

Contents

1 Introduction

- 1.1 Payments whose amounts outlive the payment
- 1.2 Harvest now, decrypt later
- 1.3 Q-Day and the deadlines
- 1.4 What Hindsight provides
- 1.5 The shape of the design

2 Background

- 2.1 Solana as Hindsight uses it
- 2.2 Token-2022 confidential transfers
- 2.3 What Hindsight keeps and what it changes

3 System overview

- 3.1 Actors and their keys
- 3.2 Components
- 3.3 On chain and off chain
- 3.4 The life of one note

4 Cryptographic construction

- 4.1 Notation and groups
- 4.2 Pedersen commitments
- 4.3 Balances
- 4.4 Range proofs
- 4.5 Envelopes
- 4.6 Self-records
- 4.7 Confidentiality after Q-Day

5 The on-chain program

- 5.1 Program model
- 5.2 Encoding conventions
- 5.3 Accounts
- 5.4 Instructions
- 5.5 Range-proof binding
- 5.6 Buyback binding
- 5.7 Zero-balance proof
- 5.8 Initial balance commitment
- 5.9 Note validation in batches
- 5.10 Solvency accounting
- 5.11 Error codes

6 Transactions

- 6.1 Why v1 transactions
- 6.2 The byte budget of one transfer
- 6.3 Compute units
- 6.4 Proof adjacency
- 6.5 Compute budget and loaded account data
- 6.6 Address lookup tables
- 6.7 Building a transaction
- 6.8 Signing
- 6.9 Submitting and confirming
- 6.10 Blockhash expiry and resending

7 Flows

- 7.1 Actors and what they sign
- 7.2 Pool setup
- 7.3 Onboarding
- 7.4 Deposit
- 7.5 Transfer
- 7.6 Attestation
- 7.7 Accept
- 7.8 Return and recovery
- 7.9 Withdraw
- 7.10 Close

- 7.11 Payroll batch
- 7.12 Chain effects at a glance

8 The auditor

- 8.1 The auditor's place in a pool
- 8.2 The attestation gate
- 8.3 The auditor service
- 8.4 Job states and transitions
- 8.5 Key epochs
- 8.6 The console
- 8.7 Operator sessions

9 Settlement and fees

- 9.1 What settles where
- 9.2 Commitment arithmetic
- 9.3 The solvency invariant
- 9.4 Deposit and pool caps
- 9.5 Fees
- 9.6 Rent
- 9.7 USDC decimals and rounding

10 Security design

- 10.1 A chain archive read with a quantum computer
- 10.2 A malicious sender
- 10.3 Why envelopes need no proof
- 10.4 A curious relay or indexer
- 10.5 Network observers
- 10.6 Domain separation
- 10.7 Wallet sessions
- 10.8 Key custody
- 10.9 The admin authority, freeze and policy controls

11 Infrastructure

- 11.1 The indexer
- 11.2 The API
- 11.3 The auditor service
- 11.4 The buyback crank
- 11.5 Deployment

12 Payroll

- 12.1 The payee CSV
- 12.2 Lanes
- 12.3 Ordered chains, computed ahead
- 12.4 Encrypted plans and journals
- 12.5 Execution, retries and reconciliation
- 12.6 Progress and settlement

13 Reference

- 13.1 Parameters
- 13.2 Glossary
- 13.3 References

1 Introduction

Hindsight is a system for confidential USDC payments on Solana whose amounts stay sealed after Q-Day, the point at which a cryptographically relevant quantum computer can solve the discrete-logarithm problems that today's elliptic-curve cryptography rests on. Each amount is recorded on chain as a Pedersen commitment. The commitment's opening, meaning the amount together with the random scalar that blinds it, reaches two parties only: the payee and the pool's auditor. Each receives it inside an X-Wing envelope, a hybrid of X25519 and ML-KEM-768, followed by AES-256-GCM. This chapter describes the payments Hindsight is built for, the adversary it is built against, the deadlines that put a date on that adversary, what the system provides, and the overall shape of the design.

1.1 Payments whose amounts outlive the payment

Hindsight is built for employers, payroll platforms, treasury teams, suppliers and stablecoin issuers that settle in USDC on Solana, a chain that held about USD 15.7 billion in stablecoins when the DefiLlama total was fetched on 2026-10-15. Their payments fall into a small number of recurring flows.

- **Payroll.** An employer funds a pool account and pays each employee or contractor every cycle. A salary is personal data with no expiry date, and the employer's payment pattern ties each amount to a named person.
- **Supplier payments.** A buyer pays invoices under multi-year contracts. The amounts disclose unit prices and volumes, which stay commercially sensitive for the life of the contract and after it.
- **Treasury movements.** A company moves funds between its own entities, counterparties and hedging venues. Treasury and hedging positions, confidential legal settlements and advisory fees paid before an M&A announcement belong to the same class.

Most retail payment amounts lose their value within weeks. These keep it for ten years or more. On a public ledger that requirement is demanding, because the ledger is also a permanent archive: validators and archival RPC providers retain every transaction ever posted, ciphertexts included, and anyone can copy them.

Solana already offers amount confidentiality through Token-2022 confidential transfers. Observers cannot see amounts, sender, recipient and timing stay visible, and an optional auditor key per mint can read every amount. Wallets, explorers and institutions have adopted this model of hidden amounts and visible parties, and Hindsight keeps it. What Hindsight changes is the cryptography beneath the hidden amount. Token-2022 protects amounts with twisted ElGamal encryption over the Ristretto group, whose security is the hardness of the discrete logarithm in that group.

1.2 Harvest now, decrypt later

A harvest-now-decrypt-later adversary records ciphertexts today and decrypts them once it can break the underlying public-key problem. Against a public blockchain the recording step costs nothing beyond disk space. The adversary intercepts no traffic, compromises no party and does not have to exist at the time of the payment; it downloads the ledger whenever it is ready.

For Token-2022 confidential transfers the decryption step is short. Every account's ElGamal public key is stored on chain. Shor's algorithm recovers the secret key from that public key, and the secret key opens every ciphertext ever addressed to the account through a discrete-logarithm search small enough for an ordinary computer. The mint's auditor key concentrates the exposure: one discrete logarithm on the auditor's public key opens every transfer amount the mint has ever carried. The Background chapter works through the construction step by step.

Confidentiality and soundness age differently under this adversary. A forged range proof or signature matters only if it is accepted when the transaction is verified, and a quantum computer built later cannot reach back and forge transactions the chain has already settled. A disclosed salary, in contrast, stays disclosed. NIST IR 8547 draws the same line: an authentication system "remains secure as long as" its algorithms are secure when the authentication is performed, whereas encrypted data is exposed to harvest-now-decrypt-later. Hindsight is built on this asymmetry. Confidentiality is post-quantum from the first transfer. Soundness, which rests on range proofs, commitment binding and Ed25519 signatures, is checked at transaction time with classical cryptography and follows Solana's own post-quantum migration.

1.3 Q-Day and the deadlines

No one knows when Q-Day will come, and the published resource estimates keep falling. Gidney (2025) estimates that RSA-2048 can be factored in under a week with fewer than a million noisy qubits. Litinski (2023) describes an architecture that recovers one 256-bit elliptic-curve key every ten minutes. The Global Risk Institute's 2024 expert survey concludes that the threat may be nearer than earlier estimates suggested.

The date itself matters less than Mosca's inequality. Let x be the number of years the data must stay confidential, y the number of years needed to migrate the system that protects it, and z the number of years until a cryptographically relevant quantum computer exists. The data is exposed when

$$x + y > z.$$

On a public ledger the migration term gives no relief for data that has already been posted. Once a ciphertext is in the archive, re-encrypting the live state leaves the archived copy unchanged, so for that ciphertext the condition reduces to $x > z$. A salary that must stay confidential for $x \geq 10$ years is exposed by any z shorter than that, whatever migration follows.

Regulators have turned the inequality into dates that buyers of payment systems answer to:

Year	Body	Requirement
2030	EU coordinated roadmap	Data that must stay confidential for 10 or more years is protected from quantum attack
2030	NIST IR 8547	112-bit ECDH, ECDSA and RSA deprecated
2031	UK NCSC	Highest-priority migrations complete
2035	NIST IR 8547	Quantum-vulnerable public-key algorithms disallowed

Elliptic-curve ElGamal is one of the quantum-vulnerable public-key mechanisms these documents retire. Payroll and long-term supplier terms posted to a public ledger are a direct instance of the EU rule. Their confidentiality requirement exceeds ten years, so their protection has to be quantum-resistant by the end of 2030, and for amounts already on chain it has to have been quantum-resistant at the moment they were posted.

1.4 What Hindsight provides

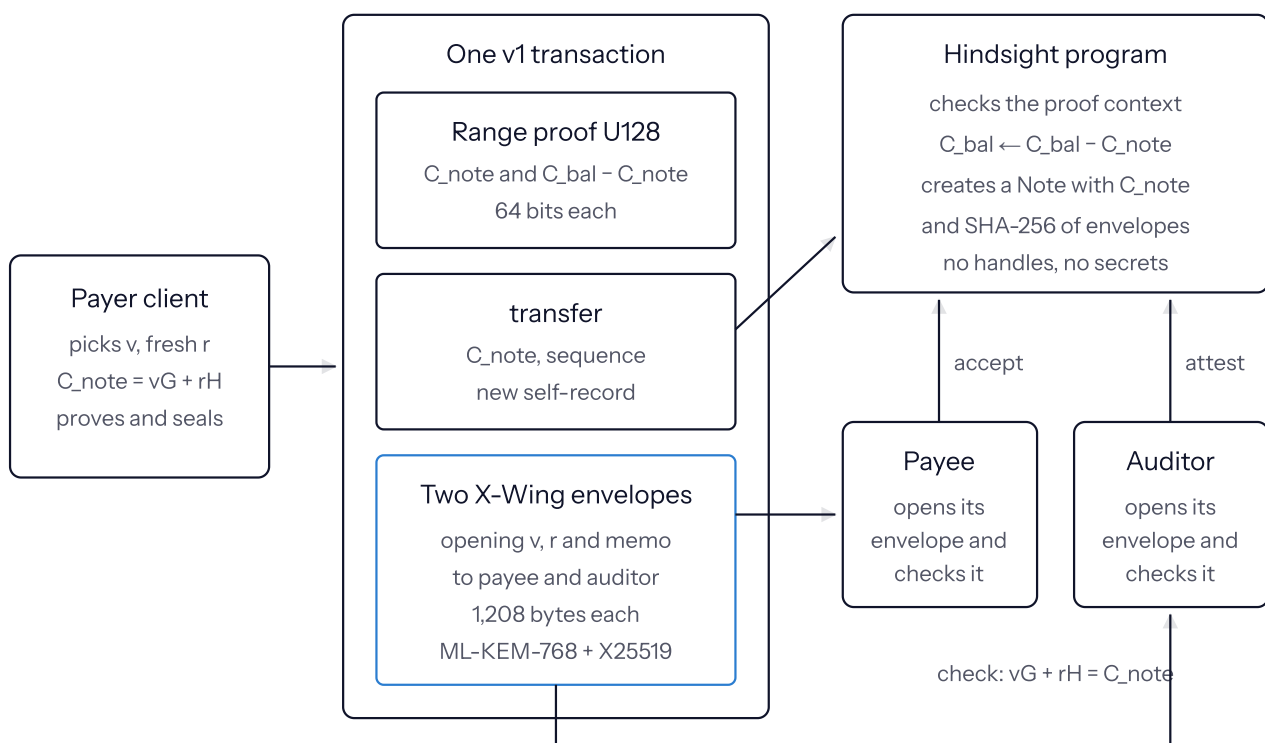
Hindsight consists of one on-chain program, a client SDK, a hosted relay and indexer, an auditor service and a buyback crank. Together they provide the following.

- **Amounts as perfectly hiding commitments.** Every balance and every payment is a Ristretto Pedersen commitment $C = vG + rH$. Perfect hiding means that the commitment carries no information about v , even to an adversary with unbounded computation, quantum or classical.
- **Post-quantum delivery of openings.** The opening (v, r) and a 32-byte memo travel to the payee and to the auditor, each in a 1,208-byte envelope: a 1,120-byte X-Wing ciphertext followed by 88 bytes of AES-256-GCM. No ElGamal decryption handle exists anywhere in the system.
- **Credit only after a check.** A balance grows only through `accept`, which the payee signs after decrypting its envelope and checking that $vG + rH = C_{\text{note}}$. When the pool requires attestation, the auditor first performs the same check on its own envelope and calls `attest`. An envelope that fails to open harms only its sender: the payee sends the note back with `return_note`, and the sender, who chose the opening, accepts it into its own balance.
- **Full 64-bit amounts.** No party ever recovers an amount by solving a discrete logarithm, so amounts are full unsigned 64-bit USDC base units, with no 48-bit cap and no split into halves.
- **One transaction per transfer.** The range proof, the `transfer` instruction and both envelopes fit in a single v1 transaction. Measured on a local Agave 4.3.0 validator, one transfer is 4,024 of 4,096 bytes and consumes 226,570 compute units.
- **An accountable auditor per pool.** Each pool names one auditor, with an Ed25519 attestation key and one X-Wing key per epoch. The auditor service attests in batches and produces ledger exports and selective-disclosure reports.

- **\$HIND, and the one fee that feeds it.** The only fee sits where amounts are already public: 5 basis points on deposits and withdrawals. Transfers carry none, so no fee depends on a hidden amount. The protocol's whole share goes, in the same transaction, to a program account no key signs for, there to buy \$HIND on the market and burn it. The program does both itself: one transaction claims the fees, swaps them for \$HIND and burns every token bought, or none of it happens. The supply of 1,000,000,000 went in full to one public launch, and no payment ever needs the token.

The visibility model is the one Token-2022 established. Senders, recipients and timing are public. USDC enters and leaves through a public vault, so deposit and withdrawal amounts are public as well. Between those two points, every amount exists on chain only as a commitment.

1.5 The shape of the design



One transfer. A single v1 transaction carries the range proof, the `transfer` instruction and two X-Wing envelopes. The program keeps only the commitment and the envelopes' hash. The auditor and the payee each open their own envelope and check the opening against the commitment before `attest` and `accept`.

A transfer begins in the payer's client. The client picks the amount v and a fresh random blinding r , computes $C_{\text{note}} = vG + rH$, and produces one batched range proof showing that both v and the payer's remaining balance lie in $[0, 2^{64})$. It seals the opening twice, once to the payee's X-Wing key and once to the auditor's, and submits one v1 transaction: the native `VerifyBatchedRangeProofU128` instruction, followed by `transfer`.

The Hindsight program reads the preceding instruction and checks that it verified exactly the commitments $[C_{\text{note}}, C_{\text{bal}} - C_{\text{note}}]$ at 64 bits each. It then subtracts C_{note} from the payer's balance commitment and creates a Note account that holds C_{note} and the SHA-256 hash of the two envelopes. The envelopes themselves remain in the transaction's instruction data. The auditor opens its envelope, checks the opening and attests. The payee opens its envelope, checks the opening and accepts, which adds C_{note} to its own balance commitment and closes the note.

The program never sees a hidden amount and holds no secret. Everything the chain keeps about a hidden amount is either perfectly hiding, as the commitments are, or encrypted: the envelopes under X-Wing, and each owner's balance opening under that owner's AES-256 key.

The Background chapter reviews the Solana facilities Hindsight builds on and the Token-2022 construction it replaces. The System overview chapter then introduces the actors and their keys, the components, the split between on-chain and off-chain data, and the life of one note.

2 Background

This chapter reviews the Solana facilities Hindsight builds on, then the Token-2022 confidential-transfer construction it replaces and the reason that construction's amounts become readable at Q-Day. It ends with what Hindsight keeps from Token-2022 and what it changes.

2.1 Solana as Hindsight uses it

2.1.1 Programs and accounts

A Solana program is executable code with no state of its own. State lives in accounts, each owned by one program, and only the owner can write an account's data. Hindsight is one program, `hindsight`, written with Anchor 0.31.1. Each of its accounts begins with an 8-byte discriminator that identifies the type, and the program checks owner and discriminator whenever it loads an account, so a caller cannot present an arbitrary byte array as a Pool or a Note.

Every account holds a refundable rent-exempt deposit of $(\text{size} + 128)$ times the cluster's lamports per byte, read live from the Rent sysvar, which is returned when the account closes. A sender pays the deposit for each Note it creates, and `accept` returns it to that sender; an owner pays for its KeyRecord and ConfAccount, and `close_key_record` and `close_account` return them.

The seven account types, with their actual allocations including the discriminator:

Account	Seeds	Allocation	Contents
Pool	"pool", mint	238 bytes	mint, vault, auditor, auditor KEM epoch, admin and freeze authorities, policy, principal
Vault	"vault", pool	165 bytes	classic SPL Token account for USDC whose authority is the Pool
KeyRecord	"kem", owner, epoch (u64 LE)	1,265 bytes	owner, epoch, 1,216-byte X-Wing encapsulation key
ConfAccount	"acct", pool, owner	190 bytes	balance commitment, 68-byte self-record, sequence, key epoch, frozen flag
Note	"note", pool, sender, sequence (u64 LE)	235 bytes	sender, recipient, original recipient, commitment, envelope hash, creation slot, key epochs, flags
BuybackConfig	"buyback"	209 bytes	\$HIND mint and token program, keeper, minimum-output ratio, total burned, the claim in progress, sequence
RentRecoveryRegistry	"rent-recovery"	21 bytes, plus 32 per protected auditor key	close nonce, protected auditor KeyRecord addresses

2.1.2 Program-derived addresses

A program-derived address (PDA) is computed from a list of seeds and a program ID and lies off the Ed25519 curve, so no private key for it exists. The deriving program signs for it by presenting the seeds and a one-byte bump. Every Hindsight account is a PDA, which fixes each address by its meaning: the ConfAccount of owner o in pool p can only be the PDA of ["acct", p , o], and the Note created by sender s at sequence n can only be the PDA of ["note", p , s , n]. For the notes that `attest` and `accept` receive as a list, the program recomputes each address with `create_program_address` from the stored sender, sequence and canonical bump, and rejects a mismatch with `InvalidNote`.

Custody needs no private key either. The vault's authority is the Pool PDA, and the program signs each withdrawal with the seeds ["pool", mint, bump]. The protocol's fee and \$HIND token accounts belong to the PDA ["protocol"], which the program signs for only inside a buyback. No key controls the vault, so none can be stolen, by a quantum computer or otherwise. Jump Crypto's analysis of Solana's quantum migration paths makes the same observation about PDAs.

Instruction	Discriminator	Aggregate bits N	Proof bytes	Instruction bytes	Verification CU
VerifyBatchedRangeProofU64	6	64	672	937	111,000
VerifyBatchedRangeProofU128	7	128	736	1,001	200,000
VerifyBatchedRangeProofU256	8	256	800	1,065	368,000

The instruction is one discriminator byte followed by the 264-byte context and the proof, with no length prefixes:

Offset	Length	Field
0	1	Instruction discriminator
1	256	Eight compressed Pedersen commitments, in order
257	8	Eight bit lengths, in the same order
265	672, 736 or 800	Proof bytes for U64, U128 or U256

Unused slots hold the identity point with bit length zero. With b the balance behind C_{bal} and a a public withdrawal amount, Hindsight proves two statements:

$$\text{transfer : } v \in [0, 2^{64}), \quad b - v \in [0, 2^{64}) \quad \text{over} \quad [C_{\text{note}}, C_{\text{bal}} - C_{\text{note}}], [64, 64]$$

$$\text{withdraw : } b - a \in [0, 2^{64}) \quad \text{over} \quad [C_{\text{bal}} - aG], [64]$$

The program computes $C_{\text{bal}} - C_{\text{note}}$ and $C_{\text{bal}} - aG$ itself from the stored balance. It requires the proof instruction to come immediately before the Hindsight instruction at the top level, to target the native program, to carry no accounts and to leave every unused slot empty. The native program can also store a verified context in a context-state account for a later transaction; Hindsight accepts only the inline form (`ProofContextStateUnsupported` otherwise), so a proof and the balance change it authorizes always share one atomic transaction. Clients generate the proofs with a pinned Rust/WASM wrapper around solana-zk-sdk 7.0.1, the version family of the Agave 4.3.0 native verifier, and verify each proof locally before signing.

2.1.5 Curve25519 syscalls

Solana exposes Ristretto group operations to programs as syscalls: validation, addition, subtraction and scalar multiplication, with addition at 521 CU and multiplication at 2,208 CU. Hindsight's balance updates

are built from them:

$$\begin{aligned}
\text{deposit}(a) : C_{\text{bal}} &\leftarrow C_{\text{bal}} + aG \\
\text{transfer} : C_{\text{bal}} &\leftarrow C_{\text{bal}} - C_{\text{note}} \\
\text{accept} : C_{\text{bal}} &\leftarrow C_{\text{bal}} + \sum_i C_{\text{note},i} \\
\text{withdraw}(a) : C_{\text{bal}} &\leftarrow C_{\text{bal}} - aG
\end{aligned}$$

The owner tracks the matching openings off chain. The program rejects any caller-supplied commitment that is not a valid Ristretto point (`InvalidCommitment`) and uses the solana-zk-sdk generators: G is the Ristretto basepoint and H is the Ristretto map of SHA3-512 of the compressed G , which lets the native verifier check proofs about commitments the program computes. Because the native context reserves the identity for unused slots, a new ConfAccount starts at $C_0 = r_0 H$, a commitment to zero whose blinding is derived publicly from the pool, the owner and the key epoch, so that an emptied balance is still a point a U64 proof can state.

2.2 Token-2022 confidential transfers

2.2.1 Twisted ElGamal over a Pedersen commitment

The confidential-transfer extension of Token-2022 gives each token account an ElGamal keypair over Ristretto. In the twisted variant the secret key is a scalar s and the public key is

$$P = s^{-1}H.$$

An amount x encrypted to P with randomness r is the pair

$$C = xG + rH, \quad D = rP.$$

C is exactly a Pedersen commitment to x , and D is a decryption handle. The commitment part can be shared across recipients: a transfer carries one commitment to the amount and three handles, $D_{\text{src}} = rP_{\text{src}}$, $D_{\text{dst}} = rP_{\text{dst}}$ and $D_{\text{aud}} = rP_{\text{aud}}$, one for each party that can decrypt. Sigma proofs show that the handles are consistent with the commitment, range proofs show that the amounts are in range, and the ZK ElGamal Proof program verifies both. Incoming amounts accumulate in a pending balance, which the recipient later applies to its available balance.

2.2.2 Decryption as a small discrete logarithm

The holder of s removes the blinding:

$$C - sD = xG + rH - s r s^{-1}H = xG.$$

What remains is xG , and recovering x from xG is a discrete logarithm. It is tractable only because x is small. Token-2022 therefore caps a transfer amount at 48 bits and encrypts it as two halves,

$$x = x_{\text{lo}} + 2^{16}x_{\text{hi}}, \quad x_{\text{lo}} < 2^{16}, \quad x_{\text{hi}} < 2^{32},$$

so that each ciphertext holds a value of at most 32 bits, small enough for a baby-step giant-step search. The 48-bit cap and the lo/hi split follow from decryption by discrete logarithm, not from the token: USDC amounts are 64-bit base units.

2.2.3 The auditor handle

A mint with the extension may name one auditor ElGamal public key P_{aud} in its configuration. Every confidential transfer under that mint then carries $D_{\text{aud}} = rP_{\text{aud}}$ next to its commitment, so the auditor can decrypt every amount. One key serves the whole mint, for its whole history.

2.2.4 What a quantum adversary recovers

Shor's algorithm computes discrete logarithms in the Ristretto group. Given a public key $P = s^{-1}H$ from the chain, it computes $s^{-1} = \log_H P$ and from it s . The classical steps above then apply to every ciphertext ever sent to P : $C - sD = xG$, followed by a search over at most 2^{48} candidate amounts, which baby-step giant-step completes in about 2^{24} group operations.

Component	Classical adversary	Quantum adversary
Pedersen commitment C	Perfectly hiding	Perfectly hiding; binding no longer holds
Sigma proofs, Bulletproofs range proofs	Zero-knowledge	Add no leakage beyond the ciphertexts
Handle $D = rP$, with P on chain	Hiding under DDH	s from P , then $sD = rH$, then xG , then x
Auditor handle	Hiding under DDH	One discrete log on the mint's auditor key reveals every transfer amount of the mint
Sender, recipient, timing	Public	Public

The commitment is not what fails: Pedersen hiding is information-theoretic and survives Shor. The handle fails, because it is only computationally hiding and sits next to a public key on chain. Key derivation does not help, since the attack starts from that public key rather than from the wallet. A related fact shapes Hindsight's own key rules: Shor run against an Ed25519 wallet key recovers the derived scalar but not the seed it came from. The auditor key turns one quantum computation into a mint's entire amount history, every salary paid through the mint included.

2.3 What Hindsight keeps and what it changes

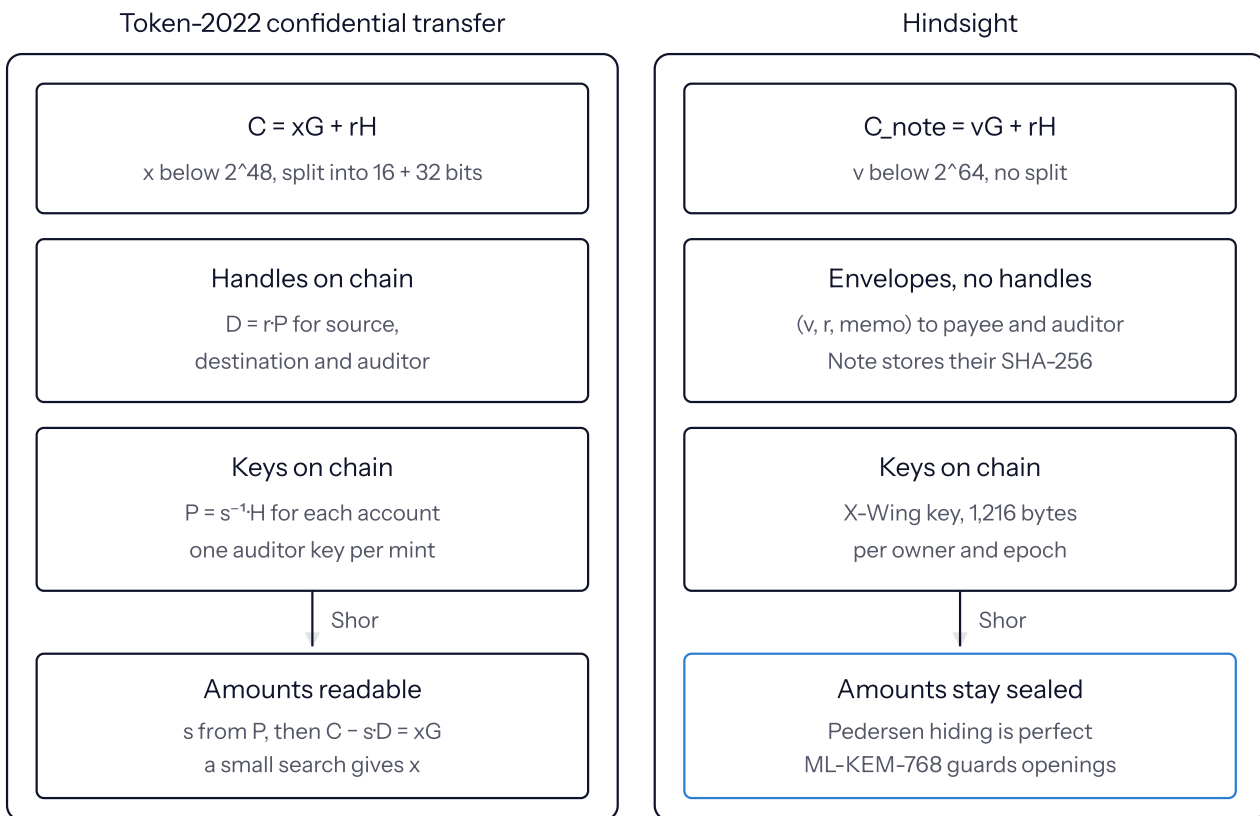
Hindsight keeps the parts of the Token-2022 design that hold against a quantum adversary and replaces the part that does not.

It keeps the Pedersen commitment, with the same generators G and H , as the on-chain form of every amount. It keeps the native ZK ElGamal Proof program and its Bulletproofs range proofs, which need no trusted setup and are verified by the runtime rather than by Hindsight's own code. It keeps the model of hidden amounts and visible parties, the auditor as a named role, and the separation between incoming and spendable value, with notes in the place of Token-2022's pending balance.

It removes the decryption handles. The opening (v, r) and a 32-byte memo go to the payee and to the auditor in X-Wing envelopes, each a 1,120-byte X-Wing ciphertext followed by AES-256-GCM over the 72-byte plaintext. Recovering an opening requires breaking ML-KEM-768 as well as X25519, and Shor's algorithm breaks only X25519. Envelopes travel in instruction data and the Note stores their SHA-256 hash, so they are part of the chain's history but not of its account state. With no decryption by discrete logarithm, amounts are full 64-bit values, and one U128 batched proof covers the note and the remaining balance at 64 bits each.

The envelopes are not proven consistent with the commitment. By the estimates made when the design was chosen, proving an ML-KEM envelope in a SNARK would take at least 10^5 constraints, and replacing ElGamal with an additively homomorphic lattice scheme would take 12.8 KB to 51 KB of ciphertext per transfer over 4 to 13 v1 transactions. Hindsight uses a rule in place of the proof: no one is credited with a commitment they cannot open. The payee accepts a note only after decrypting it and checking $vG + rH = C_{\text{note}}$, and where the pool sets `attest_required`, the auditor attests the same check on its own envelope first. A malformed envelope leaves its sender with a note to take back through `return_note`.

The auditor becomes a per-pool role with keys organized by epoch. Each Pool names its auditor and the epoch of the auditor's current X-Wing key, and each Note records the payee and auditor key epochs its envelopes were sealed to. Under `attest_required` the auditor also gates settlement: a note cannot be accepted before the auditor has opened and checked it. Custody moves to a classic SPL Token vault whose authority is the Pool PDA. USDC enters through `deposit` and leaves through `withdraw`, both with public amounts, and in between it moves only as commitments.



The two designs side by side. Token-2022 attaches a decryption handle per party to each commitment and keeps the ElGamal public keys on chain, so recovering a key with Shor's algorithm opens every amount sent to it. Hindsight keeps the commitment, removes the handles and delivers the opening in X-Wing envelopes, whose ML-KEM-768 component Shor's algorithm does not break.

	Token-2022 confidential transfers	Hindsight
Amount on chain	Pedersen commitment inside a twisted ElGamal ciphertext	Pedersen commitment $vG + rH$ with the same G and H
Decryption material on chain	Handles rP for source, destination and auditor	None; the Note stores the SHA-256 of the two envelopes
How a recipient learns the amount	Computes $C - sD = xG$ and solves a discrete logarithm	Opens its X-Wing envelope and checks $vG + rH = C$
Amount width	48 bits, split 16 + 32	64 bits, no split
Auditor	One ElGamal key per mint	One auditor per pool, an X-Wing key per epoch, an attestation gate
Incoming value	Pending balance	Notes, settled by <code>accept</code> or <code>return_note</code>
Proofs per transfer	Sigma proofs and range proofs	One batched range proof (U128)
Token custody	Token-2022 mint extension	Classic SPL Token vault, authority is the Pool PDA
After Q-Day	Every amount readable from public keys on chain	Commitments stay perfectly hiding; openings require breaking ML-KEM-768

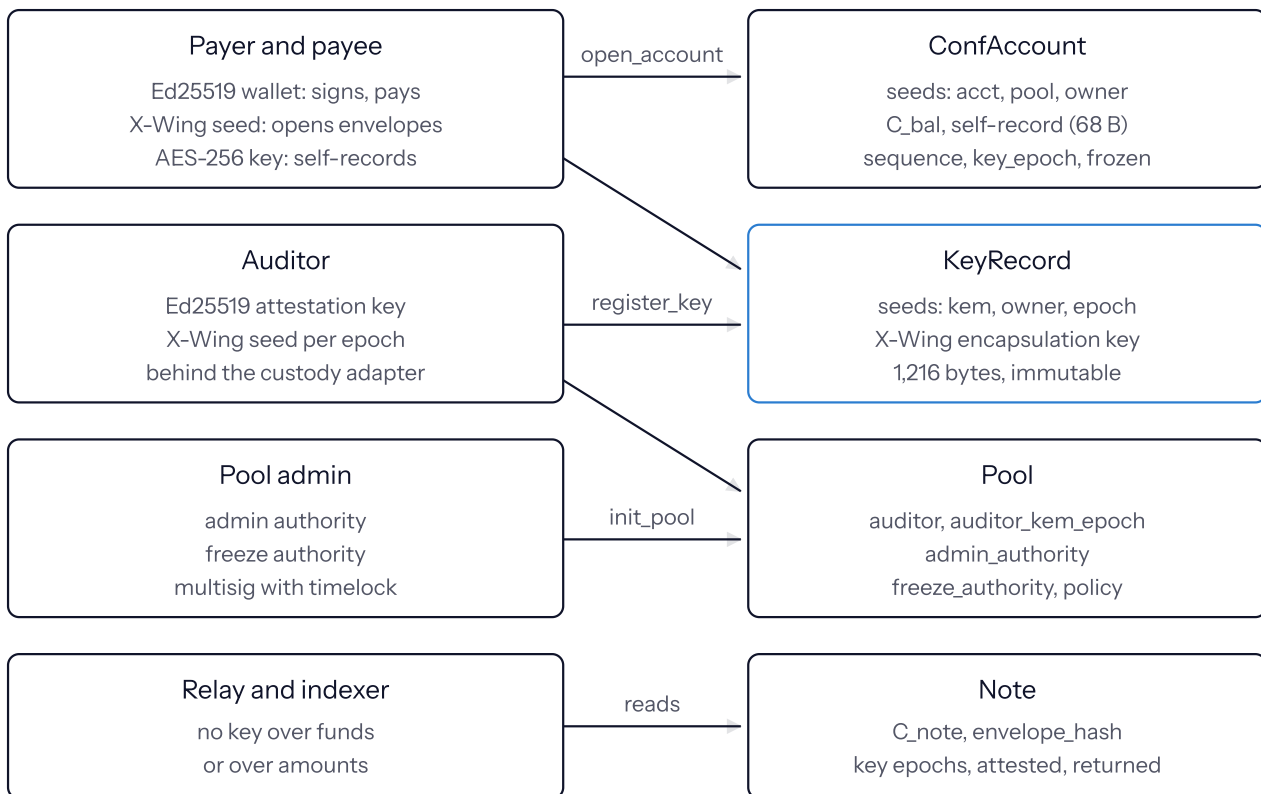
3 System overview

This chapter introduces the parties that use Hindsight and the keys each one holds, the components of the system and the boundaries between them, which data lives on chain and which off chain, and the life of one note from the payer's client to the payee's balance.

3.1 Actors and their keys

Payer and payee are roles in a payment rather than kinds of account. Every participant holds the same keys and opens the same kind of ConfAccount, and an employer that pays salaries becomes the recipient of any note returned to it.

Actor	Role	Keys	Recorded on chain as
Payer (employer, buyer, treasury)	Deposits USDC, sends transfers, recovers returned notes	Ed25519 wallet key, X-Wing keypair, AES-256 self-record key	ConfAccount owner, KeyRecord owner, Note sender
Payee (employee, supplier)	Registers a key, accepts or returns notes, withdraws	The same three keys	ConfAccount owner, KeyRecord owner, Note recipient
Auditor (one per pool)	Opens its envelope, checks the opening, attests; sees every amount in its pool	Ed25519 attestation key, one X-Wing keypair per epoch	<code>Pool.auditor</code> , <code>Pool.auditor_kem_epoch</code> , its KeyRecords
Pool admin	Creates the pool, sets the auditor and policy, freezes and thaws accounts	Admin authority and freeze authority, held by a multisig with timelock	<code>Pool.admin_authority</code> , <code>Pool.freeze_authority</code>
Buyback keeper	Claims protocol fees, swaps them for \$HIND and has the program burn what they bought, in one transaction	Ed25519 keeper key, held by the buyback crank	<code>BuybackConfig.keeper</code>
Relay and indexer	Stores envelopes by hash, indexes the chain, sends notifications	No key that moves funds or opens an amount	Nothing



Actors and keys. Payers and payees register an X-Wing key in a KeyRecord and own a ConfAccount; the auditor registers its X-Wing key the same way and is named, with its key epoch, in the Pool that the admin creates. Signing keys appear on chain only as addresses, encapsulation keys appear in KeyRecords, and seeds and self-record keys never leave their holders. The relay and indexer only read.

Wallet key. The owner's Ed25519 wallet signs the transactions that change its account and pays their fees and deposits; in the measured transfer and withdrawal it is the only signer. It also signs the one-use challenge that opens an API session. It never contributes to confidentiality. No encryption key is derived from a wallet signature, because such keys can be phished, differ between MPC and hardware wallets, and would tie confidentiality back to a classical key.

X-Wing keypair. The private key is a 32-byte seed generated from fresh entropy in the client. `register_key` writes the 1,216-byte encapsulation key into the immutable KeyRecord at `["kem", owner, epoch]` and rejects any other length (`InvalidKeyLength`), so a new key is a new epoch. The ConfAccount's `key_epoch` names the epoch its owner receives on, `transfer` accepts only that payee KeyRecord (`InvalidKeyEpoch`), and each Note records the epoch it was sealed to, so an older seed kept in the client still opens older notes.

Self-record key. A non-extractable AES-256 key generated in the client. Every balance-changing instruction stores a new 68-byte self-record in the ConfAccount: AES-256-GCM over the balance and its blinding, bound to the pool, the account, the sequence and the balance commitment. With this key alone the owner recovers its balance opening from chain state.

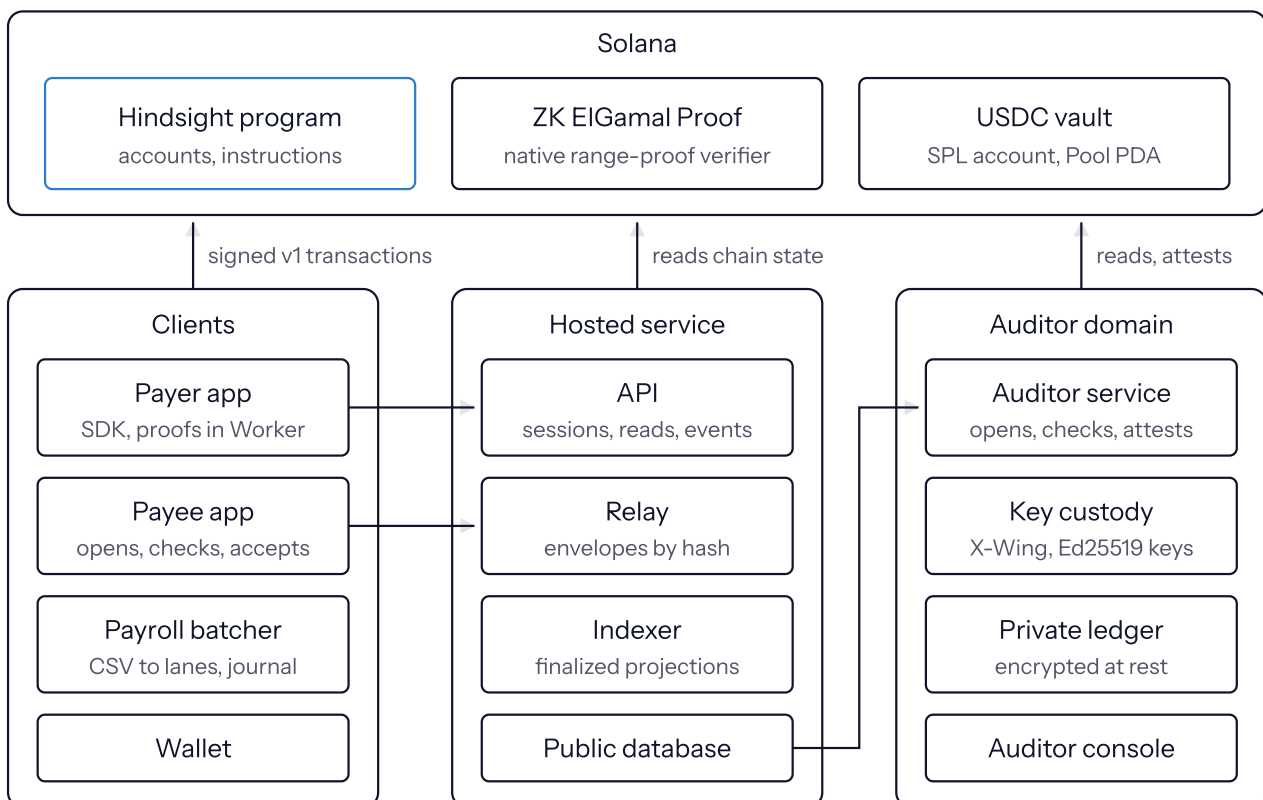
Auditor keys. `Pool.auditor` is the Ed25519 key that alone can sign `attest` (`InvalidAuthority`). `Pool.auditor_kem_epoch` selects the auditor's KeyRecord at `["kem", Pool.auditor,`

`Pool.auditor_kem_epoch]` , the only auditor key `transfer` accepts. Both keys sit behind the custody interface `AuditorHsm` , which separates opening and checking envelopes from signing transactions.

Pool authorities. `init_pool` must be signed by the program's upgrade authority (`InvalidBootstrapAuthority`), so only the deployment's governance creates the canonical pool for a mint. The same authority signs `configure_buyback` , once, which fixes the buyback's \$HIND mint, keeper and minimum-output ratio, and `initialize_rent_recovery` , once, which creates the registry that keeps every pool's auditor `KeyRecord` from being closed. The admin and freeze authorities are signers controlled by an external multisig with timelock, whose threshold and delay the policy records. A frozen account cannot change its balance or receive a transfer, while `return_note` still works on its notes.

The relay and indexer hold credentials for sessions and request integrity but no protocol key, and clients check every envelope against the hash in its Note. Three rules apply to all key material: post-quantum keys come from fresh entropy, never from a wallet signature; note blindings come from a CSPRNG and are never reused; envelopes have a fixed size and memos are padded to 32 bytes.

3.2 Components



Component architecture. Clients build, prove, sign and submit their own transactions; the hosted service relays opaque envelopes, indexes chain state and streams notifications; the auditor runs in its own domain, with its own database and key custody, reads the public tables through a read-only feed and re-checks every note against the chain before it attests.

3.2.1 The program

`hindsight` is one Anchor program with the seven account types described in the Background chapter and 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`. It holds no secret and decrypts nothing.

Only the owner changes a balance: `deposit`, `transfer`, `accept` and `withdraw` require the `ConfAccount` owner's signature and the account's current `sequence` (`StaleSequence` otherwise), and each increments it. No other party's transaction can therefore make a proof stale. `transfer` and `withdraw` bind to the preceding native range proof over exactly $[C_{\text{note}}, C_{\text{bal}} - C_{\text{note}}]$ and $[C_{\text{bal}} - aG]$. `attest` and `accept` check every note of a batch for pool, address and duplicates (`DuplicateNote`), and `accept` refunds each note's deposit only to its original sender (`InvalidRentDestination`).

The pool's public accounting bounds the hidden amounts. `Pool.principal` is total deposits less total withdrawals. With sound range proofs, the hidden balances b_j and the hidden amounts v_i of unsettled notes satisfy

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

`transfer`, `accept` and `return_note` leave principal unchanged. USDC sent to the vault outside `deposit` is public surplus and creates no confidential credit.

Each pool carries its policy, set at `init_pool` :

Field	Effect
<code>attest_required</code>	<code>accept</code> requires each note to be attested (<code>AttestationRequired</code>), except a note returned to its own sender
<code>return_timeout_slots</code>	Slots after creation from which the sender may return its own unaccepted note
<code>deposit_cap</code>	Largest single deposit (<code>DepositCapExceeded</code>)
<code>pool_cap</code>	Largest principal (<code>PoolCapExceeded</code>)
<code>ramp_fee_bps</code>	Fee in basis points on deposits and withdrawals
<code>flat_transfer_fee</code>	Flat fee per transfer note
<code>auditor_rotation_interval_seconds</code>	Interval between auditor key epochs
<code>admin_multisig_threshold</code> , <code>admin_timelock_seconds</code>	Threshold and delay of the admin multisig

Every field must be given explicitly (`ConfigurationRequired`); the program assumes no default. The deposit cap may not exceed the pool cap (`InvalidPolicy`). The ramp fee is 5 basis points (the program accepts up to 100); the flat transfer fee is zero, and the program refuses any other transfer fee with `FeeCustodyNotConfigured` .

The whole ramp fee goes to the protocol fee account, the token account of the program address `["protocol"]` for the pool's mint, and leaves it only through the buyback. `configure_buyback` creates the global `BuybackConfig`, which fixes the \$HIND mint, the one keeper and a minimum-output ratio. The keeper's transaction runs `claim_buyback` , a swap and `complete_buyback` last; the completion requires the whole claim to have been spent and at least the minimum to have reached the protocol token account, then burns everything that arrived.

3.2.2 The SDK

`@hindsight/sdk` is a browser-safe TypeScript package that performs all confidential work on the client, and `@hindsight/contract` supplies the typed API calls and protocol constants. The SDK implements no cryptographic primitive itself.

- **Commitments and envelopes.** Ristretto commitments over the solana-zk-sdk generators, X-Wing through `@noble/post-quantum` 0.7.1, HKDF-SHA256 and AES-256-GCM through WebCrypto. Sealing binds role, pool, sender, original payee wallet, sequence, key epoch and commitment, and `open` returns an opening only after checking it against the commitment.
- **Range proofs.** The pinned WASM wrapper around solana-zk-sdk 7.0.1, run in a browser Worker. Witnesses never leave the client, and every proof is verified locally.

- **Transactions and signing.** Builders produce unsigned v1 transactions from current chain state. The wallet adapter accepts browser-injected, Wallet Standard and mobile deep-link wallets, excludes USB hardware wallets, and verifies that the signature covers the unchanged message.
- **Openings and self-records.** Helpers such as `openingAfterTransfer` and `openingAfterAccept` carry the balance opening through every instruction with exact u64 and scalar-field arithmetic.
- **Payroll.** A CSV with `payee, amount` and optional `row_id, memo` columns is planned across lanes, each the ConfAccount of a distinct payer-controlled wallet. Preparation checks the Pool and every account and KeyRecord of the batch on chain before sealing anything, then prepares every note, intermediate opening, envelope and self-record. Plan and journal are encrypted locally; lanes run concurrently, each in order, and every signed transaction is persisted before submission.

3.2.3 Relay and indexer

The relay stores envelope pairs by hash. A sender uploads its pair under a wallet session before submitting the transfer. The relay checks that both envelopes are exactly 1,208 bytes and that the pair hashes to $\text{SHA256}(e_{\text{payee}} \parallel e_{\text{auditor}})$, and serves an upload as verified only after reconciling it with the Note on chain. Anyone may fetch a pair by its hash, since it is ciphertext, so recipients do not depend on archival RPC history.

The indexer reads the chain over RPC and websocket. Finalized snapshots populate the projections of Pools, ConfAccounts, KeyRecords and Notes; confirmed observations speed up discovery but never credit a balance or settle a note. Each cycle commits atomically in PostgreSQL, missing history is recorded as an explicit gap, and finalized events feed per-wallet notifications.

3.2.4 API

The hosted API exposes the relay, the projections and the notifications through typed calls. A wallet session starts with `authChallenge`, a one-use message bound to wallet, origin, service audience, cluster, program, genesis, nonce and expiry, and `authVerify`, which checks the Ed25519 signature and sets an HttpOnly cookie; mutations carry a CSRF token. Public calls serve pools, the vault balance and KeyRecords; owner-scoped calls serve accounts, inbox, outbox, notes and public ramp history. Notifications arrive through `notifications` or the server-sent event stream `GET /v1/events`. Payroll progress records only plan digests and note references.

The API never generates a proof, opens an envelope or self-record, or signs a transaction. On a cluster whose RPC endpoint carries a provider key, the browser's JSON-RPC calls, `sendTransaction` included, pass through the API's restricted relay (`POST /v1/rpc`), which forwards the wallet's signed bytes unchanged and keeps the key on the server. Its database has no column for a hidden amount, an opening, a memo, a payroll CSV or a private key.

3.2.5 Auditor service

The auditor service runs in its own domain, with its own process, origin, database and credentials. It discovers notes through a read-only feed of the indexer's public tables and treats that feed as untrusted: before opening an envelope, and again before signing, it fetches the Note, the Pool and the KeyRecord

from the chain and checks owner, address, pool, parties, commitment, epochs, current auditor and envelope hash. It then opens its envelope, checks $vG + rH = C_{\text{note}}$ and attests in batches.

Each note is a durable job that moves from QUEUED through VERIFIED, SUBMITTED and CONFIRMED to FINALIZED. An envelope that fails to open or check is held in FAILED_TO_OPEN or OPENING_MISMATCH and never attested. Signed batch bytes are persisted before submission, so a retry resends the same transaction. Ledger and exports are encrypted at rest under an auditor-only key. The console provides the queue, attestation history, metrics, key epochs, CSV and ledger exports and selective-disclosure reports, scoped per pool and operator role. The attestation target is a p95 under 10 seconds.

3.2.6 Buyback crank

The buyback crank is the keeper's side of the buyback: `npm run buyback`, run as its own scheduled service with the keeper key named in the BuybackConfig. Each run reads the protocol fee balance, takes an exact-input USDC-to-\$HIND quote from Jupiter on mainnet-beta whose minimum output meets the on-chain floor, and submits `claim_buyback`, the swap and `complete_buyback` as one transaction. It has no database and needs no USDC of its own; the bought tokens go straight to the protocol token account, where `complete_buyback` burns them.

3.2.7 Applications

The reference web app at `/app` runs the SDK in the browser and keeps keys in client storage, with views for the account and its key backup, sending, the inbox with sent notes, payroll and withdrawing. The auditor console is served from its own origin. The landing page runs a real seal in the browser: fresh X-Wing keys, a commitment and two envelopes, each opened and checked.

3.3 On chain and off chain

Data	Where	Protected by
Balance commitment C_{bal} , 32 bytes	ConfAccount	Perfect hiding
Balance opening, 68-byte self-record	ConfAccount	AES-256-GCM under the owner's key
Note commitment C_{note} , envelope hash, parties, epochs, flags	Note	Perfect hiding for the amount; the rest is public
X-Wing encapsulation keys, 1,216 bytes	KeyRecord	Public by design
Envelopes, $2 \times 1,208$ bytes	Transaction history, relay copy	X-Wing (ML-KEM-768 and X25519) and AES-256-GCM
Range proof and context, $736 + 264$ bytes	Transaction history	Zero-knowledge
Deposit and withdrawal amounts, principal, vault balance	Pool, vault, transaction history	Public by design
Protocol fees, buyback configuration, total burned	Protocol fee account, BuybackConfig	Public by design
Openings, plaintext balances, memos, X-Wing seeds, self-record keys	Payer and payee clients	Never leave the client
Auditor openings, ledger, exports	Auditor service	Custody adapter; AES-256-GCM at rest
Payroll CSV, plan, journal	Payer client	AES-256-GCM under a payer-held key
Keeper key	Buyback crank	Secret variable of the crank service only
Sessions, notifications, projections, envelope copies	Hosted database	Holds public or opaque data only

The adversary described in the Introduction archives everything in the on-chain rows, envelopes included. What it obtains is a set of commitments that carry no information about amounts, envelopes that open only if ML-KEM-768 is broken as well as X25519, and self-records under AES-256.

3.4 The life of one note

A note is the unit of payment. `transfer` creates it, it waits in the payee's inbox, and it ends when it is accepted, by the payee or, after a return, by its sender.

1. **Prepare.** The payer's client reads from the chain the payee's `ConfAccount`, the payee's `KeyRecord` at that account's key epoch, and the auditor's `KeyRecord` at the pool's current epoch. It picks v and a fresh r , computes $C_{\text{note}} = vG + rH$, seals the opening into the two envelopes, generates the U128 proof in a `Worker`, and seals the next self-record for the remaining opening $(b - v, r_{\text{bal}} - r)$.
2. **Upload.** The client uploads the envelope pair to the relay under its hash.
3. **Submit.** The wallet signs one v1 transaction, `VerifyBatchedRangeProofU128` followed by `transfer`, and the client submits it to RPC. The program checks the proof context, the sender's sequence, the payee's key epoch and frozen flag and both envelope lengths, debits C_{note} , and creates the Note at `["note", pool, sender, sequence]`, with the sender paying its deposit.
4. **Attest.** The auditor service picks up the note, checks it against the chain, opens its envelope, checks the opening and includes the note in an `attest` batch.
5. **Open.** The payee receives a notification, fetches the pair by hash, checks it against `Note.envelope_hash`, opens its envelope and checks $vG + rH = C_{\text{note}}$.
6. **Settle.** If the check holds, the payee calls `accept`, which adds C_{note} to its balance, closes the note and refunds the deposit to the sender. If it fails, the payee calls `return_note`, which re-targets the note to its sender, and the sender accepts it with the opening it kept, without attestation. If the payee does nothing, the sender may return the note itself once `return_timeout_slots` have passed since `created_slot`.

State	Note fields	Entered through	Left through
pending	<code>attested = false</code> , <code>returned = false</code> , recipient is the payee	<code>transfer</code>	<code>attest</code> , <code>return_note</code> , or <code>accept</code> when the pool does not set <code>attest_required</code>
attested	<code>attested = true</code>	<code>attest</code> , signed by <code>Pool.auditor</code>	<code>accept</code> or <code>return_note</code>
returned	<code>returned = true</code> , <code>attested = false</code> , recipient is the sender	<code>return_note</code> , by the payee at any time or by the sender after the timeout	<code>accept</code> by the sender
accepted	account closed, deposit refunded to the sender	<code>accept</code>	none

A returned note cannot be attested or returned again (`NoteAlreadyReturned`), and `original_recipient` keeps the payee's identity after re-targeting, which the auditor's disclosure reports use. A payroll run repeats this life once per row: each lane is an ordered chain of notes from one `ConfAccount`, with every intermediate commitment and opening prepared before the first transaction of the run is signed.

4 Cryptographic construction

Hindsight places two kinds of cryptographic object on chain. A Pedersen commitment over Ristretto255 stands in for every amount that must stay confidential: the balance of each confidential account and the value of each note in transit. The opening of a note commitment is carried to exactly two parties, the payee and the pool's auditor, inside X-Wing envelopes. The envelopes travel in the data of the `transfer` instruction, and the Note account stores their hash. Around these two objects sits the machinery that keeps the ledger sound: batched Bulletproofs range proofs, verified by Solana's native ZK ElGamal Proof program, and a 68-byte encrypted self-record that lets an account owner recover its own balance opening from chain state alone.

Each amount-bearing object in the archive is therefore either free of computational assumptions or encrypted under a key that a quantum computer cannot recover from public data. Commitments are perfectly hiding, and the range proofs are treated as perfectly zero-knowledge. Envelopes are protected by ML-KEM-768 inside the X-Wing combiner. Self-records are encrypted under an AES-256 key that never leaves the owner. Soundness, the guarantee that no one spends more than they hold, rests on the discrete-logarithm problem and on the soundness of Bulletproofs, and it is needed only at the moment a transaction is verified. This chapter specifies each object as the SDK (`@hindsight/sdk`) and the on-chain program (`hindsight`) define it: the group and its encodings, commitments, balance arithmetic, range proofs, envelopes and self-records, and then the confidentiality argument that follows from them.

4.1 Notation and groups

4.1.1 The group and its scalar field

All curve arithmetic takes place in Ristretto255, the prime-order group constructed over Curve25519. Its order is

$$\ell = 2^{252} + 27742317777372353535851937790883648493,$$

and its scalar field is $\mathbb{Z}_\ell$. A group element is written as its 32-byte canonical compressed encoding, and decoding rejects any non-canonical byte string. The identity element encodes as 32 zero bytes. That encoding matters later, because the native proof format uses it as the sentinel for an unused commitment slot. A scalar is a 32-byte little-endian integer, and it is canonical only when its value is less

than ℓ . The SDK rejects non-canonical scalars wherever it reads one: in commitments, openings, envelope plaintexts, self-records and proof witnesses.

The program never implements curve arithmetic itself. It calls Solana's curve25519 syscalls through the maintained `solana-curve25519` wrapper: `validate_ristretto`, `add_ristretto`, `subtract_ristretto` and `multiply_ristretto`. An addition costs 521 CU and a multiplication 2,208 CU. The SDK uses the noble Ristretto255 implementation, and it does not implement a curve, KEM or symmetric primitive of its own either.

Symbol	Meaning
G, H	Pedersen generators (value and blinding)
$\ell, \mathbb{Z}_\ell$	Group order and scalar field
v, r	Note amount and note blinding
b, ρ	Balance amount and balance blinding
a	Public deposit or withdrawal amount
C_{note}, C_{bal}	Note commitment, balance commitment
r_0	Public initial balance blinding
k, R, c, s	Zero-balance proof: secret nonce, nonce point, challenge, response
$x y$	Byte concatenation
$\text{LE}_n(x)$	x as n little-endian bytes
$\text{int}_{\text{LE}}(s)$	The integer whose little-endian encoding is s
pk_R, ct, ss	X-Wing public key, ciphertext, shared secret
δ_{init}	ASCII <code>hindsight:initial-balance-blinding:v1</code> (37 bytes)
δ_{env}	ASCII <code>hindsight:envelope:v0:</code> (22 bytes)
δ_{kdf}	ASCII <code>hindsight:envelope:aes256gcm:key-nonce:v0:</code> (42 bytes)
δ_{rec}	ASCII <code>hindsight:self-record:v1:</code> (25 bytes)
δ_{close}	ASCII <code>hindsight:close-account:v1</code> (26 bytes)

4.1.2 The generators G and H

The value generator G is the standard Ristretto basepoint, compressed as

```
e2f2ae0a6abc4e71a884a961c500515f58e30b6aa582dd8db6a65945e08d2d76 .
```

The blinding generator H is derived from G by hashing:

$$H = \text{RistrettoMap}(\text{SHA3-512}(\text{enc}(G))).$$

The 32-byte encoding of G is hashed with SHA3-512, and the 64-byte digest is mapped into the group with the Ristretto one-way map, which maps each 32-byte half and adds the two points. This is exactly the construction in solana-zk-sdk's `encryption/pedersen.rs` (`RistrettoPoint::hash_from_bytes::<Sha3_512>(G_compressed)`). The program pins the result as the constant `PEDERSEN_H`:

```
8c9240b456a9e6dc65c377a1048d745f94a08cdb7f44cbcd7b46f34048871134 .
```

The SDK computes the same point with noble's Ristretto map over the SHA3-512 digest and exports both encodings as `PEDERSEN_GENERATORS`. The browser SDK and the Rust tests check the generator byte for byte. Two properties of this choice carry the rest of the design. First, the native ZK ElGamal Proof program proves ranges over Pedersen commitments with exactly these generators, so a commitment formed under any other H could not be range-proven by it. The SDK documentation therefore rules out substituting RFC 9380 `hash_to_curve`, which produces a different point. Second, H is the output of a hash, so no one knows the discrete logarithm η with $H = \eta G$. Binding depends on that.

4.1.3 Encodings

Every value that enters a hash, an AEAD's associated data or an account has one fixed encoding.

Object	Bytes	Encoding
Amount v, b, a	8	Unsigned 64-bit little-endian, USDC base units (6 decimals)
Scalar r, ρ, r_0	32	Canonical little-endian, value $< \ell$
Group element C	32	Canonical compressed Ristretto; identity is 32 zero bytes
Address (pool, owner, sender, recipient, account)	32	Raw public-key bytes
Sequence, key epoch	8	Unsigned 64-bit little-endian
Memo	32	UTF-8 or raw bytes, zero-padded
Hash	32	SHA-256 digest

A public amount enters the group as a scalar. The program's `public_commitment(a)` writes $\text{LE}_8(a)$ into the first eight bytes of a zeroed 32-byte scalar and multiplies G by it, which yields aG . In memory, the SDK holds amounts, sequences and epochs as exact `bigint` values in $[0, 2^{64})$, and JSON carries them as decimal strings, so no value passes through a floating-point number.

4.2 Pedersen commitments

4.2.1 Definition

A Pedersen commitment to an amount v with blinding r is

$$C = vG + rH, \quad v \in [0, 2^{64}), \quad r \in \mathbb{Z}_\ell.$$

The pair (v, r) is the opening. The SDK's `commit(amount, { blinding, memo })` returns the 32-byte commitment together with a `NoteOpening` of amount, blinding and a 32-byte padded memo. The memo travels with the opening inside envelopes, but it is not part of the commitment; only v and r determine C . When $v = 0$ or $r = 0$, the SDK uses the identity for the corresponding term instead of multiplying by zero, so every amount and blinding in range has a defined commitment.

4.2.2 Hiding

Pedersen commitments are perfectly hiding. Fix a commitment $C = vG + rH$ and an arbitrary alternative amount v' . Since $H = \eta G$ for some nonzero η , the equation $v'G + r'H = C$ has exactly

one solution,

$$r' = r + (v - v')\eta^{-1} \pmod{\ell}.$$

When r is drawn uniformly from $\mathbb{Z}_\ell$, the commitment C is a uniformly distributed group element whatever v is, and every candidate amount is consistent with it through exactly one blinding. An adversary with unbounded computation, one that knows η , still learns nothing about v from C . This is the property that lets commitments sit in a permanent public archive.

4.2.3 Binding

Binding is computational. Two different openings of the same commitment,

$$vG + rH = v'G + r'H, \quad (v, r) \neq (v', r'),$$

give $(v - v')G = (r' - r)H$ and therefore $\eta = (v - v')(r' - r)^{-1}$. Opening a commitment in two ways is as hard as computing the discrete logarithm of H with respect to G . Binding is what makes a range proof about C a statement about the amount the committer actually holds. It is a soundness property: it must hold when a proof is verified. A later break of the discrete-logarithm problem lets an adversary open old commitments to any value, but the commitments still carry no information about the amounts behind them.

4.2.4 Fresh canonical random blindings

Every note blinding is sampled fresh by `randomBlinding()` with exact rejection sampling over canonical scalars:

$$\begin{aligned} s &\xleftarrow{\$} \{0, 1\}^{256}, \\ x &= \text{int}_{\text{LE}}(s) \bmod 2^{253}, \\ r &= x \text{ if } x < \ell, \text{ otherwise draw a new } s. \end{aligned}$$

The reduction modulo 2^{253} is implemented by masking the top three bits of byte 31 (`candidate[31] & 31`). Since ℓ lies just above 2^{252} , a candidate is accepted with probability $\ell/2^{253}$, slightly above one half, and the accepted value is exactly uniform over $\mathbb{Z}_\ell$. Rejected candidates are zeroed before the next draw. The randomness comes from the platform CSPRNG.

A blinding is never reused. If two notes shared r , their difference $C_1 - C_2 = (v_1 - v_2)G$ would carry no blinding at all, and the relation between the two amounts would no longer be hidden. Fresh, independent openings are also what keeps a balance blinding secret: every transfer subtracts a new uniform r from the sender's balance blinding, and every accepted note adds one to the payee's.

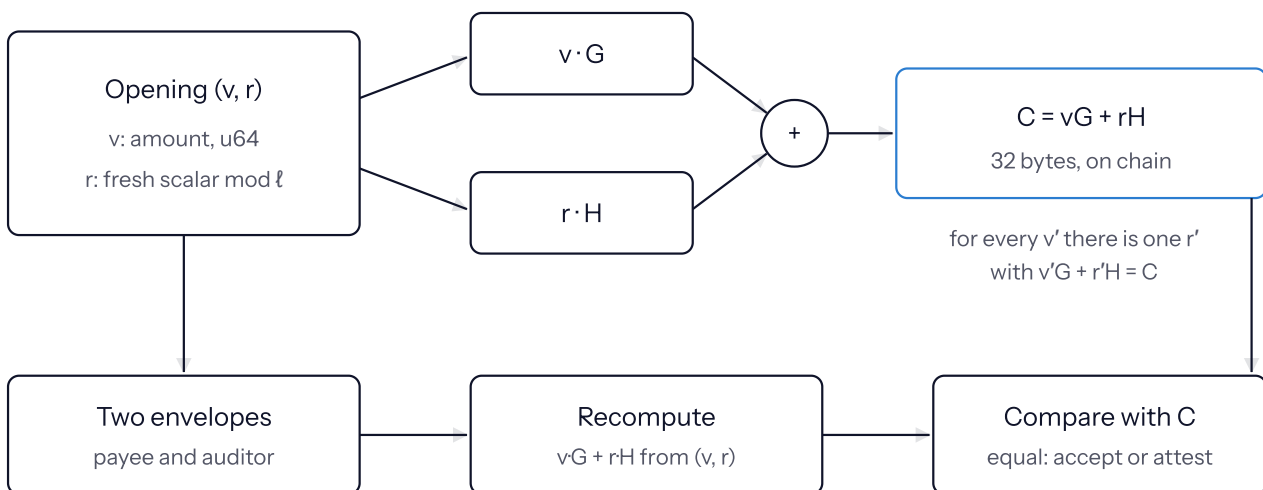
4.2.5 Full 64-bit amounts

No party ever recovers an amount by solving a discrete logarithm. The payee and the auditor read v directly from the envelope plaintext, and an owner reads its balance from its self-record. Amounts can therefore use the full unsigned 64-bit range, 0 to 18,446,744,073,709,551,615 base units, with no 48-bit cap and no split into 16-bit and 32-bit halves as in Token-2022 confidential transfers. Every range proof in the protocol uses a bit length of 64, and the SDK accepts values above JavaScript's safe-integer limit without loss.

4.2.6 Checking an opening

An opening is checked by recomputation. `checkOpening(C, (v, r, memo))` decodes C as a valid group element, recomputes $vG + rH$ from the opening, and compares the two encodings byte for byte. The same check guards every point at which an opening enters or leaves a client:

- `seal` refuses with `OPENING_MISMATCH` to encrypt an opening that does not match the note commitment it will be bound to.
- `open` returns an opening only after it has matched the note commitment supplied in its context.
- The range-proof wrapper recomputes each commitment from its amount and blinding before proving.
- `sealSelfRecord` and `openSelfRecord` check the balance opening against the balance commitment.
- The auditor service checks the recovered opening before it signs an attestation.



The commitment and its opening. The opening (v, r) determines the 32-byte commitment $C = vG + rH$, which is all the chain stores. Because every alternative amount v' fits C with exactly one blinding r' , C reveals nothing about v . The opening reaches its two recipients in envelopes, and each recipient recomputes $vG + rH$ and compares it with C before it accepts or attests.

4.3 Balances

4.3.1 The balance commitment

Each owner holds one confidential account per pool, a ConfAccount at the PDA seeds `["acct", pool, owner]`. It stores the balance as a commitment $C_{bal} = bG + \rho H$, together with the encrypted self-record that carries (b, ρ) for the owner. The program's 190-byte allocation is laid out as follows.

Offset	Length	Field
0	8	Anchor account discriminator
8	32	<code>pool</code>
40	32	<code>owner</code>
72	32	<code>balance_commitment</code> (C_{bal})
104	68	<code>self_record</code>
172	8	<code>sequence</code> (u64 LE)
180	8	<code>key_epoch</code> (u64 LE)
188	1	<code>frozen</code>
189	1	<code>bump</code>

Only the owner can change C_{bal} . Every balance-changing instruction (`deposit`, `transfer`, `accept`, `withdraw`) is signed by the owner and runs `ensure_mutable`, which requires the account not to be frozen (`AccountFrozen`), its sequence to be below the last value of its 32-bit range (`SequenceRangeExhausted`) and the caller's `expected_sequence` to equal the stored `sequence` (`StaleSequence`). `update_balance` then increments the sequence with checked arithmetic and writes the new commitment and the new self-record together. A range proof built against C_{bal} cannot be made stale by anyone else's transaction, because no one else can move that commitment. Incoming value waits in Notes until the owner accepts it.

4.3.2 The initial zero balance

`open_account` sets the initial balance to a commitment to zero with a public, deterministic blinding r_0 , derived by the program from the pool, the owner and the owner's key epoch e (the epoch of the KeyRecord the account is opened with). For a retry counter $j = 0, 1, 2, \dots$:

$$d_j = \text{SHA-256}(\delta_{init} \parallel \text{pool} \parallel \text{owner} \parallel \text{LE}_8(e) \parallel \text{LE}_4(j)),$$

$$r_0 = \text{int}_{\text{LE}}(d_j) \bmod 2^{252}, \quad \text{for the smallest } j \text{ with } r_0 \neq 0,$$

$$C_0 = 0 \cdot G + r_0 H = r_0 H.$$

The preimage is 113 bytes: the 37-byte ASCII domain string, the raw 32-byte pool address, the raw 32-byte owner address, the epoch as eight little-endian bytes and the counter as four little-endian bytes. The reduction modulo 2^{252} is implemented by clearing the high four bits of digest byte 31 (`digest[31] &= 0x0f`). The result is canonical without further checks, because $2^{252} < \ell$. An all-zero result is retried with the next checked u32 counter, and the program returns `InitialBalanceDerivationFailed` if the counter is exhausted. The program computes `initial_balance_blinding` and `initial_balance_commitment` itself, so a caller supplies only the encrypted self-record and cannot select an initial commitment. The SDK's `deriveInitialBalanceOpening({ pool, owner, keyEpoch })` returns the same opening $(0, r_0)$, so the client can seal its first self-record against C_0 before the account exists.

The blinding is deliberately public. The account's amount is known to be zero, so its opening hides nothing, and every later public deposit is visible anyway. The reason for a nonidentity start lies in the native proof format. The ZK ElGamal Proof program reads the commitments of a batched context up to the first all-zero encoding, which marks the unused slots, so an identity commitment cannot be an active proof statement. If accounts started at the identity, an owner who deposited a and later withdrew all of it would leave a remaining balance of $0 \cdot G + 0 \cdot H$, the identity, which the mandatory U64 proof could not cover. Starting at $r_0 H$ leaves $r_0 H$ after such a withdrawal, a nonidentity commitment to zero that the proof covers in the normal way. The program still rejects any proof statement that equals the identity (`IdentityProofContextUnsupported`), with no exception that skips verification.

r_0 is used only for this known-zero initial state, never as a note blinding. Before an account's first confidential operation, its amount is the public total of its deposits and withdrawals. The first fresh note blinding subtracted from it (by a transfer) or added to it (by an accept) makes the balance blinding secret to everyone who does not hold that note's opening.

4.3.3 Homomorphic updates

Each instruction updates the balance commitment by group operations alone, and the owner's client applies the matching update to the opening with the SDK's opening helpers. The program never sees an opening.

Instruction	Commitment update (program)	Opening update (SDK helper)	Proof required
<code>open_account</code>	$C_{bal} \leftarrow r_0 H$	$(0, r_0)$, <code>deriveInitialBalanceOpening</code>	None
<code>deposit(a)</code>	$C_{bal} \leftarrow C_{bal} + (a - f)G$, with f the ramp fee	$(b + a - f, \rho)$, <code>openingAfterDeposit</code>	None; a is public
<code>transfer</code>	$C_{bal} \leftarrow C_{bal} - C_{note}$	$(b - v, \rho - r)$, <code>openingAfterTransfer</code>	U128 over $[C_{note}, C_{bal} - C_{note}]$
<code>accept</code>	$C_{bal} \leftarrow C_{bal} + \sum_k C_{note,k}$	$(b + \sum_k v_k, \rho + \sum_k r_k)$, <code>openingAfterAccept</code>	None; each opening checked by the client
<code>return_note</code>	Unchanged	(b, ρ) , <code>openingAfterReturn</code>	None
<code>withdraw(a)</code>	$C_{bal} \leftarrow C_{bal} - aG$	$(b - a, \rho)$, <code>openingAfterWithdraw</code>	U64 over $[C_{bal} - aG]$
<code>close_account</code>	Account closed; requires $C_{bal} = \rho H$	$(0, \rho)$, <code>generateZeroBalanceProof</code>	Schnorr proof of knowledge of ρ

Blinding arithmetic is in $\mathbb{Z}_\ell$; amount arithmetic is exact unsigned 64-bit arithmetic.

Deposit. The amount is public, and so is its ramp fee f . The program requires $0 < a \leq \text{deposit_cap}$ and that the pool's public `principal` plus $a - f$ stay within `pool\_cap`. It moves $a - f$ from the owner's token account into the vault and f into the protocol fee account with classic SPL Token transfers, and adds the public commitment:

$$C'_{bal} = C_{bal} + (a - f)G = (b + a - f)G + \rho H.$$

The blinding is unchanged, so `openingAfterDeposit` returns $(b + a - f, \rho)$.

Transfer. The payer chooses v , draws a fresh r , and forms $C_{note} = vG + rH$. The program checks that C_{note} is a valid Ristretto encoding (`InvalidCommitment`) and computes the remaining balance from the stored commitment, never from a caller-supplied field:

$$C'_{bal} = C_{bal} - C_{note} = (b - v)G + (\rho - r)H.$$

It then requires the preceding instruction to have proven both C_{note} and C'_{bal} to lie in $[0, 2^{64})$, creates the Note holding C_{note} , and stores C'_{bal} . `openingAfterTransfer` returns $(b - v, \rho - r \bmod \ell)$ and fails if $b < v$, so an honest client never builds a transfer it could not prove.

Accept. The recipient passes its Notes as `(Note, original sender)` account pairs. For each pair, the program validates the Note's PDA from its stored seeds and bump, requires the Note's current `recipient` to be the signer (`InvalidRecipient`) and the rent destination to be the Note's `sender` (`InvalidRentDestination`), and enforces the pool's attestation policy (`AttestationRequired`). It rejects duplicate Notes in a batch (`DuplicateNote`). It then adds every commitment and closes every Note, refunding rent to the sender:

$$C'_{bal} = C_{bal} + \sum_{k=1}^n C_{note,k} = \left(b + \sum_k v_k\right)G + \left(\rho + \sum_k r_k\right)H.$$

`openingAfterAccept` takes only openings that have already been recovered from envelopes and checked against their on-chain commitments. It adds them in $\mathbb{Z}_\ell$ and checks the running amount against $2^{64} - 1$ after each addition.

Return. `return_note` changes no balance. It re-targets the Note to its sender (`recipient ← sender`), sets `returned`, and clears `attested`. The recipient may return a Note at any time; the sender may return it once the current slot reaches `created_slot + return_timeout_slots` (computed with saturating addition). `openingAfterReturn` preserves the sender's current opening unchanged. The sender created the Note's opening and keeps it; when it accepts the returned Note, it adds that retained (v, r) through `openingAfterAccept` like any other note.

Withdraw. The amount is public, $a > 0$, and the destination must not be the vault (`InvalidDestination`). The program computes

$$C'_{bal} = C_{bal} - aG = (b - a)G + \rho H,$$

requires a U64 proof that C'_{bal} opens to a value in $[0, 2^{64})$, pays a from the vault under the Pool PDA's signature, and decreases `principal` with checked arithmetic. `openingAfterWithdraw` returns $(b - a, \rho)$ and fails if $b < a$.

4.3.4 Ordered chains

Because only the owner moves its balance, a client can precompute a whole sequence of transfers. The SDK's payroll preparation does this for each lane, a `ConfAccount` that sends its rows in order. With $C^{(0)}$ the current balance and $(b^{(0)}, \rho^{(0)})$ its checked opening, step k produces

$$C^{(k)} = C^{(0)} - \sum_{i=1}^k C_{note,i},$$

$$(b^{(k)}, \rho^{(k)}) = \left(b^{(0)} - \sum_{i=1}^k v_i, \rho^{(0)} - \sum_{i=1}^k r_i \right),$$

with a fresh r_i per Note and a self-record sealed against $C^{(k)}$ at sequence $s_0 + k$. All intermediate commitments, openings, envelopes and self-records are fixed before the first transaction is sent. The U128 proof for step k is generated at execution over $[C_{note,k}, C^{(k)}]$. A retried transaction keeps its Note PDA, sequence, commitment, opening, envelopes and self-record, and no new payout randomness is drawn.

4.3.5 The solvency invariant

The Pool keeps a public counter, `principal`, which starts at zero, increases by $a - f$ on every deposit of a with ramp fee f and decreases by a on every withdrawal; the fee itself goes to the protocol fee account and never enters the vault. Transfers, accepts and returns leave it unchanged. Let $\mathcal{A}$ be the pool's open confidential accounts with balance openings (b_i, ρ_i) , and let $\mathcal{N}$ be its outstanding Notes with openings (v_j, r_j) . The invariant the program maintains is

$$\sum_{i \in \mathcal{A}} b_i + \sum_{j \in \mathcal{N}} v_j = \text{principal} \leq \text{vault.amount},$$

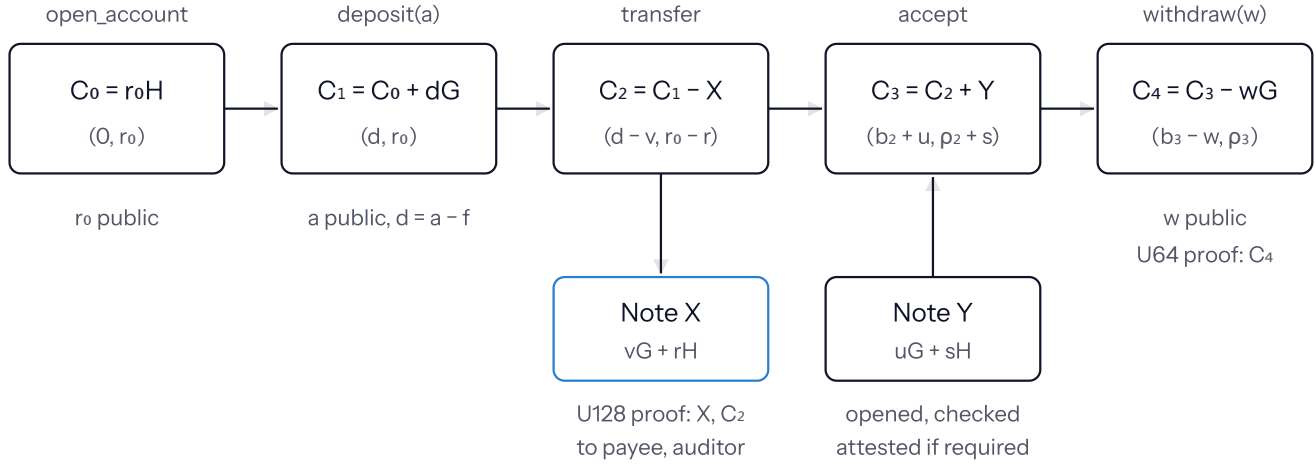
with equality on the right in every flow that receives no unsolicited token transfers. A direct SPL transfer into the vault creates public surplus, which credits no confidential balance, consumes no pool cap and cannot be withdrawn without a valid balance proof.

The commitment arithmetic gives the same statement in the group. A transfer moves C_{note} from an account to a Note, an accept moves it back into an account, only deposits and withdrawals add or remove aG , and `close_account` sets aside an account's last commitment, which its proof shows to be a multiple of H . With $\mathcal{K}$ the account generations closed so far and $C_{close,k}$ the commitment each held when it closed, summed over the pool,

$$\sum_{i \in \mathcal{A}} C_{bal,i} + \sum_{j \in \mathcal{N}} C_{note,j} + \sum_{k \in \mathcal{K}} C_{close,k} = \text{principal} \cdot G + \left(\sum_{i \in \mathcal{A} \cup \mathcal{K}} r_{0,i} \right) H.$$

Every term on the right is public, because each $r_{0,i}$ is derived from public account fields, and each $C_{close,k}$ is a commitment the chain held, so the identity can be checked from chain state alone. With binding, and with every $C_{close,k}$ proven to open to zero, it gives the amount equation modulo ℓ , and the range proofs make that an equality of integers. Every note amount and every balance left by a transfer or withdrawal is proven to lie in $[0, 2^{64})$, and deposits and accepts only add nonnegative values. The left-

hand side is then a sum of nonnegative integers far below ℓ , so the congruence is an equality, and because `principal` is a u64, every balance stays below 2^{64} . The accepted run on a local Agave 4.3.0 validator recorded both invariants as holding: the balance and outstanding-note openings summed to `principal`, and the vault backed `principal`.



One balance across its operations. The account opens at the public commitment r_0H , receives a public deposit a , of which $d = a - f$ reaches the balance after the ramp fee f , sends Note X (opening v , r) with a U128 proof over X and the remaining balance, accepts an incoming Note Y (opening u , s) after opening and checking it, and withdraws a public w with a U64 proof over what remains. Under each commitment is the opening the owner tracks; only the owner's transactions move the commitment.

4.3.6 Closing a zero balance

`close_account` returns a `ConfAccount`'s rent to its owner and removes the account, so it must not remove value. A balance commitment opens to zero exactly when it has no G component, $C_{bal} = 0 \cdot G + \rho H = \rho H$, and the owner shows this with a noninteractive Schnorr proof of knowledge of ρ with respect to H . After a full withdrawal the owner's opening is $(0, \rho)$, and `generateZeroBalanceProof` computes

$$\begin{aligned}
 k &\xleftarrow{\$} \{1, \dots, 2^{252} - 1\}, & R &= kH, \\
 m &= \delta_{close} \parallel \text{program} \parallel \text{pool} \parallel \text{account} \parallel \text{owner} \parallel \text{LE}_8(seq) \parallel \text{LE}_8(n) \parallel C_{bal} \parallel R, \\
 c &= \text{int}_{\text{LE}}(\text{SHA-256}(m)) \bmod 2^{252}, \\
 s &= k + c\rho \bmod \ell,
 \end{aligned}$$

where seq is the account's current sequence and n the `RentRecoveryRegistry`'s close nonce. The nonce k comes from 32 bytes of the platform CSPRNG with the high four bits of byte 31 cleared, redrawn if zero, and is zeroed once s is computed. The proof is the pair (R, s) , 64 bytes. The program recomputes c from its own ID and the state of the instruction's accounts, requires s to be canonical, R to be a valid point other than the identity and C_{bal} to be valid, and checks

$$sH = R + cC_{bal}$$

with the multiply and add syscalls. An honest proof passes because $sH = kH + c\rho H = R + cC_{bal}$.

Soundness. Two accepting responses $s \neq s'$ to the same R under challenges $c \neq c'$ give $\rho = (s - s')(c - c')^{-1}$, a representation $C_{bal} = \rho H$. If the same commitment also opened to $bG + \rho'H$ with $b \neq 0$, then $bG = (\rho - \rho')H$ would give $\eta = b(\rho - \rho')^{-1}$, the discrete logarithm of H . Closing an account whose balance is not zero is therefore as hard as breaking binding, under the same assumption on G and H that the rest of the ledger rests on, with the challenge modeled as a random oracle.

Zero knowledge. The pair (R, s) can be produced without ρ : choose s and c at random, set $R = sH - cC_{bal}$, and program the hash to return c . In a real proof, $s = k + c\rho$ is masked by a fresh k that is uniform over the nonzero scalars below 2^{252} , which leave out a negligible part of $\mathbb{Z}_\ell$ because ℓ lies just above 2^{252} ; so the real and simulated pairs are statistically indistinguishable. The proof therefore reveals nothing beyond its statement that C_{bal} commits to zero: not the current blinding ρ , and not any earlier amount or blinding of the account. A nonce is never reused, for the same reason a note blinding is not: two responses under one k would give ρ by the equation above.

Binding to one close. The challenge names the program, the pool, the ConfAccount, the owner, the sequence, the close nonce, the commitment and R , so a proof verifies only for the account and state it was made for. Every `close_account` advances the close nonce, which makes each proof single-use: once any close in the deployment has advanced the nonce, its own included, the proof no longer verifies, even for an account reopened at the same PDA.

4.4 Range proofs

4.4.1 The statements

The program cannot see amounts, so it relies on two range statements, each verified by the native ZK ElGamal Proof program in the same transaction.

For a transfer, a batched proof over two 64-bit components, `VerifyBatchedRangeProofU128`, proves knowledge of openings for the note and for the remaining balance:

$$\begin{aligned} \exists (v, r), (b', \rho') : \quad & C_{note} = vG + rH, & 0 \leq v < 2^{64}, \\ & C_{bal} - C_{note} = b'G + \rho'H, & 0 \leq b' < 2^{64}. \end{aligned}$$

For a withdrawal of a public a , a single 64-bit proof, `VerifyBatchedRangeProofU64`, covers the remaining balance:

$$\exists (b', \rho') : \quad C_{bal} - aG = b'G + \rho'H, \quad 0 \leq b' < 2^{64}.$$

These statements imply that no one spends more than they hold. Suppose C_{bal} opens to an amount $b \in [0, 2^{64})$. By binding, $b \equiv v + b' \pmod{\ell}$. Since $v + b' < 2^{65} < \ell$, the congruence is an equality of integers, $b = v + b'$, and therefore $v \leq b$. Without the second component, a payer could commit to a v larger than its balance and leave a remaining commitment to $b - v + \ell$, which is a valid group element but no valid amount. The first component rules out the mirror case, a "negative" note that would add value to the payer. The withdrawal statement gives $a \leq b$ in the same way.

4.4.2 Batched Bulletproofs

Both proofs are Bulletproofs range proofs aggregated over the batch, with no trusted setup. A proof over an aggregate of n bits consists of $2 \log_2 n + 9$ group and scalar elements of 32 bytes each:

$$|\pi_{U64}| = (2 \cdot 6 + 9) \cdot 32 = 672 \text{ bytes}, \quad |\pi_{U128}| = (2 \cdot 7 + 9) \cdot 32 = 736 \text{ bytes}.$$

Native verification costs 111,000 CU for U64 and 200,000 CU for U128. The proof's Fiat-Shamir transcript uses the SDK's `batched-range-proof-instruction` domain and absorbs the full context, including all eight commitment slots and bit lengths, so a proof is valid only for the exact context it was built with. The program's proof parser is pinned to `solana-zk-sdk` 2.3.13, a version family that includes the patched range-proof transcript binding the inner-product values before the batching challenge is derived. Clients prove with `solana-zk-sdk` 7.0.1, the version the native verifier of Agave 4.3.0 depends on. Each generation uses fresh proof randomness, so two proofs of the same statement differ in their proof bytes while sharing the same context.

4.4.3 The proof instruction and its context

The proof travels as an inline instruction to the native program at `ZkE1Gama1Proof11111111111111111111111111111111`, with no accounts. Its data is a one-byte discriminator followed by the exact POD proof-data struct, context first and proof second, with no length prefix.

Offset	Length	Field
0	1	Discriminator: 6 for <code>VerifyBatchedRangeProofU64</code> , 7 for <code>VerifyBatchedRangeProofU128</code>
1	256	Eight compressed Pedersen commitments, in order
257	8	Eight bit lengths, one byte each, in the same order
265	672 or 736	U64 or U128 proof

The context is the 264 bytes at offsets 1 to 264: eight 32-byte commitments followed by eight bit lengths. The complete instruction is 937 bytes for U64 and 1,001 bytes for U128. Unused slots hold the

identity encoding, written 0 below, with a bit length of zero. The contexts Hindsight accepts are fixed:

Operation	Proof	Commitment slots	Bit lengths
transfer	U128	$[C_{note}, C_{bal} - C_{note}, 0, 0, 0, 0, 0, 0]$	$[64, 64, 0, 0, 0, 0, 0, 0]$
withdraw	U64	$[C_{bal} - aG, 0, 0, 0, 0, 0, 0, 0]$	$[64, 0, 0, 0, 0, 0, 0, 0]$

Both the SDK's proof wrapper and its transaction builder check this exact shape before a transaction is assembled: active slots nonidentity with a bit length of 64, inactive slots zero.

4.4.4 How the program checks the proof instruction

Native verification runs as its own instruction earlier in the same atomic transaction. If the proof is invalid, the transaction fails before Hindsight runs. What remains for the program is to establish that the instruction it relies on was a proof of exactly the right statement. `verify_range_context` does this from the instructions sysvar, whose canonical address the account constraints check, before any balance changes or tokens move:

- The requested proof type matches the operation: U128 with two expected commitments for `transfer`, U64 with one for `withdraw` (`InvalidProofType`).
- No expected commitment is the identity (`IdentityProofContextUnsupported`).
- The current instruction index is greater than zero (`MissingRangeProof`), and the current instruction belongs to Hindsight itself. A call through another program's CPI is rejected (`ProofCpiUnsupported`).
- The instruction at index minus one targets the ZK ElGamal Proof program (`InvalidProofProgram`) and carries no accounts. The context-state and proof-account routes are not accepted (`ProofContextStateUnsupported`).
- Its discriminator names the required proof type (`InvalidProofType`), and its data parses as the exact POD struct for that type, with no truncated or trailing bytes (`InvalidProofData`).
- Each context slot equals the expected commitment with a bit length of 64, or the identity with a bit length of zero (`ProofContextMismatch`).

For a transfer, the expected commitments are the caller's C_{note} and the program's own $C_{bal} - C_{note}$, computed from the stored balance with the subtraction syscall. For a withdrawal, the expected commitment is $C_{bal} - aG$, computed with `public_commitment(a)` and the subtraction syscall. Since the native verifier has accepted the proof for this context, and the transcript binds the context, the proof is a proof about this balance transition and no other. The sequence check binds it further to this particular state of the account.

4.4.5 Proof generation in the client

Proofs are generated in the payer's browser, in a Worker, or in a payer-controlled process. No hosted service receives a witness. The SDK ships a thin Rust wrapper compiled to WebAssembly, pinned to `solana-zk-sdk` 7.0.1, that calls the official U64 and U128 builders and verifier and reimplements no

Bulletproof, generator or transcript code. The asset's SHA-256 digest is checked before initialization, and a mismatch fails closed. Before proving, the wrapper rejects an all-zero commitment, a non-canonical scalar or an opening that does not recompute to its commitment, and after proving it verifies the proof locally and checks the strict context shape. `generateRangeProof` returns the 264-byte context and the proof bytes separately, and the transaction builder prepends the discriminator and places the proof instruction immediately before the Hindsight instruction. In a v1 transaction the compute-unit and loaded-data limits live in the message configuration, not in compute-budget instructions, so nothing intervenes between the two.

Measured end to end, a transfer with its U128 proof and both envelopes is 4,024 of the 4,096 bytes a v1 transaction allows and consumes 226,570 CU, of which native verification accounts for 200,000. A withdrawal with its U64 proof is 1,479 bytes and 140,575 CU. Both figures come from one accepted run on a local Agave 4.3.0 validator with native proof verification enabled, each transaction signed by its owner alone.

4.5 Envelopes

4.5.1 What an envelope carries

A commitment is useless to a recipient without its opening, and the opening must reach exactly two parties: the payee, who will accept the note, and the pool's auditor, who sees every amount. Hindsight carries the opening in envelopes. An envelope is a hybrid public-key encryption of a 72-byte plaintext: an X-Wing key encapsulation followed by AES-256-GCM, with the key and nonce derived by HKDF-SHA256 and with associated data that binds the ciphertext to the identity of the Note it belongs to. The program checks an envelope's length and hashes it, but it cannot check its contents, and envelopes are not proven correct. Instead, no one is credited with a commitment they cannot open, as described at the end of this chapter.

4.5.2 X-Wing keys and encapsulation

X-Wing is the hybrid KEM of the IETF draft `draft-connelly-cfrg-xwing-kem` (draft 11), combining X25519 with ML-KEM-768. The SDK uses `@noble/post-quantum 0.7.1 ( ml_kem768_x25519 )` through a provider interface, and checks its key generation, encapsulation and decapsulation against the draft's published test vectors byte for byte.

Object	Bytes	Composition
Public (encapsulation) key pk_R	1,216	ML-KEM-768 encapsulation key (1,184) and X25519 public key (32)
Ciphertext ct	1,120	ML-KEM-768 ciphertext (1,088) and X25519 ephemeral public key (32)
Private key	32	Compact seed from which both component private keys are expanded
Shared secret ss	32	X-Wing combiner output

Encapsulation runs both component KEMs against the recipient's public key. The X-Wing combiner hashes the ML-KEM-768 shared secret, the X25519 shared secret, the X25519 ciphertext, the X25519 public key and a fixed label with SHA3-256 into the 32-byte ss . The combined secret is unpredictable as long as either component holds, and against a quantum adversary the ML-KEM-768 component carries it.

Public keys live on chain. `register_key(epoch, encapsulation_key)` writes exactly 1,216 bytes (`InvalidKeyLength` otherwise) into an owner-signed KeyRecord at `["kem", owner, LE_8(epoch)]`, one immutable record per epoch. The private key is a 32-byte seed drawn from fresh entropy. It is never derived from a wallet signature, and authentication signatures never serve as X-Wing seeds. Decapsulation in the SDK first checks that the public key derived from the seed equals the supplied public key. An ML-KEM-768-only profile, with 1,184-byte public keys and 1,088-byte ciphertexts, would need its own versioned key registry and envelope format. The SDK reserves a profile selector for it and does not substitute such keys into X-Wing fields.

4.5.3 Plaintext layout

The plaintext is the serialized opening, 72 bytes of fixed size.

Offset	Length	Field
0	8	Amount v , u64 little-endian
8	32	Blinding r , canonical little-endian scalar
40	32	Memo, zero-padded

`padMemo` encodes a string as UTF-8 (or takes raw bytes) and pads it with zeros to 32 bytes. It rejects any input longer than 32 bytes, so a memo is limited to 32 bytes, not 32 characters. `decodeMemo` strips trailing zeros. Because the plaintext has a fixed size, every envelope has the same length whatever the amount or memo.

4.5.4 Associated data

Each envelope is bound to its Note through 167 bytes of associated data:

$$AAD = \delta_{env} \parallel role \parallel pool \parallel sender \parallel recipient \\ \parallel LE_8(seq) \parallel LE_8(epoch) \parallel C_{note}.$$

Offset	Length	Field
0	22	ASCII <code>hindsight:envelope:v0:</code>
22	1	Role: 0 for payee, 1 for auditor
23	32	Pool address
55	32	Sender: the payer's wallet, the Note's <code>sender</code>
87	32	Recipient: the original payee's wallet, for both roles
119	8	Sequence: the Note's <code>sequence</code> , the payer account's sequence at transfer
127	8	Key epoch of the decrypting key: payee epoch or auditor epoch
135	32	Note commitment C_{note}

The pool, sender and sequence together are the Note's PDA seeds, `["note", pool, sender, LE_8(seq)]`, so the associated data names exactly one Note, and it adds that Note's payee and commitment. The recipient field is the original payee's wallet address in both envelopes. The auditor envelope differs only in its role byte and its key epoch, and it is encrypted to the auditor's X-Wing key. The KEM public key selects who can decrypt, while the associated data says which payment the plaintext belongs to. The `transfer` instruction itself names the payee by its ConfAccount address, and clients resolve that account's owner wallet for the associated data, keeping the two addresses distinct.

4.5.5 Key and nonce derivation

The AES-256-GCM key and its 96-bit nonce are derived from each fresh shared secret with HKDF-SHA256:

$$(ct, ss) \leftarrow \text{XWing.Encaps}(pk_R), \\ salt = \text{SHA-256}(pk_R \parallel ct), \\ info = \delta_{kdf} \parallel AAD, \\ OKM = \text{HKDF-SHA256}(IKM = ss, salt, info, L = 44), \\ K = OKM[0, 32), \quad N = OKM[32, 44).$$

The salt binds the full recipient public key and the full KEM ciphertext, both their ML-KEM and their X25519 parts, into the derivation. The info string is the 42-byte ASCII domain followed by the 167-byte associated data, 209 bytes in all, so the key and nonce themselves depend on the Note's identity, the role and the key epoch, in addition to the GCM tag's check of the same associated data. The derivation outputs 352 bits, split 32/12 into K and N . A new encapsulation is performed for every envelope, so each (K, N) pair is used for exactly one encryption and no nonce needs to be transmitted. This keeps the AEAD part at 88 bytes.

4.5.6 The 1,208-byte envelope

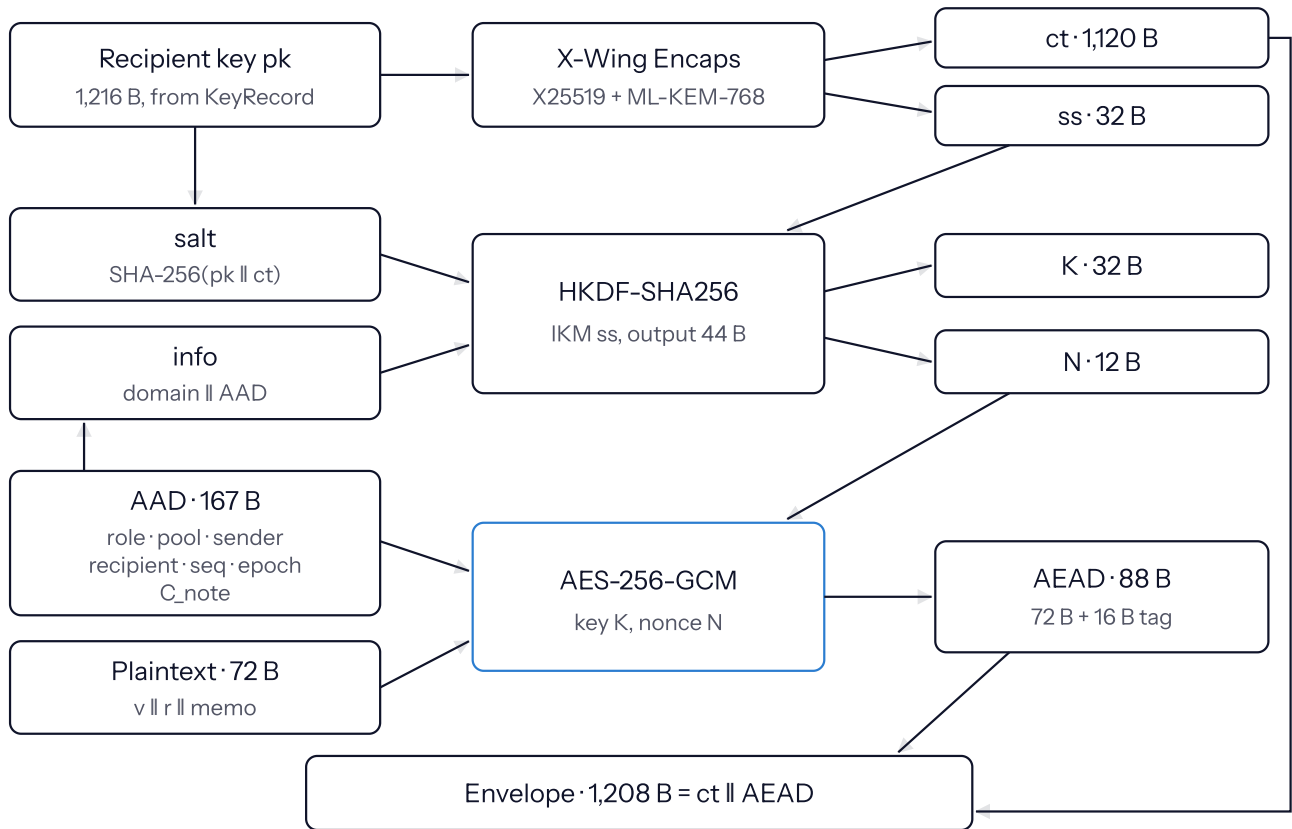
The plaintext P is encrypted under K and N with the associated data and a 128-bit tag:

$$E = ct \parallel \text{AES-256-GCM.Enc}(K, N, AAD, P),$$

$$|E| = 1,120 + 72 + 16 = 1,208 \text{ bytes.}$$

Offset	Length	Field
0	1,120	X-Wing ciphertext ct
1,120	72	AES-256-GCM ciphertext of the plaintext
1,192	16	GCM tag

The program requires both envelopes in `TransferArgs` to be exactly 1,208 bytes (`InvalidEnvelopeLength`). Together they make up 2,416 of the 2,540 bytes of `transfer` instruction data. The domain strings carry version tags (`v0` for the envelope codec), so a revised framing is a distinct, versioned format rather than a reinterpretation of existing bytes.



The envelope construction. X-Wing encapsulation to the recipient's 1,216-byte key yields a 1,120-byte ciphertext and a 32-byte shared secret. HKDF-SHA256 expands the shared secret, salted with SHA-256 of the key and ciphertext and with the associated data in its info string, into a 32-byte AES key and a 12-byte nonce. AES-256-GCM encrypts the 72-byte opening under the same associated data. The envelope is the KEM ciphertext followed by the 88-byte AEAD output.

4.5.7 Opening and checking an envelope

`open({ envelope, recipientKeyPair, context })` reverses the construction and returns an opening only if every step succeeds:

- The envelope is exactly 1,208 bytes, and the key pair is a 1,216-byte public key with the 32-byte seed it derives from.
- The associated data is rebuilt from the context, which the recipient takes from the Note on chain: `pool`, `sender`, `original_recipient`, `sequence`, the epoch for its role and `commitment`.
- The first 1,120 bytes are decapsulated to `ss`; K and N are derived as above; the remaining 88 bytes are decrypted and authenticated.
- The 72-byte plaintext is parsed, and its blinding must be canonical.
- The recovered (v, r) must satisfy $vG + rH = C_{note}$.

Failures return fixed messages with no upstream error or supplied value: `ENVELOPE_OPEN_FAILED` for key, decapsulation and authentication failures, and `OPENING_MISMATCH` when the plaintext authenticates but does not open the commitment. The shared secret and plaintext buffers are zeroed after use on a best-effort basis.

4.5.8 The envelope pair hash

Envelopes are instruction data. The Note does not store them, only a commitment to both:

$$envelope_hash = \text{SHA-256}(E_{payee} \parallel E_{auditor}),$$

computed by the program over the two envelopes in `TransferArgs`, payee first. The SDK computes the same value with `envelopePairHash`. A relay stores envelopes by this hash and serves them to recipients without being trusted. It checks exact lengths and the ordered hash on upload, and every client checks the bytes it receives against `Note.envelope_hash` before opening them, so substituted bytes are rejected before decryption. Recipients also do not depend on archival RPC history to find their envelopes.

Offset	Length	Note field
0	8	Anchor account discriminator
8	32	<code>pool</code>
40	32	<code>sender</code>
72	32	<code>recipient</code> (current; the sender after a return)
104	32	<code>original_recipient</code>
136	32	<code>commitment</code> (C_{note})
168	32	<code>envelope_hash</code>
200	8	<code>sequence</code>
208	8	<code>created_slot</code>
216	8	<code>recipient_key_epoch</code>
224	8	<code>auditor_kem_epoch</code>
232	1	<code>attested</code>
233	1	<code>returned</code>
234	1	<code>bump</code>

4.5.9 Payee and auditor roles, key epochs

The two envelopes of a transfer are sealed together by `sealToRecipients`, which requires the roles to be payee and auditor and the pool, sender, recipient, commitment and sequence of both contexts to be identical. Only the key epoch differs between the two contexts. The program pins which keys are current. The `Transfer` accounts include the payee's `KeyRecord`, which must belong to the payee `ConfAccount`'s owner (`InvalidRecipient`) and carry that account's `key_epoch` (`InvalidKeyEpoch`), and the auditor's `KeyRecord` at `["kem", pool.auditor, pool.auditor_kem_epoch]`, owned by the pool's auditor identity (`InvalidAuthority`). The `Note` records both epochs, `recipient_key_epoch` and `auditor_kem_epoch`, so each recipient knows which private key opens its envelope, including for historical `Notes` after a key has been replaced. The auditor's X-Wing keys are held per pool and per epoch in the auditor service's own trust domain, behind an HSM interface that separates opening and checking from transaction signing. The hosted API never loads that key and never opens an envelope.

4.6 Self-records

4.6.1 Purpose

An owner needs the opening (b, ρ) of its balance to build every proof and every update. The self-record stores that opening on chain, encrypted under a symmetric key only the owner holds, so a client can recover it from the `ConfAccount` alone. Every instruction that sets the balance (`open_account`, `deposit`, `transfer`, `accept`, `withdraw`) takes a new 68-byte self-record and writes it together with the new commitment. The program treats the 68 bytes as opaque; their integrity comes from the AEAD and from the opening check on recovery.

4.6.2 Layout

$$R = N_s \parallel \text{AES-256-GCM.Enc}(K_s, N_s, AAD_s, \text{LE}_8(b) \parallel \rho),$$
$$|R| = 12 + 40 + 16 = 68 \text{ bytes.}$$

Offset	Length	Field
0	12	Nonce N_s , fresh random for every seal
12	8	Encrypted balance amount b (u64 LE)
20	32	Encrypted balance blinding ρ
52	16	GCM tag

The key K_s is a 256-bit AES-GCM key generated by `generateSelfRecordKey()` as a non-extractable WebCrypto key. Like the X-Wing seed, it never comes from a wallet signature. Unlike an envelope key, it is long-lived, so every seal draws a fresh random 96-bit nonce and transmits it in the record.

4.6.3 Associated data

$$AAD_s = \delta_{rec} \parallel pool \parallel account \parallel LE_8(seq) \parallel C_{bal},$$

129 bytes: the 25-byte ASCII domain, the raw pool address, the raw ConfAccount address, the sequence and the balance commitment. The sequence and commitment are the values the account holds after the instruction that writes the record. A record sealed for `open_account` uses the account's initial sequence, `close_nonce << 32`, which is 0 until the first close in the deployment, and $C_0 = r_0 H$, and a record for any later instruction uses the incremented sequence $s + 1$ and the new C'_{bal} . The associated data therefore ties each record to one account at one point in its history: a record copied to another account, replayed at another sequence or paired with another commitment fails authentication.

4.6.4 Recovering and checking a balance

To recover its balance, a client reads its ConfAccount, takes `pool`, its own address, `sequence` and `balance_commitment` from it as the context, and calls `openSelfRecord`. The function decrypts the 56 bytes after the nonce under the rebuilt associated data, parses b and ρ , and returns them only if

$$bG + \rho H = C_{bal}.$$

A wrong key, modified ciphertext, wrong context, wrong sequence or an opening that does not match the commitment all return the same fixed failure. A successful recovery is therefore a checked statement: the client holds the exact opening of the commitment the chain stores. Payroll preparation relies on this. Before it seals any envelope, it opens every source account's self-record against its current on-chain sequence and commitment and compares the result with the opening it was given.

4.7 Confidentiality after Q-Day

4.7.1 What the archive holds

An adversary who records the chain and the relay from the first transfer onward holds, for every pool:

- every balance commitment and every note commitment, 32 bytes each;
- every range proof, 736 bytes per transfer and 672 bytes per withdrawal, with its 264-byte context;
- every envelope, 1,208 bytes, two per transfer, from instruction data or the relay;

- every self-record, 68 bytes per balance change;
- every zero-balance proof, a 32-byte nonce point and a 32-byte response per closed account;
- every KeyRecord's 1,216-byte X-Wing public key;
- the public amounts of deposits and withdrawals, the public initial blindings r_0 , and the parties and timing of every operation.

Hindsight stores no ElGamal decryption handle. In Token-2022 confidential transfers, a handle $D = rP$ sits beside each commitment, with the public key P on chain, so one discrete logarithm of P leads to every amount encrypted to that key. Here, the only paths from the archive to an amount run through the envelopes and the self-records.

4.7.2 What a quantum adversary can compute

A cryptographically relevant quantum computer computes discrete logarithms in elliptic-curve groups. Applied to the archive, this yields:

- $\eta = \log_G H$, with which it can open any commitment to any value it chooses;
- the X25519 private key behind each KeyRecord, and from it the X25519 shared secret of every envelope, since both the X25519 ciphertext and the X25519 public key are in the archive;
- the Ed25519 private scalar of every wallet and of the auditor's attestation key.

None of these values yields an amount. Knowing η breaks binding, which is a soundness property needed at verification time and does not affect hiding. The X25519 shared secret is one of the two inputs to the X-Wing combiner. The wallet and attestation keys are unrelated to the X-Wing seeds and the self-record keys, which come from fresh entropy and never from a signature.

4.7.3 Why amounts remain sealed

Commitments. For every commitment C and every candidate amount v' , there is exactly one r' with $v'G + r'H = C$. Writing $C = (v + r\eta)G$, the exponent $v + r\eta$ is uniform in $\mathbb{Z}_\ell$ for a fresh uniform r , whatever v is. Knowledge of η does not help, because the uncertainty is in r , not in the group. This holds for note commitments and, after the first confidential operation, for balance commitments, whose blindings are sums and differences of fresh note blindings.

Range proofs. Hindsight's security design treats the Bulletproofs as perfectly zero-knowledge: a proof's distribution depends only on the statement, which is the context of public commitments, so it adds nothing beyond the commitments themselves.

Zero-balance proofs. A closing proof (R, s) can be simulated from the closed commitment alone, and the simulation argument uses no computational assumption, so the proof adds nothing beyond the commitment and the public fact that the account closed at zero.

Envelopes. The AES key and nonce are derived from ss , and ss is a SHA3-256 hash of both component secrets. A quantum adversary recovers the X25519 secret but not the ML-KEM-768 secret, which would require breaking ML-KEM-768, the lattice KEM standardized in FIPS 203. Without it, ss is unknown, so K and N are unknown and the 88-byte AEAD ciphertext reveals nothing. X25519 adds a classical hedge

alongside ML-KEM-768. The envelope has a fixed size and the memo is padded, so length reveals nothing either.

Self-records. The AES-256 key never appears on chain or in any request. The record's associated data is public, but it identifies the account and the commitment, not the amount.

Archived object	Size	Classical adversary	Quantum adversary with the archive
Balance and note commitments	32 B	Perfectly hiding	Perfectly hiding; binding no longer holds
Range proofs	736 B or 672 B	Zero-knowledge	Zero-knowledge (treated as perfect)
Envelopes	1,208 B	X25519 and ML-KEM-768	ML-KEM-768 within X-Wing
Self-records	68 B	AES-256-GCM, key off chain	AES-256-GCM, key off chain
Zero-balance proofs	64 B	Zero-knowledge	Zero-knowledge
X-Wing public keys	1,216 B	Public	X25519 part yields its private key; ML-KEM-768 part does not
Deposits, withdrawals, r_0	Public	Public	Public

Confidentiality and soundness thus age differently. Confidentiality has to hold for as long as the data matters, which is why every amount-bearing object is either unconditionally hiding or protected by ML-KEM-768 from the first transfer. Soundness has to hold when a transaction is verified, so it rests on classical assumptions today and follows Solana's own post-quantum migration.

4.7.4 Nobody is credited with a commitment they cannot open

Because envelopes are not proven correct, the protocol has to rule out a balance commitment whose opening its owner does not know. Such an account could neither prove a transfer or withdrawal nor read its own balance. Hindsight rules it out by construction, and an induction over the instructions shows it:

- `open_account` sets $C_0 = r_0 H$, whose opening $(0, r_0)$ is public.
- `deposit` and `withdraw` add or subtract aG with a public a , which leaves the blinding unchanged.
- `transfer` subtracts a note whose opening the sender chose itself.
- `accept` is the only instruction that adds a commitment chosen by another party. The program executes it only on the recipient's signature, and the recipient's client signs only after it has checked

the envelope bytes against `Note.envelope_hash` , opened its envelope and checked $vG + rH = C_{note}$.

Every balance commitment in the pool therefore has an opening its owner knows, and the self-record carries it.

The attestation gate repeats the same check for the auditor. When the pool's policy sets `attest_required` , `accept` requires each Note to be `attested` . Only the pool's auditor can set that flag, through `attest` , which rejects returned Notes (`NoteAlreadyReturned`). The auditor service treats the indexer and the relay as untrusted. Before opening, and again before signing, it fetches the confirmed Note, Pool and KeyRecord accounts and checks their owners, discriminators, PDAs, pool, sender, recipients, commitment, epochs and current auditor. It then checks the ordered envelope hash, opens its own envelope with the original payee's wallet in the associated data and the Note's stored auditor epoch, and checks $vG + rH = C_{note}$. A Note whose auditor envelope fails to open or does not match is held in the `FAILED_TO_OPEN` or `OPENING_MISMATCH` state and is never attested. A payee in such a pool can be credited only with a commitment that the auditor has also opened. Holding every note opening, together with the public r_0 values and the public deposits and withdrawals, the auditor can recompute any balance in the pool, which is the visibility the auditor role is defined to have.

A note with a garbage envelope therefore harms only its sender. The payee cannot open it and returns it with `return_note` , which re-targets the Note to its sender and clears `attested` . If the payee never acts, the sender can return the Note itself after `return_timeout_slots` . The sender created the opening, so it needs neither envelope: `accept` admits a returned Note held by its own sender without attestation (`note.returned && note.recipient == note.sender`), and the sender adds its retained (v, r) to its balance opening. Every Note can therefore end in the balance of a party that holds its opening, and the solvency identity holds at every step. The accepted validator run exercised this path end to end. Both recipients opened and checked the honest envelopes, a Note with garbage envelopes was rejected, returned and accepted back by its sender, and every self-record opened and checked after each transition.

5 The on-chain program

Hindsight's on-chain component is one Anchor program, `hindsight`. It keeps every confidential balance as a Pedersen commitment, carries value between accounts as notes, and holds the pool's USDC in a vault whose authority is a program-derived address. The program holds no secret key and stores no amount in plaintext. It never decrypts anything: the opening of each note travels to the payee and the auditor inside X-Wing envelopes, and the program only checks their length and stores their hash.

The program is written against Anchor 0.31.1. Commitment arithmetic runs on the curve25519 Ristretto syscalls through `solana-curve25519`, and proof-instruction parsing uses the official `solana-zk-sdk` decoders, both pinned to 2.3.13. Range proofs are verified by the native ZK ElGamal Proof program in the same transaction. Token custody uses the classic SPL Token program; the vault is an ordinary token account with no confidential-transfer extension. The buyback burns \$HIND through that token's own program, classic SPL Token or Token-2022.

This chapter describes the program as built: its model, its seven account types, its seventeen instructions with their accounts, arguments, checks and state changes, the proof-binding and buyback-binding routines, the zero-balance proof that closes an account, the solvency accounting and the error codes.

5.1 Program model

5.1.1 Commitments and generators

A value v (an unsigned 64-bit amount of USDC base units, six decimals) with blinding scalar r is committed over Ristretto255 as

$$C = v \cdot G + r \cdot H$$

where G is the standard Ristretto basepoint and H is the Ristretto map of SHA3-512 of the compressed G . These are the Pedersen generators of `solana-zk-sdk`, so a commitment built in the client, checked by the native proof verifier and updated by the program is the same group element in all three places. The program embeds both points as constants.

Constant	Compressed point (hex)
PEDERSEN_G	e2f2ae0a6abc4e71a884a961c500515f58e30b6aa582dd8db6a65945e08d2d76
PEDERSEN_H	8c9240b456a9e6dc65c377a1048d745f94a08cdb7f44cbcd7b46f34048871134

The program never learns v or r for a note. It works with two kinds of commitment. A public amount a , as in a deposit or withdrawal, becomes `public_commitment(a)` = $a \cdot G$, computed with the multiply syscall from the little-endian bytes of a . A confidential amount arrives as a 32-byte compressed point that the program validates and then adds or subtracts. The syscalls cost 521 CU for an addition and 2,208 CU for a multiplication.

Each balance-changing instruction is a homomorphic update of the stored balance commitment C_{bal} :

Instruction	Balance update	Range statement verified in the same transaction
deposit	$C'_{bal} = C_{bal} + (a - f) \cdot G$, with f the ramp fee	none; a is public and checked to be positive
transfer	$C'_{bal} = C_{bal} - C_{note}$	C_{note} and C'_{bal} both open to values in $[0, 2^{64})$
accept	$C'_{bal} = C_{bal} + \sum_i C_{note,i}$	none; each note was range-checked when it was created
withdraw	$C'_{bal} = C_{bal} - a \cdot G$	C'_{bal} opens to a value in $[0, 2^{64})$

Because commitments are additively homomorphic, the openings follow the same arithmetic. If C_{bal} opens to (b, ρ) and C_{note} to (v, r) , then $C_{bal} - C_{note}$ opens to $(b - v, \rho - r)$ with the blinding taken modulo the group order. The client tracks these openings; the program tracks only the points.

5.1.2 Owner-only balances and sequence numbers

Only the owner of a confidential account can change its balance commitment. Every balance-changing instruction (`deposit` , `transfer` , `accept` , `withdraw`) is signed by the owner, names the owner's account through its PDA seeds, and carries an `expected_sequence` that must equal the account's stored `sequence` . On success the program increments the sequence with checked arithmetic and replaces the account's encrypted self-record.

Two properties follow. A range proof is computed against one specific C_{bal} , and since no other party can change that commitment, no other transaction can invalidate a proof the owner has built. And a transaction built against an older state fails with `StaleSequence` instead of being applied to a balance its author did not see. Incoming value never writes to the recipient's balance directly; it waits in a Note until the recipient's own `accept` , which is itself an owner-signed, sequence-checked update.

A sequence has two halves. `open_account` sets it to `close_nonce << 32`, which puts the RentRecoveryRegistry's global close nonce in the high 32 bits, and each balance update counts up in the low 32 bits. Each opening of an account at its PDA, a generation, therefore has its own range of 2^{32} sequence values above the ranges of its earlier generations, and a balance update refuses to leave that range (`SequenceRangeExhausted`). Before any account has been closed the nonce is zero and sequences start at 0.

5.1.3 Notes as pending value

A transfer debits the sender immediately and creates a Note holding C_{note} , an account of its own that plays the part of a pending balance. The recipient accepts it after checking its opening, or returns it; when the pool requires attestation, the auditor marks it attested first.

5.1.4 Proof binding

`transfer` and `withdraw` require the instruction immediately before them to be a native range-proof verification whose context names exactly the commitments of this balance transition, computed by the program from stored state. The routine is described step by step in its own section below.

5.1.5 Public principal

The pool keeps one public counter, `principal` : deposits less withdrawals, in base units. Transfers, attestations, acceptances and returns leave it unchanged. It is what the pool cap limits, and it is the quantity the confidential balances and outstanding notes sum to.

5.1.6 Protocol fees and the buyback

The ramp fee never enters the vault or a commitment. `deposit` and `withdraw` pay it into the protocol fee account, a token account of the program address `["protocol"]`, and the fees leave that account only through a buyback. A buyback is one transaction: `claim_buyback` hands an exact amount to the configured keeper, one or more swap instructions turn all of it into \$HIND delivered to the protocol's own token account, and `complete_buyback`, as the final instruction, burns everything that arrived. The BuybackConfig fixes the token, the keeper and the minimum output once. The instructions, and the routine that binds a claim to its completion, are described below.

5.1.7 Rent recovery

`close_account` closes an owner's ConfAccount at a zero balance, shown by a Schnorr proof of knowledge of ρ in $C_{bal} = \rho \cdot H$, and `close_key_record` closes a KeyRecord that no pool names as its auditor key; both pay every lamport of the account to its owner. The global RentRecoveryRegistry holds the close nonce each close consumes and the protected auditor keys.

5.2 Encoding conventions

All program accounts and instruction arguments use Anchor's Borsh encoding.

Item	Encoding
Account prefix	8-byte Anchor discriminator
Integers	little-endian (<code>u8</code> , <code>u16</code> , <code>u32</code> , <code>u64</code> , <code>u128</code>)
<code>bool</code>	one byte, <code>0</code> or <code>1</code>
Fixed arrays, <code>Pubkey</code>	inline bytes, no prefix
<code>Vec<u8></code>	4-byte little-endian length, then the bytes
<code>Vec<Pubkey></code>	4-byte little-endian count, then 32 bytes per key
<code>Option<T></code>	1-byte tag (<code>0</code> none, <code>1</code> some), then <code>T</code> when present
Instruction data	8-byte Anchor instruction discriminator, then the Borsh arguments

PDA seeds are concatenated byte strings: literal seeds as UTF-8 (`pool` , `vault` , `kem` , `acct` , `note` , `buyback` , `rent-recovery` , `protocol`), public keys as their 32 raw bytes, and epochs and sequences as 8-byte little-endian `u64` . Every PDA the program creates uses the canonical bump, which is stored in the account as one byte and reused for later address checks.

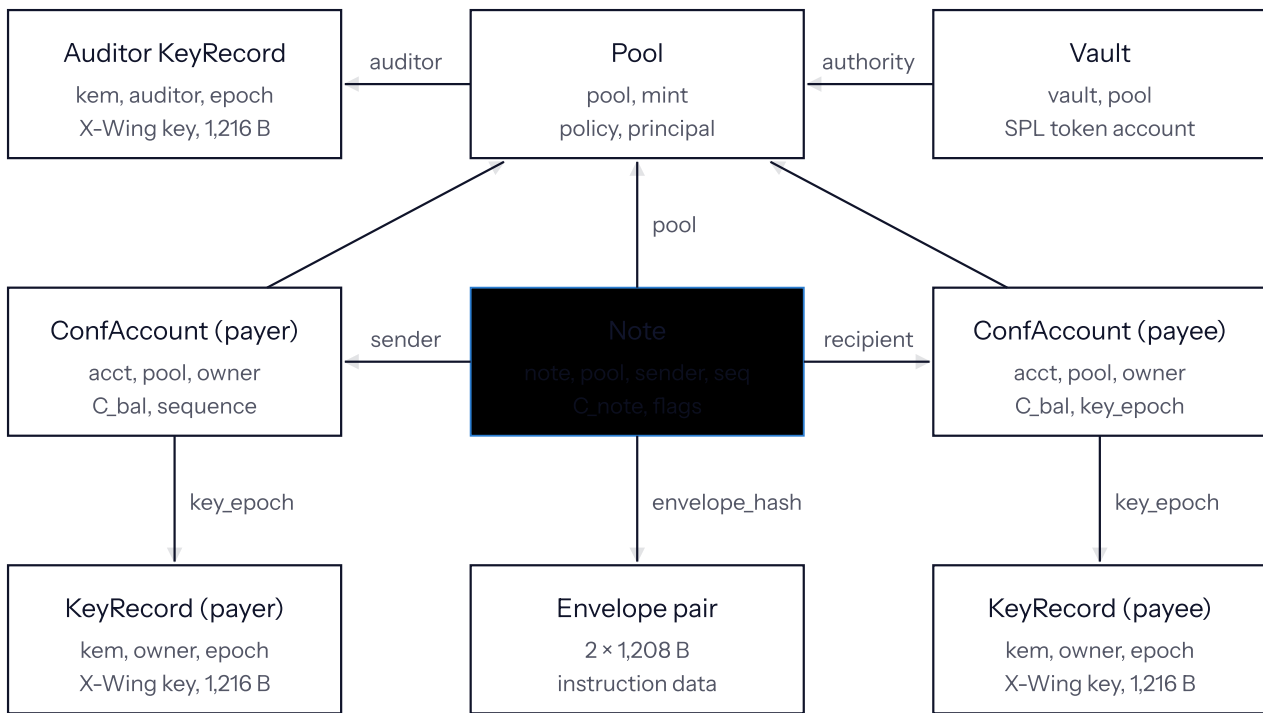
Account sizes below are the exact declared allocations, including the discriminator. An account's rent is

$$\text{rent} = (\text{bytes} + 128) \times \text{the cluster's lamports per byte}$$

read live from the Rent sysvar, and every participant account's rent returns to its payer when it closes: a Note's on `accept` , a ConfAccount's on `close_account` and a KeyRecord's on `close_key_record` .

5.3 Accounts

The program owns six account types (Pool, KeyRecord, ConfAccount, Note, BuybackConfig, RentRecoveryRegistry) and creates one SPL token account (the vault). The pool's accounts and their references are shown below; the BuybackConfig and the RentRecoveryRegistry are global, and of the pool's accounts only the auditor KeyRecord is named by the registry.



Program accounts and their references. Each arrow points from an account to the account its fields or seeds name. Every ConfAccount and Note names its Pool, and the vault's token authority is the Pool PDA. A Note's seeds name its sender and the sender's sequence, and its recipient field names the payee wallet. A ConfAccount's key_epoch selects its owner's KeyRecord; the Pool's auditor and auditor_kem_epoch select the auditor's KeyRecord. The envelope pair is instruction data, not an account: the Note keeps its SHA-256 hash.

5.3.1 Pool

One Pool exists per token mint. It records the mint and vault, the auditor identity and the auditor's KEM epoch, the two governance signers, the policy, and the public principal.

- **Seeds:** ["pool", mint]
- **Size:** 238 bytes
- **Discriminator:** f19a6d0411b16dbc

Field	Type	Offset	Size	Meaning
discriminator	[u8; 8]	0	8	Anchor account discriminator
mint	Pubkey	8	32	The pool's token mint; must have 6 decimals
vault	Pubkey	40	32	Address of the vault token account
auditor	Pubkey	72	32	Auditor identity: the only key that can sign <code>attest</code> , and the owner of the auditor <code>KeyRecord</code>
auditor_kem_epoch	u64	104	8	Epoch of the auditor <code>KeyRecord</code> that every transfer must reference
admin_authority	Pubkey	112	32	Signer that created the pool; must be the program's upgrade authority at <code>init_pool</code>
freeze_authority	Pubkey	144	32	The only key that can sign <code>freeze</code> and <code>thaw</code>
policy	PoolPolicy	176	53	The pool's policy (next table)
principal	u64	229	8	Deposits less withdrawals, in base units; excludes tokens sent to the vault outside <code>deposit</code>
bump	u8	237	1	Canonical PDA bump

The pool's auditor, its KEM epoch, its authorities and its policy are written once, by `init_pool`. No instruction rewrites them. A `Note` records the auditor epoch it was sealed to, so the key that opens each auditor envelope is fixed by the chain.

5.3.2 Pool policy

`PoolPolicy` is a 53-byte struct inside the `Pool`. At initialization every field is supplied as an `Option` in `PoolPolicyConfig`, and `init_pool` fails unless every field is present; `Some(false)` is a valid attestation policy. No field has a default in the program.

Field	Type	Offset in Pool	Size	Meaning	Rule at init_pool
attest_required	bool	176	1	When true, <code>accept</code> takes only notes the auditor has attested (and a sender's own returned notes)	must be present
return_timeout_slots	u64	177	8	Slots after a note's creation from which its sender may return it	at most $2^{32} - 1$
deposit_cap	u64	185	8	Largest single deposit, base units	at most <code>pool_cap</code>
pool_cap	u64	193	8	Largest <code>principal</code> the pool may hold, base units	must be present
ramp_fee_bps	u16	201	2	Fee in basis points on deposits and withdrawals, whose amounts are public	at most 100; pools are created at 5
flat_transfer_fee	u64	203	8	Flat fee per transfer note, base units	zero
auditor_rotation_interval_seconds	u64	211	8	The age at which the auditor console reports an epoch as due for renewal	greater than zero
admin_multisig_threshold	u16	219	2	Signer threshold of the multisig that holds the	greater than zero

Field	Type	Offset in Pool	Size	Meaning	Rule at init_pool
				admin and freeze authorities	
admin_timelock_seconds	u64	221	8	Timelock the governance system applies to admin actions	must be present

The two fee fields put fees where amounts are already public or where a fixed price leaks nothing: a basis-point charge on deposits and withdrawals, and a flat charge per note. The ramp fee is in force: on an amount a it is $f = \lfloor a \cdot \text{ramp_fee_bps} / 10\,000 \rfloor$, computed in 128-bit arithmetic and rounded down so the participant keeps the remainder. The flat transfer fee is zero, and `init_pool` refuses any other value with `FeeCustodyNotConfigured`.

All of the ramp fee goes to the **protocol fee account**: the associated token account, for the pool's mint, of the program address with seeds `["protocol"]` (`protocol_authority`). No key signs for that address, and `deposit` and `withdraw` require the fee account by its derived address, so every fee lands there. The fees collected there buy \$HIND, the project's token, on the market, and the program burns what they buy: tokens leave the fee account only through `claim_buyback`, inside a transaction that must end with `complete_buyback` (see the protocol accounts and the buyback instructions below).

The last three fields describe the off-chain governance and key-rotation regime that surrounds the pool. The admin and freeze authorities are signer addresses intended to be held by an external multisig with a timelock; the program checks their signatures, and the threshold and timelock are recorded so the regime is published with the pool. The rotation interval is read by the auditor service. The program does not convert seconds to slots and no instruction evaluates these three fields.

The return timeout is measured in slots. Its upper bound, $2^{32} - 1$, is an engineering bound on the configured delay, not a default.

5.3.3 Vault

The vault is a classic SPL Token account for the pool's mint, created by `init_pool`.

- **Seeds:** `["vault", pool]`
- **Size:** 165 bytes (SPL Token account)
- **Mint:** the pool's mint. **Token authority:** the Pool PDA.

The Pool PDA has no private key, so tokens leave the vault only through `withdraw`, where the program signs the token transfer with the seeds `["pool", mint, bump]`. Every instruction that touches the vault checks its address against `pool.vault`, its PDA seeds, that its token authority is the Pool and that its mint is the pool mint.

5.3.4 KeyRecord

A KeyRecord publishes one X-Wing encapsulation key (X25519 + ML-KEM-768) for one owner and one epoch. Payers seal the payee envelope to the payee's key, and the auditor envelope to the auditor's key.

- **Seeds:** `["kem", owner, epoch]` with `epoch` as `u64` little-endian
- **Size:** 1,265 bytes (rent paid by the owner and returned to it by `close_key_record`)
- **Discriminator:** `4703e028106f1ce6`

Field	Type	Offset	Size	Meaning
discriminator	<code>[u8; 8]</code>	0	8	Anchor account discriminator
<code>owner</code>	<code>Pubkey</code>	8	32	Wallet that registered the key
<code>epoch</code>	<code>u64</code>	40	8	Key epoch chosen by the owner
<code>encapsulation_key</code>	<code>[u8; 1216]</code>	48	1,216	X-Wing encapsulation key
<code>bump</code>	<code>u8</code>	1264	1	Canonical PDA bump

A KeyRecord is immutable: while it exists, the same owner and epoch cannot be registered again, and a new key is a new epoch at a new address. Its owner can close it with `close_key_record` , which frees the address for a later `register_key` of the same epoch, unless the RentRecoveryRegistry lists it as a pool's auditor key. Key records belong to wallets, not to pools, so one KeyRecord can serve accounts in several pools and the auditor's KeyRecord exists independently of the Pool that names it. The X-Wing decapsulation key never reaches the chain; it is a 32-byte seed held by the owner's client, or by the auditor's custody.

5.3.5 ConfAccount

A ConfAccount is one owner's confidential balance in one pool.

- **Seeds:** `["acct", pool, owner]` , so each owner has one ConfAccount per pool
- **Size:** 190 bytes (rent paid by the owner and returned to it by `close_account`)
- **Discriminator:** `850152477f986806`

Field	Type	Offset	Size	Meaning
discriminator	[u8; 8]	0	8	Anchor account discriminator
pool	Pubkey	8	32	Pool the account belongs to
owner	Pubkey	40	32	Wallet that controls the account
balance_commitment	[u8; 32]	72	32	C_{bal} , compressed Ristretto point
self_record	[u8; 68]	104	68	Owner's encrypted record of the balance opening
sequence	u64	172	8	The close nonce at opening in the high 32 bits, the count of balance updates in the low 32 bits; must match <code>expected_sequence</code>
key_epoch	u64	180	8	Epoch of the owner's KeyRecord that transfers to this account must use
frozen	bool	188	1	Set by <code>freeze</code> , cleared by <code>thaw</code>
bump	u8	189	1	Canonical PDA bump

The self-record lets the owner recover its balance opening from chain state alone. Its 68 bytes are a 12-byte nonce followed by the AES-256-GCM encryption of the 8-byte balance and the 32-byte blinding scalar, with its 16-byte tag. The authenticated data binds it to the exact state it describes:

$$AAD = \text{"hindsight:self-record:v1:"} \parallel pool \parallel account \parallel \text{le}_{64}(sequence) \parallel C_{bal}$$

The key is an AES-256 key held by the owner's client. The program stores the record and replaces it on every balance update; it does not interpret it. A client opening a self-record checks that the recovered opening reproduces C_{bal} before using it.

`key_epoch` is set by `open_account` from the KeyRecord passed to it. Every transfer to the account must present the KeyRecord of that epoch, so payers always seal to the key the account was opened with.

5.3.6 Note

A Note is one confidential payment in flight: the commitment, who sent it, who may accept it, and which keys its envelopes were sealed to.

- **Seeds:** ["note", pool, sender, sequence], where sender is the sender's wallet and sequence is the sender account's sequence before the transfer, as u64 little-endian
- **Size:** 235 bytes (rent paid by the sender and refunded to the sender when the Note closes)
- **Discriminator:** cb4bfcc451d27a7e

Field	Type	Offset	Size	Meaning
discriminator	[u8; 8]	0	8	Anchor account discriminator
pool	Pubkey	8	32	Pool the note belongs to
sender	Pubkey	40	32	Wallet whose account was debited; receives the rent on close
recipient	Pubkey	72	32	Wallet that may accept the note now: the payee, or the sender after a return
original_recipient	Pubkey	104	32	Payee wallet at creation; unchanged by a return
commitment	[u8; 32]	136	32	$C_{note} = v \cdot G + r \cdot H$
envelope_hash	[u8; 32]	168	32	SHA-256 of the payee envelope followed by the auditor envelope
sequence	u64	200	8	Sender's sequence at creation; a seed of the address
created_slot	u64	208	8	Slot of the transfer; start of the sender's return timeout
recipient_key_epoch	u64	216	8	Payee KeyRecord epoch the payee envelope was sealed to
auditor_kem_epoch	u64	224	8	Auditor KeyRecord epoch the auditor envelope was sealed to
attested	bool	232	1	Set by attest ; cleared by return_note
returned	bool	233	1	Set by return_note
bump	u8	234	1	Canonical PDA bump

The envelope hash is

$$h = \text{SHA256}(E_{\text{payee}} \parallel E_{\text{auditor}})$$

in that order, over the two 1,208-byte envelopes exactly as they appeared in the `transfer` instruction data. The envelopes are not stored in the account. Anyone who holds the two envelopes, from the transaction itself or from the relay, can check them against the Note before opening them. The Note keeps `original_recipient` and both key epochs so that the right keys can still be found after a return has re-targeted it.

A sender's sequence only increases, across a close and a reopening too, since a reopened account starts at the current close nonce shifted into the high 32 bits, above every sequence of its earlier generations. The pair (sender, sequence) is therefore never repeated within a pool, and no Note address is ever reused, even after the earlier Note at a lower sequence has closed.

The flags give each Note one of four observable states:

State	<code>attested</code>	<code>returned</code>	<code>recipient</code>	Next possible actions
pending	false	false	payee	<code>attest</code> by the auditor; <code>accept</code> by the payee when <code>attest_required</code> is false; <code>return_note</code> by the payee, or by the sender after the timeout
attested	true	false	payee	<code>accept</code> or <code>return_note</code> by the payee; <code>return_note</code> by the sender after the timeout
returned	false	true	sender	<code>accept</code> by the sender
accepted	account closed			none

5.3.7 BuybackConfig

One BuybackConfig exists per deployment. It fixes what the protocol fees buy and who cranks the buyback, and it carries the burn counter and the claim of a buyback in progress. Pools can have different fee mints; the bought token is one global setting.

- **Seeds:** `["buyback"]`
- **Size:** 209 bytes
- **Discriminator:** `e21e278b429f99ab`

Field	Type	Offset	Size	Meaning
discriminator	[u8; 8]	0	8	Anchor account discriminator
token_mint	Pubkey	8	32	The \$HIND mint, the token the buyback buys and burns
token_program	Pubkey	40	32	That mint's token program: SPL Token or Token-2022
keeper	Pubkey	72	32	The only key that can sign <code>claim_buyback</code> and <code>complete_buyback</code>
minimum_output_numerator	u64	104	8	n : \$HIND base units of the minimum-output ratio
minimum_output_denominator	u64	112	8	d : fee-mint base units of the minimum-output ratio
total_burned	u128	120	16	\$HIND base units burned by every completed buyback
pending_fee_mint	Pubkey	136	32	Fee mint of the claim in progress
pending_amount	u64	168	8	Fee base units claimed; zero when no claim is in progress
pending_minimum_output	u64	176	8	The claim's <code>minimum_output</code>
pending_input_before	u64	184	8	The keeper fee account's balance before the claim
pending_output_before	u64	192	8	The protocol token account's balance before the swap
sequence	u64	200	8	Count of completed buybacks; must match <code>expected_sequence</code>
bump	u8	208	1	Canonical PDA bump

`configure_buyback` writes every field once, with `total_burned`, `sequence` and every pending field zero, and no instruction rewrites `token_mint`, `token_program`, `keeper` or the ratio afterwards. The ratio counts \$HIND base units per fee-mint base unit: a claim of a fee base units must name a minimum output of at least $\lceil a \cdot n / d \rceil$. `claim_buyback` fills the pending fields and `complete_buyback`, in the same transaction, clears them, so between transactions every pending field is zero.

5.3.8 RentRecoveryRegistry

One RentRecoveryRegistry exists per deployment. It holds the global close nonce, which every zero-balance proof names and every `close_account` consumes, and the auditor KeyRecords that `close_key_record` refuses to close.

- **Seeds:** `["rent-recovery"]`
- **Size:** $21 + 32N$ bytes for N protected auditor keys: 21 bytes when created, 53 with one key
- **Discriminator:** `3859032794e7079c`

Field	Type	Offset	Size	Meaning
<code>discriminator</code>	<code>[u8; 8]</code>	0	8	Anchor account discriminator
<code>close_nonce</code>	<code>u64</code>	8	8	Count of completed <code>close_account</code> calls in the deployment; must match <code>expected_close_nonce</code> in <code>open_account</code> and <code>close_account</code>
<code>auditor_keys</code>	<code>Vec<Pubkey></code>	16	$4 + 32N$	The count N as a little-endian <code>u32</code> , then from offset 20 the N protected auditor KeyRecord addresses
<code>bump</code>	<code>u8</code>	$20 + 32N$	1	Canonical PDA bump

`initialize_rent_recovery` creates the registry with `close_nonce = 0` and no keys. A protected key is the address of a pool's auditor KeyRecord, `["kem", pool.auditor, pool.auditor_kem_epoch]`, derived by the program, so a pool's key is protected whether or not the KeyRecord has been registered yet. Adding a key that is already listed changes nothing; adding a new one grows the account by 32 bytes, with the instruction's payer first topping its lamports up to the rent-exempt minimum of the new size from the Rent sysvar. `initialize_rent_recovery` adds the key of every existing pool and `init_pool` the key of each new one. No instruction removes a key, resets the nonce or closes the registry.

5.3.9 Protocol accounts

The program address `protocol_authority`, with seeds `["protocol"]`, has no data account, and no key signs for it. It is the token owner of two kinds of associated token account, each derived by the associated-token program from `[protocol_authority, token program, mint]`:

Account	Mint	Token program	Moved by
Protocol fee account	a pool's mint	SPL Token	<code>deposit</code> and <code>withdraw</code> pay the ramp fee in; <code>claim_buyback</code> moves a claimed amount out
Protocol token account	<code>BuybackConfig.token_mint</code>	<code>BuybackConfig.token_program</code>	the swap delivers the bought tokens in; <code>complete_buyback</code> burns them

The protocol fee account is a 165-byte classic SPL Token account. Both are created by operator setup through the associated-token program, and the buyback transaction also creates the protocol token account idempotently before its claim. The program signs for `protocol_authority`, with the seeds `["protocol", bump]`, in two places only: the transfer in `claim_buyback` and the burn in `complete_buyback`.

5.3.10 Allocation summary

Account	Seeds	Bytes	Paid by	Closed by
Pool	<code>pool</code> , <code>mint</code>	238	<code>init_pool</code> payer	not closed
Vault	<code>vault</code> , <code>pool</code>	165	<code>init_pool</code> payer	not closed
KeyRecord	<code>kem</code> , owner, epoch	1,265	owner	<code>close_key_record</code> , refunded to the owner
ConfAccount	<code>acct</code> , <code>pool</code> , owner	190	owner	<code>close_account</code> , refunded to the owner
Note	<code>note</code> , <code>pool</code> , sender, sequence	235	sender	<code>accept</code> , refunded to the sender
BuybackConfig	<code>buyback</code>	209	<code>configure_buyback</code> payer	not closed
RentRecoveryRegistry	<code>rent-</code> <code>recovery</code>	21 + 32 <i>N</i>	<code>initialize_rent_recovery</code> payer; growth by the <code>init_pool</code> payer	not closed

5.4 Instructions

The program has seventeen instructions. Four change a confidential balance (`deposit` , `transfer` , `accept` , `withdraw`); two create per-wallet state (`register_key` , `open_account`) and two close it (`close_account` , `close_key_record`); two change only a Note (`attest` , `return_note`); three belong to pool administration (`init_pool` , `freeze` , `thaw`); one creates the rent recovery registry (`initialize_rent_recovery`); and three run the buyback (`configure_buyback` , `claim_buyback` , `complete_buyback`).

Instruction	Discriminator	Signer	Arguments	Effect
<code>init_pool</code>	74e9c7cc739fab24	payer and admin authority	<code>InitPoolConfig</code>	Create the vault, register the auditor
<code>register_key</code>	38084361807a50d5	owner	<code>epoch</code> , <code>encapsulation_key</code>	Create
<code>open_account</code>	65985bf3b024bd65	owner	<code>self_record</code> , <code>expected_close_nonce</code>	Create ConfAc known committer sequencer <code>close_</code>
<code>deposit</code>	f223c68952e1f2b6	owner	<code>amount</code> , <code>expected_sequence</code> , <code>self_record</code>	Moves vault; C
<code>transfer</code>	a334c8e78c0345ba	owner (sender)	<code>TransferArgs</code>	C_{bal} – creates
<code>attest</code>	53947877908b75a0	pool auditor	none; Notes as remaining accounts	Sets a each N
<code>accept</code>	419646d885066b04	owner (recipient)	<code>expected_sequence</code> , <code>self_record</code> ; (Note, sender) pairs as remaining accounts	C_{bal} + closes
<code>return_note</code>	9e58488eb5d10935	payee, or sender after the timeout	none	Re-target its sender
<code>withdraw</code>	b712469c946da122	owner	<code>amount</code> , <code>expected_sequence</code> , <code>self_record</code>	C_{bal} – moves the vault
<code>close_account</code>	7dff950e6e224818	owner	<code>expected_sequence</code> , <code>expected_close_nonce</code> , <code>nonce_point</code> , <code>response</code>	Closes ConfAc owner; advances

Instruction	Discriminator	Signer	Arguments	Effect
<code>close_key_record</code>	<code>755304f9cf37303b</code>	owner	none	Closes that is i auditor owner
<code>freeze</code>	<code>ff5bcf54fbc2fe3f</code>	freeze authority	none	Sets f ConfAc
<code>thaw</code>	<code>e2f92239bd15b165</code>	freeze authority	none	Clears
<code>initialize_rent_recovery</code>	<code>759246be85359cff</code>	payer and upgrade authority	none; every existing Pool as remaining accounts	Create RentRe and pro pool's a
<code>configure_buyback</code>	<code>ff699821c0dd4a18</code>	payer and upgrade authority	<code>ConfigureBuybackArgs</code>	Create Buybac
<code>claim_buyback</code>	<code>c7fff3ecc7acc98d</code>	keeper	<code>amount</code> , <code>minimum_output</code> , <code>expected_sequence</code>	Moves the pro accour keeper claim
<code>complete_buyback</code>	<code>d04ca859f26f8d4b</code>	keeper	none	Burns a receive <code>total</code> sequer

In the account tables below, "w" marks a writable account and "s" a signer. "PDA" means Anchor re-derives the address from the listed seeds and the stored bump and rejects any other account. Every typed account is also checked for program ownership and discriminator before the instruction body runs.

5.4.1 `init_pool`

Creates the Pool for a mint and its vault, fixes the auditor, the authorities and the policy, and protects the auditor's key in the RentRecoveryRegistry.

Account	Flags	Constraint
<code>payer</code>	w, s	Funds the Pool and vault rent and any growth of the registry
<code>admin_authority</code>	s	Must equal the upgrade authority recorded in <code>program_data</code>
<code>program</code>		The Hindsight program; its ProgramData address must equal <code>program_data</code>
<code>program_data</code>		The program's upgradeable-loader ProgramData account
<code>mint</code>		The token mint
<code>pool</code>	w	Created; PDA <code>["pool", mint]</code>
<code>vault</code>	w	Created; PDA <code>["vault", pool]</code> ; token mint is <code>mint</code> , token authority is <code>pool</code>
<code>token_program</code> , <code>system_program</code> , <code>rent</code>		SPL Token, System program, Rent sysvar
<code>rent_recovery</code>	w	PDA <code>["rent-recovery"]</code>

Arguments. `config: InitPoolConfig`:

Field	Type	Meaning
<code>auditor</code>	<code>Pubkey</code>	Auditor identity
<code>auditor_kem_epoch</code>	<code>u64</code>	Epoch of the auditor's KeyRecord
<code>freeze_authority</code>	<code>Pubkey</code>	Freeze and thaw signer
<code>policy</code>	<code>PoolPolicyConfig</code>	Every policy field as an <code>Option</code>

Checks.

1. The program account's ProgramData address equals `program_data`, and that ProgramData's upgrade authority equals `admin_authority`; otherwise `InvalidBootstrapAuthority`.
2. The Pool and vault addresses are unused (Anchor `init`).
3. The mint has 6 decimals; otherwise `InvalidMint`.
4. Neither `auditor` nor `freeze_authority` is the all-zero key; otherwise `InvalidAuthority`.

5. Every policy field is present; otherwise `ConfigurationRequired` .
6. `flat_transfer_fee` is zero; otherwise `FeeCustodyNotConfigured` . `ramp_fee_bps` is at most 100; otherwise `InvalidPolicy` .
7. `admin_multisig_threshold` and `auditor_rotation_interval_seconds` are positive, `deposit_cap` does not exceed `pool_cap` , and `return_timeout_slots` is at most $2^{32} - 1$; otherwise `InvalidPolicy` .
8. The `RentRecoveryRegistry` exists at its PDA, so `init_pool` runs once `initialize_rent_recovery` has.

State changes. Writes every `Pool` field from the accounts and the configuration, with `principal = 0` , and creates an empty vault whose token authority is the `Pool`. Adds the address of the auditor `KeyRecord`, `["kem", auditor, auditor_kem_epoch]` , to the registry's `auditor_keys` unless it is already there; a new key grows the registry by 32 bytes, with the payer topping up its rent.

Binding pool creation to the program's upgrade authority means that only the holder of that authority can create the canonical pool for a mint; another signer cannot claim the address first. Because the `Pool` seeds contain only the mint, there is exactly one `Pool` per mint for the deployment. The bootstrap uses the upgradeable loader's `ProgramData`, so pools are initialized while the program is still upgradeable.

5.4.2 register_key

Publishes an X-Wing encapsulation key for the signing wallet at an epoch.

Account	Flags	Constraint
<code>owner</code>	w, s	Pays rent; becomes <code>KeyRecord.owner</code>
<code>key_record</code>	w	Created; PDA <code>["kem", owner, epoch]</code>
<code>system_program</code>		System program

Arguments. `epoch: u64` ; `encapsulation_key: Vec<u8>` .

Checks.

1. The key record address is unused, so an epoch can be registered once (Anchor `init`).
2. `encapsulation_key` is exactly 1,216 bytes; otherwise `InvalidKeyLength` .

State changes. Creates the `KeyRecord` with `owner` , `epoch` , the key and the bump.

The instruction data carries the full 1,216-byte key with its 4-byte length prefix. The whole transaction fits in one v1 transaction, which is how both payee and auditor keys reach the chain.

5.4.3 open_account

Creates the signer's `ConfAccount` in a pool, bound to one of the signer's `KeyRecords`.

Account	Flags	Constraint
owner	w, s	Pays rent; becomes <code>ConfAccount.owner</code>
pool		PDA <code>["pool", pool.mint]</code>
key_record		PDA <code>["kem", owner, key_record.epoch]</code> ; <code>key_record.owner == owner</code> , otherwise <code>WrongOwner</code>
conf_account	w	Created; PDA <code>["acct", pool, owner]</code>
system_program		System program
rent_recovery		PDA <code>["rent-recovery"]</code>

Arguments. `self_record: [u8; 68]`, the encrypted opening of the initial balance;
`expected_close_nonce: u64`, the registry's `close_nonce` the self-record was sealed against.

Checks.

1. The KeyRecord belongs to the signer (`WrongOwner`).
2. The ConfAccount address is unused, so an owner has one account per pool at a time: never opened, or closed by `close_account` .
3. `expected_close_nonce` equals the registry's `close_nonce` (`StaleRentRecoveryNonce`).
4. `expected_close_nonce` is at most $2^{32} - 1$ (`SequenceRangeExhausted`).
5. The initial blinding derivation succeeds (`InitialBalanceDerivationFailed` only if its 32-bit retry counter is exhausted; `InvalidCommitment` if the scalar multiplication fails).

State changes. Writes `pool`, `owner`, `balance_commitment =  $r_0 \cdot H$` , the supplied `self_record`, `sequence = expected_close_nonce << 32`, `key_epoch = key_record.epoch`, `frozen = false` and the bump.

The initial commitment is derived by the program, not supplied by the caller; the derivation is given in the section on the initial balance commitment. The self-record is sealed for the initial sequence, which the SDK's `getInitialAccountSequence({ closeNonce })` returns as `closeNonce << 32`; if a close lands between reading the registry and opening, the nonce check fails and the client reseals against the new value. Reopening an account after `close_account` is the same instruction at the same PDA, with the current nonce and a KeyRecord the owner holds.

5.4.4 deposit

Moves a public amount of tokens from the owner into the pool: the amount less the ramp fee into the vault and onto the owner's commitment, the fee into the protocol fee account.

Account	Flags	Constraint
<code>owner</code>	s	Token authority for the source account
<code>pool</code>	w	PDA <code>["pool", pool.mint]</code>
<code>conf_account</code>	w	PDA <code>["acct", pool, owner]</code> ; <code>conf_account.pool = pool (WrongPool)</code> ; <code>conf_account.owner = owner (WrongOwner)</code>
<code>source_tokens</code>	w	Token owner is <code>owner (WrongOwner)</code> ; mint is <code>pool.mint (InvalidMint)</code>
<code>vault</code>	w	Address <code>pool.vault</code> ; PDA <code>["vault", pool]</code> ; token authority is the Pool <code>(InvalidAuthority)</code> ; mint is <code>pool.mint (InvalidMint)</code>
<code>protocol_fee_account</code>	w	Address: the associated token account of <code>protocol_authority</code> for <code>pool.mint</code> ; mint is <code>pool.mint (InvalidMint)</code> ; token owner is <code>protocol_authority (InvalidAuthority)</code>
<code>token_program</code>		SPL Token

Arguments. `amount: u64` (the gross amount a); `expected_sequence: u64`; `self_record: [u8; 68]`.

Checks.

1. The account is not frozen (`AccountFrozen`), its sequence is below the last value of its 32-bit range (`SequenceRangeExhausted`), and `expected_sequence` equals its sequence (`StaleSequence`).
2. $0 < a \leq \text{deposit_cap}$; otherwise `DepositCapExceeded`.
3. With $f = \lfloor a \cdot \text{ramp_fee_bps} / 10\,000 \rfloor$, `principal + (a - f)` does not overflow (`ArithmeticOverflow`) and does not exceed `pool_cap` (`PoolCapExceeded`).
4. The vault's token amount $+(a - f)$ does not overflow (`ArithmeticOverflow`).
5. $C_{bal} + (a - f) \cdot G$ is computed with the multiply and add syscalls.
6. The SPL transfers of $a - f$ from `source_tokens` to the vault and of f to the protocol fee account (skipped when $f = 0$), authorized by the owner, succeed.

State changes. `principal += a - f`; `balance_commitment =  $C_{bal} + (a - f) \cdot G$` ; `sequence += 1`; `self_record` replaced.

The pool cap is checked against `principal`, not against the vault's token balance. Tokens sent to the vault by an ordinary SPL transfer are public surplus: they create no confidential credit, cannot be withdrawn without a confidential balance to back them, and do not consume the cap, so they cannot block valid deposits. A deposit keeps the balance blinding unchanged: the new opening is $(b + a - f, \rho)$.

5.4.5 transfer

Debits the sender by a hidden amount and creates a Note for the payee. The transaction must contain the native `VerifyBatchedRangeProofU128` instruction immediately before this one.

Account	Flags	Constraint
<code>owner</code>	w, s	Sender wallet; pays the Note rent
<code>pool</code>		PDA <code>["pool", pool.mint]</code>
<code>conf_account</code>	w	Sender's account; PDA <code>["acct", pool, owner]</code> ; <code>has_one</code> pool and owner
<code>recipient_account</code>		Payee's account; PDA <code>["acct", pool, recipient_account.owner]</code> ; <code>has_one</code> pool
<code>recipient_key_record</code>		PDA <code>["kem", recipient_account.owner, epoch]</code> ; owner is the payee (<code>InvalidRecipient</code>); epoch equals <code>recipient_account.key_epoch</code> (<code>InvalidKeyEpoch</code>)
<code>auditor_key_record</code>		PDA <code>["kem", pool.auditor, pool.auditor_kem_epoch]</code> ; owner and epoch match the Pool (<code>InvalidAuthority</code>)
<code>note</code>	w	Created; PDA <code>["note", pool, owner, expected_sequence]</code>
<code>instructions</code>		The instructions sysvar (address checked)
<code>system_program</code>		System program

Arguments. `args`: `TransferArgs` :

Field	Type	Encoded bytes	Meaning
<code>expected_sequence</code>	<code>u64</code>	8	Sender's current sequence; also the Note seed
<code>commitment</code>	<code>[u8; 32]</code>	32	C_{note}
<code>payee_envelope</code>	<code>Vec<u8></code>	4 + 1,208	X-Wing envelope to the payee
<code>auditor_envelope</code>	<code>Vec<u8></code>	4 + 1,208	X-Wing envelope to the auditor
<code>self_record</code>	<code>[u8; 68]</code>	68	Sender's new encrypted balance opening

With its 8-byte discriminator the instruction data is 2,540 bytes; the preceding proof instruction adds 1,001.

Checks.

1. The sender's account is not frozen (`AccountFrozen`), its sequence is below the last value of its 32-bit range (`SequenceRangeExhausted`), and `expected_sequence` matches (`StaleSequence`).
2. The recipient's account is not frozen (`AccountFrozen`).
3. Both envelopes are exactly 1,208 bytes (`InvalidEnvelopeLength`).
4. `commitment` decodes to a valid Ristretto point (`InvalidCommitment`).
5. The program computes $C'_{bal} = C_{bal} - C_{note}$ from the stored balance.
6. The previous instruction is a native U128 range proof whose context is exactly $[C_{note}, C'_{bal}]$ with bit lengths $[64, 64]$ (see the proof-binding section).
7. The Note address is unused (Anchor `init`).

State changes.

Account	Field	New value
Note	<code>pool</code> , <code>sender</code>	the pool, the signer
Note	<code>recipient</code> , <code>original_recipient</code>	<code>recipient_account.owner</code>
Note	<code>commitment</code>	C_{note}
Note	<code>envelope_hash</code>	$\text{SHA256}(E_{payee} \mid E_{auditor})$
Note	<code>sequence</code>	<code>expected_sequence</code>
Note	<code>created_slot</code>	current slot from the Clock sysvar
Note	<code>recipient_key_epoch</code>	<code>recipient_key_record.epoch</code>
Note	<code>auditor_kem_epoch</code>	<code>pool.auditor_kem_epoch</code>
Note	<code>attested</code> , <code>returned</code>	false, false
Sender ConfAccount	<code>balance_commitment</code>	$C_{bal} - C_{note}$
Sender ConfAccount	<code>sequence</code> , <code>self_record</code>	incremented, replaced

`principal` is unchanged: value moves from a balance into an outstanding note.

The program computes the second proof commitment itself, from the stored balance, so a caller cannot prove a range statement about any other point. The two KeyRecords are required accounts, so a transfer names only a payee whose key for its account's epoch exists, and the Note records both epochs. The

program checks the envelopes' lengths but not their contents: no one is credited with a commitment they cannot open, because the payee checks its opening before `accept` and, when the pool requires it, the auditor checks its own envelope first. The client samples each note's blinding r from a CSPRNG over the canonical scalar field, independently for every note.

5.4.6 **attest**

Marks a batch of Notes as attested by the pool's auditor.

Account	Flags	Constraint
<code>pool</code>		<code>PDA ["pool", pool.mint] ; pool.auditor = auditor (InvalidAuthority)</code>
<code>auditor</code>	s	The pool's auditor key
remaining accounts	w	One or more Notes

Arguments. None.

Checks.

1. At least one Note is supplied (`EmptyBatch`).
2. No Note address appears twice (`DuplicateNote`).
3. Each Note account is writable (`AccountNotWritable`).
4. Each deserializes as a Note owned by the program.
5. Each belongs to this pool (`WrongPool`) and sits at its canonical address (`InvalidNote`), checked as described under note validation below.
6. No Note has been returned (`NoteAlreadyReturned`).

State changes. `attested = true` on every Note in the batch. The batch is atomic with its transaction.

The attestation itself is the auditor's signature on this transaction. The work behind it happens off chain: the auditor opens its envelope with the key of the Note's `auditor_kem_epoch`, checks the opening against C_{note} , and attests only Notes that pass.

5.4.7 **accept**

Adds a batch of Notes to the signer's balance and closes them.

Account	Flags	Constraint
owner	w, s	Recipient wallet
pool		PDA ["pool", pool.mint]
conf_account	w	PDA ["acct", pool, owner] ; has_one pool and owner
remaining accounts	w	Pairs: (Note, the Note's sender wallet)

Arguments. expected_sequence: u64 ; self_record: [u8; 68] .

Checks.

1. The account is not frozen (AccountFrozen), its sequence is below the last value of its 32-bit range (SequenceRangeExhausted), and expected_sequence matches (StaleSequence).
2. The remaining accounts are a non-empty, even-length list (InvalidBatch).
3. No Note appears twice (DuplicateNote).
4. Both accounts of each pair are writable (AccountNotWritable).
5. Each Note deserializes, belongs to this pool and sits at its canonical address (WrongPool , InvalidNote).
6. Each Note's recipient is the signer (InvalidRecipient).
7. The second account of each pair is the Note's sender (InvalidRentDestination).
8. Each Note satisfies the attestation policy (AttestationRequired): $\neg \text{attest_required} \vee \text{attested} \vee (\text{returned} \wedge \text{recipient} = \text{sender})$.

State changes. $C'_{bal} = C_{bal} + \sum_i C_{note,i}$; every Note is closed and its lamports go to its sender; sequence += 1 ; self_record replaced. principal is unchanged.

The client accepts only notes it has decrypted and checked; the new balance opening is $(b + \sum v_i, \rho + \sum r_i)$. The owner account is writable because a sender accepting its own returned Note is also that Note's rent destination. The policy's third clause lets a sender recover a returned Note without the auditor: return_note clears attested , and the sender already knows the opening it created.

5.4.8 return_note

Hands a Note back to its sender without changing any balance.

Account	Flags	Constraint
authority	s	The payee, or the sender after the timeout
pool		PDA ["pool", pool.mint]
note	w	PDA ["note", pool, note.sender, note.sequence]; note.pool = pool (WrongPool)

Arguments. None.

Checks.

1. The Note has not been returned (`NoteAlreadyReturned`).
2. If the signer is not the Note's current `recipient`, it must be the Note's `sender` and the current slot must satisfy

$$slot \geq \text{sat}(\text{created_slot} + \text{return_timeout_slots})$$

where `sat` saturates at $2^{64} - 1$; otherwise `ReturnNotAuthorized`.

State changes. `recipient = sender`; `returned = true`; `attested = false`. No balance, sequence or principal changes.

The payee can return a Note at any slot, attested or not. The sender can take back a Note the payee has left unaccepted once the pool's timeout has passed. Saturating addition keeps the last representable slot reachable for any configured timeout. A Note is returned at most once, so a returned Note cannot be bounced back to the payee. `return_note` changes no balance, so it does not check the frozen flag: a frozen payee can still hand a note back.

5.4.9 withdraw

Subtracts a public amount from the owner's commitment and pays it out of the vault, less the ramp fee, which goes to the protocol fee account. The transaction must contain the native `VerifyBatchedRangeProofU64` instruction immediately before this one.

Account	Flags	Constraint
owner	s	Account owner
pool	w	PDA ["pool", pool.mint] ; signs the token transfer
conf_account	w	PDA ["acct", pool, owner] ; has_one pool and owner
vault	w	Address pool.vault ; PDA ["vault", pool] ; authority is the Pool; mint is pool.mint
protocol_fee_account	w	As in deposit
destination_tokens	w	Any token account of pool.mint (InvalidMint) other than the vault
instructions		The instructions sysvar (address checked)
token_program		SPL Token

Arguments. amount: u64 ; expected_sequence: u64 ; self_record: [u8; 68] .

Checks.

1. The account is not frozen (AccountFrozen), its sequence is below the last value of its 32-bit range (SequenceRangeExhausted), and expected_sequence matches (StaleSequence).
2. $a > 0$ (InvalidPublicAmount).
3. The destination is not the vault (InvalidDestination).
4. The program computes $C'_{bal} = C_{bal} - a \cdot G$.
5. The previous instruction is a native U64 range proof whose context is exactly $[C'_{bal}]$ with bit length [64].
6. principal $-a$ does not underflow (ArithmeticOverflow).
7. The SPL transfers from the vault of $a - f$ to the destination and of f to the protocol fee account (skipped when $f = 0$), signed by the Pool PDA, succeed.

State changes. principal -= a ; balance_commitment = $C_{bal} - a \cdot G$; sequence += 1 ; self_record replaced.

The range proof shows the remaining balance is non-negative, so no owner can withdraw more than its committed balance. A withdrawal keeps the blinding unchanged: the new opening is $(b - a, \rho)$.

5.4.10 close_account

Closes the signer's ConfAccount once its balance is zero, paying every lamport of the account to the owner.

Account	Flags	Constraint
<code>owner</code>	w, s	Account owner; receives the account's lamports
<code>pool</code>		PDA <code>["pool", pool.mint]</code>
<code>conf_account</code>	w	PDA <code>["acct", pool, owner]; conf_account.pool = pool (WrongPool); conf_account.owner = owner (WrongOwner);</code> closed to <code>owner</code>
<code>rent_recovery</code>	w	PDA <code>["rent-recovery"]</code>

Arguments.

Argument	Type	Meaning
<code>expected_sequence</code>	<code>u64</code>	The account's current sequence
<code>expected_close_nonce</code>	<code>u64</code>	The registry's current <code>close_nonce</code>
<code>nonce_point</code>	<code>[u8; 32]</code>	$R = k \cdot H$, the proof's nonce point
<code>response</code>	<code>[u8; 32]</code>	$s = k + c\rho \bmod \ell$, the proof's response

Checks.

1. The account is not frozen (`AccountFrozen`).
2. `expected_sequence` equals its sequence (`StaleSequence`).
3. `expected_close_nonce` equals the registry's `close_nonce` (`StaleRentRecoveryNonce`).
4. The zero-balance proof (R, s) verifies against the stored C_{bal} (`InvalidZeroBalanceProof`), as described under the zero-balance proof below.
5. `close_nonce + 1` does not overflow (`ArithmeticOverflow`).

State changes. `close_nonce = expected_close_nonce + 1`. Anchor's `close` moves every lamport of the `ConfAccount` to `owner` and returns the account to the System program. `principal`, the vault and every `Note` are unchanged.

The proof's challenge binds the program, the pool, the account, the owner, the current sequence, the close nonce, C_{bal} and R , so each proof serves one close only: every close in the deployment advances the nonce, after which a proof built against the old value fails with `StaleRentRecoveryNonce` and is rebuilt with a fresh k . The sequence-range check does not apply, so a zero balance closes at any sequence. Notes sent from or to the account stay open with their commitments and rent destinations; the owner can register its key again if it closed it, reopen the account at the same PDA with `open_account` and accept a returned or late `Note` into the new generation.

5.4.11 close_key_record

Closes the signer's KeyRecord, paying every lamport of the account to the owner.

Account	Flags	Constraint
owner	w, s	Key owner; receives the account's lamports
key_record	w	PDA ["kem", owner, key_record.epoch] ; key_record.owner == owner (WrongOwner); closed to owner
rent_recovery		PDA ["rent-recovery"]

Arguments. None.

Checks.

1. The KeyRecord's address is not in the registry's auditor_keys (AuditorKeyInUse).

State changes. Anchor's close moves every lamport of the KeyRecord to owner and returns the account to the System program.

A closed KeyRecord leaves the owner's ConfAccounts, balances and Notes as they are. transfer requires the payee KeyRecord at the account's key_epoch , so a payee registers that epoch again, at the same address, before it receives new transfers, and the X-Wing seed it keeps still opens its earlier envelopes. The key a pool names for its auditor stays on chain for every transfer of that pool, because init_pool and initialize_rent_recovery list it in the registry.

5.4.12 freeze and thaw

Set or clear the frozen flag on one ConfAccount.

Account	Flags	Constraint
freeze_authority	s	Must equal pool.freeze_authority (InvalidAuthority)
pool		PDA ["pool", pool.mint]
conf_account	w	PDA ["acct", pool, conf_account.owner] ; has_one pool

Arguments. None. **State changes.** frozen = true (freeze) or frozen = false (thaw); the sequence is unchanged.

Operation	On a frozen account
<code>deposit</code> , <code>transfer</code> (as sender), <code>accept</code> , <code>withdraw</code>	Rejected with <code>AccountFrozen</code>
<code>transfer</code> naming it as recipient	Rejected with <code>AccountFrozen</code>
<code>close_account</code>	Rejected with <code>AccountFrozen</code>
<code>attest</code> of Notes it sent or receives	Allowed
<code>return_note</code> by it or to it	Allowed; no balance changes
Accepting a Note returned to it	Waits until <code>thaw</code>

Freezing stops value from entering or leaving the account and leaves every Note in a state from which it can still be attested, returned and later accepted.

5.4.13 `initialize_rent_recovery`

Creates the `RentRecoveryRegistry` and protects the auditor key of every pool that already exists.

Account	Flags	Constraint
<code>payer</code>	w, s	Funds the registry rent and its growth
<code>authority</code>	s	Must equal the upgrade authority recorded in <code>program_data</code>
<code>program</code>		The Hindsight program; its <code>ProgramData</code> address must equal <code>program_data</code>
<code>program_data</code>		The program's upgradeable-loader <code>ProgramData</code> account
<code>rent_recovery</code>	w	Created; PDA <code>["rent-recovery"]</code>
<code>system_program</code>		System program
remaining accounts		Every existing Pool

Arguments. None.

Checks.

1. The program account's `ProgramData` address equals `program_data` , and that `ProgramData`'s upgrade authority equals `authority` ; otherwise `InvalidBootstrapAuthority` .
2. The registry address is unused, so the registry is created once (Anchor `init`).

3. Each remaining account is owned by the program (`InvalidRentRecoveryRegistry`).
4. Each deserializes as a Pool, discriminator included (`InvalidRentRecoveryRegistry`).
5. Each sits at its canonical address, `create_program_address(["pool", pool.mint, pool.bump])` (`InvalidRentRecoveryRegistry`).

State changes. Creates the registry with `close_nonce = 0`, no keys and the bump, then adds the auditor KeyRecord address `["kem", pool.auditor, pool.auditor_kem_epoch]` of each supplied Pool, once per distinct address, growing the registry by 32 bytes per key with the payer topping up its rent.

The signer is the program's current upgrade authority, checked through the same ProgramData relationship as `init_pool`, and its signature certifies that the remaining accounts are every Pool of the deployment. The operator commands `program:init` and `local:setup` scan the program's Pool accounts, supply all of them, and run this instruction before they create a pool. `init_pool`, `open_account`, `close_account` and `close_key_record` all name the registry, so they run once it exists.

5.4.14 **configure_buyback**

Creates the BuybackConfig and fixes the bought token, the keeper and the minimum-output ratio.

Account	Flags	Constraint
<code>payer</code>	w, s	Funds the BuybackConfig rent
<code>authority</code>	s	Must equal the upgrade authority recorded in <code>program_data</code>
<code>program</code>		The Hindsight program; its ProgramData address must equal <code>program_data</code>
<code>program_data</code>		The program's upgradeable-loader ProgramData account
<code>buyback_config</code>	w	Created; PDA <code>["buyback"]</code>
<code>token_mint</code>		The \$HIND mint, an SPL Token or Token-2022 mint
<code>output_token_program</code>		SPL Token or Token-2022
<code>system_program</code>		System program

Arguments. `args: ConfigureBuybackArgs :`

Field	Type	Meaning
<code>keeper</code>	Pubkey	The only signer of <code>claim_buyback</code> and <code>complete_buyback</code>
<code>minimum_output_numerator</code>	u64	n , in \$HIND base units
<code>minimum_output_denominator</code>	u64	d , in fee-mint base units

Checks.

1. The program account's ProgramData address equals `program_data`, and that ProgramData's upgrade authority equals `authority`; otherwise `InvalidBootstrapAuthority`.
2. The BuybackConfig address is unused, so the configuration is created once (Anchor `init`).
3. `keeper` is not the all-zero key, and $n > 0$ and $d > 0$; otherwise `InvalidBuybackConfiguration`.
4. `token_mint` is owned by `output_token_program` (`InvalidBuybackMint`). Under Token-2022 the mint unpacks with its extensions (`InvalidBuybackMint`) and carries no extension other than `MetadataPointer` and `TokenMetadata` (`UnsupportedBuybackMintExtension`); any other program must be SPL Token (`InvalidBuybackMint`).

State changes. Writes `token_mint`, `token_program = output_token_program`, `keeper`, both ratio fields, `total_burned = 0`, `sequence = 0`, every pending field zero and the bump.

The signer is the program's current upgrade authority, checked through the same ProgramData relationship as `init_pool`. No instruction edits or closes the BuybackConfig afterwards. Until it exists, `claim_buyback` and `complete_buyback` fail with `BuybackNotConfigured`, while `deposit` and `withdraw` go on paying fees into the protocol fee account.

5.4.15 `claim_buyback`

Moves an exact amount of fees from the protocol fee account to the keeper, at the start of a buyback transaction.

Account	Flags	Constraint
keeper	s	Equals <code>buyback_config.keeper</code> (<code>InvalidAuthority</code>)
buyback_config	w	PDA <code>["buyback"]</code> ; owned by the program and initialized (<code>BuybackNotConfigured</code>)
fee_mint		A classic SPL Token mint with 6 decimals (<code>InvalidMint</code>): the mint whose fees are claimed
protocol_fee_account	w	Address: the associated token account of <code>protocol_authority</code> for <code>fee_mint</code> under SPL Token (<code>InvalidProtocolFeeAccount</code>); token owner is <code>protocol_authority</code> (<code>InvalidAuthority</code>); mint is <code>fee_mint</code> (<code>InvalidMint</code>)
keeper_fee_account	w	Address: the keeper's associated token account for <code>fee_mint</code> under SPL Token (<code>InvalidBuybackAccount</code>); token owner is the keeper (<code>InvalidAuthority</code>); mint is <code>fee_mint</code> (<code>InvalidMint</code>)
token_mint	w	Equals <code>buyback_config.token_mint</code> and is owned by <code>buyback_config.token_program</code> (<code>InvalidBuybackMint</code>); under Token-2022, only <code>MetadataPointer</code> and <code>TokenMetadata</code> extensions (<code>UnsupportedBuybackMintExtension</code>)
protocol_token_account	w	Address: the associated token account of <code>protocol_authority</code> for <code>token_mint</code> under <code>buyback_config.token_program</code> (<code>InvalidBuybackAccount</code>); owned by that token program (<code>InvalidBuybackMint</code>); token owner is <code>protocol_authority</code> (<code>InvalidAuthority</code>); mint is <code>token_mint</code> (<code>InvalidBuybackMint</code>)
protocol_authority		PDA <code>["protocol"]</code> ; signs the token transfer
token_program		SPL Token
output_token_program		Equals <code>buyback_config.token_program</code> (<code>InvalidBuybackMint</code>)
instructions		The instructions sysvar (address checked)

Arguments. `amount`: `u64` (the claim a , in fee base units); `minimum_output`: `u64` (the least amount m , in \$HIND base units, that the swap must deliver); `expected_sequence`: `u64` .

Checks.

1. The BuybackConfig is initialized (`BuybackNotConfigured`).
2. Every account meets the constraints in the table above.
3. The transaction has the shape described under buyback binding below (`InvalidBuybackTransaction`).
4. No claim is pending: `pending_amount` is zero (`BuybackAlreadyPending`).
5. `expected_sequence` equals `sequence` (`BuybackSequenceMismatch`).
6. $0 < a \leq$ the protocol fee account's balance (`InvalidBuybackAmount`).
7. $m \geq \lceil a \cdot n/d \rceil$, computed in 128-bit arithmetic, and $m > 0$ (`BuybackOutputTooSmall`).
8. The SPL transfer of a from the protocol fee account to `keeper_fee_account` , signed by `protocol_authority` , succeeds.

State changes.

Account	Field	New value
BuybackConfig	<code>pending_fee_mint</code>	<code>fee_mint</code>
BuybackConfig	<code>pending_amount</code>	a
BuybackConfig	<code>pending_minimum_output</code>	m
BuybackConfig	<code>pending_input_before</code>	the keeper fee account's balance before the transfer
BuybackConfig	<code>pending_output_before</code>	the protocol token account's balance before the swap
Protocol fee account	token amount	$-a$
Keeper fee account	token amount	$+a$

The claim moves exactly a and leaves the rest of the fee account in place. The floor $\lceil a \cdot n/d \rceil$ comes from the BuybackConfig, so it is fixed at configuration; the keeper may name a higher minimum, such as its quote's minimum output, and never a lower one. Only a successful completion advances `sequence` , so each claim names the one sequence value that is current.

5.4.16 complete_buyback

Checks the swap's result and burns all the \$HIND it delivered. It is the final instruction of a buyback transaction and takes the same eleven accounts, in the same order, as the `claim_buyback` before it.

Offset	Length	Field
0	1	Discriminator: 6 for <code>VerifyBatchedRangeProofU64</code> , 7 for <code>VerifyBatchedRangeProofU128</code>
1	256	Eight 32-byte compressed commitments, in order
257	8	Eight one-byte bit lengths, in the same order
265	672 (U64) or 736 (U128)	Proof bytes

The U64 instruction is 937 bytes and the U128 instruction is 1,001. The proof sizes follow $(2 \log_2 n + 9) \times 32$ bytes for n aggregate range bits: 21 group elements for 64 bits, 23 for 128. Native verification costs 111,000 CU for U64 and 200,000 CU for U128. Unused context slots hold the all-zero encoding, which the native format uses as its end-of-list sentinel.

The routine performs these checks in order:

Step	Check	Error
1	The proof type matches the number of expected commitments: U128 with two, U64 with one	InvalidProofType
2	No expected commitment is the identity encoding	IdentityProofContextUnsupported
3	The current top-level instruction index is greater than zero	MissingRangeProof
4	The top-level instruction at the current index is a Hindsight instruction, so the call is direct and not a CPI from another program	ProofCpiUnsupported
5	The instruction at index - 1 targets the ZK ElGamal Proof program	InvalidProofProgram
6	That instruction lists no accounts: the proof is inline, not read from a proof account or context-state account	ProofContextStateUnsupported
7	Its discriminator is the expected proof type	InvalidProofType
8	Its payload decodes as the exact proof-data struct for that type	InvalidProofData
9	Context slot i equals expected commitment i with bit length 64 for every expected slot, and every other slot is the zero encoding with bit length 0	ProofContextMismatch

The expected contexts are:

Instruction	Native proof	Commitments	Bit lengths
transfer	VerifyBatchedRangeProofU128	$[C_{note}, C_{bal} - C_{note}, 0, 0, 0, 0, 0, 0]$	$[64, 64, 0, 0, 0, 0, 0, 0]$
withdraw	VerifyBatchedRangeProofU64	$[C_{bal} - a \cdot G, 0, 0, 0, 0, 0, 0, 0]$	$[64, 0, 0, 0, 0, 0, 0, 0]$

For a transfer, the verified statement is that the sender knows openings with

$$C_{note} = v \cdot G + r \cdot H, \quad C_{bal} - C_{note} = (b - v) \cdot G + (\rho - r) \cdot H, \\ v, b - v \in [0, 2^{64})$$

so the payment is a non-negative 64-bit amount and the sender's remaining balance is non-negative. For a withdrawal it is $b - a \in [0, 2^{64})$ for $C_{bal} - a \cdot G$.

The guarantee comes from atomicity: a Solana transaction executes its instructions in order and fails as a whole if any one fails. If `transfer` succeeds, the instruction before it verified a proof whose context names C_{note} and the balance the program computed from the account at the sequence the transaction names. Amounts are never decrypted by solving a discrete log, so they are full 64-bit values with no split into limbs.

Step 4 makes `transfer` and `withdraw` top-level instructions: a wrapper program calling them through CPI fails, because its own instruction occupies the current top-level index. Step 6 keeps the proof bytes inline in the transaction itself.

5.6 Buyback binding

`claim_buyback` and `complete_buyback` call one routine, `validate_buyback_transaction`, before they move or burn anything. It reads every top-level instruction of the transaction through the instructions sysvar and requires one shape:

Position	Instruction
before the claim	any instructions, such as compute budget, associated-token-account creation and the route's setup
	<code>claim_buyback</code>
between	one or more instructions: the swap
last	<code>complete_buyback</code>

The routine performs these checks:

Step	Check	Error
1	The top-level instruction at the current index is a Hindsight instruction carrying this instruction's discriminator, so the call is direct and not a CPI from another program	InvalidBuybackTransaction
2	Every top-level Hindsight <code>claim_buyback</code> or <code>complete_buyback</code> in the transaction lists exactly the eleven accounts of this call, in the same order	InvalidBuybackTransaction
3	The transaction holds exactly one <code>claim_buyback</code> and exactly one <code>complete_buyback</code>	InvalidBuybackTransaction
4	<code>complete_buyback</code> is the last instruction of the transaction	InvalidBuybackTransaction
5	The claim comes before the completion with at least one instruction between them	InvalidBuybackTransaction

Both endpoints run the same routine, so each one finds the other in place. The guarantee again comes from atomicity: a claim takes effect only in a transaction whose last instruction is a completion over the same accounts, and that completion succeeds only if the swap spent the whole claim and delivered at least `minimum_output` to the protocol token account. If any check, the swap or the burn fails, the transaction fails as a whole and the claimed fees never leave the protocol fee account. The swap instructions themselves are not inspected: any route that meets the completion's checks qualifies, Jupiter's on mainnet-beta and the co-signing stand-in market on localnet and devnet alike.

5.7 Zero-balance proof

`close_account` calls one routine, `verify_zero_balance`, before it changes anything. It checks a noninteractive Schnorr proof that the owner knows a scalar ρ with

$$C_{bal} = \rho \cdot H$$

which is to say that the balance commitment has no G component and opens to the amount zero. The client, holding the opening $(0, \rho)$, draws a fresh nonzero scalar k , sends $R = k \cdot H$ as `nonce_point` and $s = k + c\rho \bmod \ell$ as `response`, where the challenge is

$$c = \text{SHA256}(\text{"hindsight:close-account:v1"} \parallel \text{program} \parallel \text{pool} \\ \parallel \text{account} \parallel \text{owner} \parallel \text{le}_{64}(\text{sequence}) \\ \parallel \text{le}_{64}(\text{close_nonce}) \parallel C_{bal} \parallel R) \bmod 2^{252}$$

The domain string is 26 ASCII bytes, the four addresses are 32 raw bytes each, and the preimage is 234 bytes. As for the initial blinding, the reduction clears the high four bits of the last digest byte. The program exposes the challenge as `close_account_challenge`, and the SDK's `generateZeroBalanceProof` builds the proof from the account's current opening.

The routine performs these checks in order:

Step	Check	Error
1	<code>response</code> is a canonical scalar, below the group order ℓ	<code>InvalidZeroBalanceProof</code>
2	<code>nonce_point</code> is not the identity encoding	<code>InvalidZeroBalanceProof</code>
3	<code>nonce_point</code> is a valid Ristretto point	<code>InvalidZeroBalanceProof</code>
4	The stored C_{bal} is a valid Ristretto point	<code>InvalidZeroBalanceProof</code>
5	The program computes c from its own ID, the pool, the ConfAccount and owner addresses, the stored sequence, <code>expected_close_nonce</code> , the stored C_{bal} and R	
6	$s \cdot H = R + c \cdot C_{bal}$, computed with two multiply syscalls and one add	<code>InvalidZeroBalanceProof</code>

Every input to the challenge except R comes from the program and its accounts, so a proof made for one account, owner, pool, sequence or nonce fails on every other. The construction, and why the proof reveals nothing about ρ , is given in the Cryptographic construction chapter.

5.8 Initial balance commitment

A new account's balance is known to be zero, but its commitment cannot be the identity point: the native proof format reads an all-zero commitment as an unused slot. An owner who deposited and then withdrew everything from an identity-initialized account would be left with $0 \cdot G + 0 \cdot H$ and could not produce the U64 proof the withdrawal requires. `open_account` therefore starts every account at a nonidentity commitment to zero with a public, deterministic blinding:

$$\begin{aligned}
 r_0 &= \text{SHA256}(\text{"hindsight:initial-balance-blinding:v1"} \\
 &\quad \parallel \text{pool} \parallel \text{owner} \parallel \text{le}_{64}(e) \parallel \text{le}_{32}(k)) \bmod 2^{252}, \\
 C_0 &= r_0 \cdot H
 \end{aligned}$$

The domain string is 37 ASCII bytes, `pool` and `owner` are 32 raw bytes each, e is the account's key epoch, and k is a retry counter starting at zero. Reducing modulo 2^{252} is done by clearing the high four bits of the last digest byte; the result is below the Ristretto group order and therefore canonical. A zero

result is retried with the next counter value. The program exposes the derivation as `initial_balance_blinding` and `initial_balance_commitment`, and the SDK's `deriveInitialBalanceOpening` reproduces it byte for byte.

r_0 is public by construction: it opens a balance everyone knows is zero, and the deposits that follow are public too. After deposits only, the opening is $(\sum a, r_0)$ and a full withdrawal leaves $r_0 \cdot H$, which is a valid nonidentity commitment for the U64 proof. Confidential notes never use this scalar; each takes a fresh secret blinding. The first transfer or accept mixes a secret blinding into the balance, after which its opening is known only to its owner.

5.9 Note validation in batches

`attest` and `accept` receive their Notes as remaining accounts, so Anchor's per-account seed constraints do not apply to them. The program validates each one explicitly:

1. `Account::<Note>::try_from` checks that the account is owned by the program and carries the Note discriminator.
2. `validate_note` requires `note.pool` to equal the instruction's Pool (`WrongPool`).
3. It recomputes the address as `create_program_address(["note", pool, note.sender, le64(note.sequence), note.bump])` and requires it to equal the account's key (`InvalidNote`).

The stored bump was written by the program's own `init` with the canonical bump, so a single address computation replaces a bump search. Duplicate addresses are rejected before any mutation, because the same Note listed twice would otherwise be credited twice by `accept`.

5.10 Solvency accounting

Let $v(C)$ denote the amount a commitment opens to. The program maintains

$$\sum_{\text{ConfAccounts}} v(C_{bal}) + \sum_{\text{open Notes}} v(C_{note}) = \text{principal} \leq \text{vault.amount}$$

Instruction	Confidential balances	Open notes	principal	Vault
deposit	$+(a - f)$		$+(a - f)$	$+(a - f)$
transfer	$-v$	$+v$		
accept	$+\sum v_i$	$-\sum v_i$		
return_note		(re-targeted)		
withdraw	$-a$		$-a$	$-a$
close_account	one account of amount 0 leaves			

Each row preserves the equality. `close_account` removes a ConfAccount only on a proof that its commitment is $\rho \cdot H$, an amount of zero, and the Notes it sent or receives stay in the second sum. The range proofs carry the confidential rows: a transfer cannot create a negative note or leave a negative balance, and a withdrawal cannot take more than the balance holds. Public rows use checked arithmetic on `principal` and the SPL Token program's own balance checks on the vault. The inequality with the vault allows for tokens sent to the vault by ordinary transfers, which raise `vault.amount` without raising `principal`. Because `principal` is a `u64` bounded by `pool_cap`, every sum of honest openings also fits in 64 bits, so accepting a batch of notes cannot produce a balance beyond the range the next proof can express.

The buyback instructions touch none of these columns, and neither do `initialize_rent_recovery` and `close_key_record`. The fees the buyback spends went to the protocol fee account and were never part of the vault, of `principal` or of any commitment.

5.11 Error codes

Anchor numbers custom errors from 6000 in declaration order. The table lists every error, the instructions that raise it, and what it tells a client. Anchor's own constraint errors (for example a seeds mismatch or an account already in use) are reported with Anchor's standard codes.

Code	Name	Raised by	Meaning for the client
6000	ConfigurationRequired	init_pool	A policy field was not supplied
6001	InvalidPolicy	init_pool	Threshold or rotation interval is zero, deposit cap exceeds pool cap, or the timeout exceeds $2^{32} - 1$ slots
6002	FeeCustodyNotConfigured	init_pool	A nonzero flat transfer fee was configured
6003	InvalidMint	init_pool , deposit , withdraw , claim_buyback , complete_buyback	The mint does not have 6 decimals, a token account is for another mint, or the fee mint differs from the pending claim's
6004	InvalidBootstrapAuthority	init_pool , initialize_rent_recovery , configure_buyback	The admin or configuring signer is not the program's upgrade authority, or the ProgramData account is wrong
6005	InvalidKeyEpoch	transfer	The payee KeyRecord's epoch differs from the payee account's key_epoch
6006	InvalidAuthority	init_pool , deposit , withdraw , transfer , attest , freeze , thaw ,	Wrong auditor, freeze authority, keeper or auditor

Code	Name	Raised by	Meaning for the client
		claim_buyback , complete_buyback	KeyRecord, a zero authority key, a vault not owned by the Pool, or a buyback token account with the wrong token owner
6007	WrongOwner	open_account , deposit , transfer , accept , withdraw , close_account , close_key_record	The account, KeyRecord or source tokens do not belong to the signer
6008	WrongPool	most instructions	An account or Note belongs to another pool
6009	AccountFrozen	deposit , transfer , accept , withdraw , close_account	The signer's account, or the transfer's recipient account, is frozen
6010	StaleSequence	deposit , transfer , accept , withdraw , close_account	The account changed since the transaction was built; rebuild from fresh state
6011	InvalidPublicAmount	withdraw	The withdrawal amount is zero
6012	InvalidDestination	withdraw	The destination is the vault
6013	DepositCapExceeded	deposit	The amount is zero or above the pool's deposit cap

Code	Name	Raised by	Meaning for the client
6014	PoolCapExceeded	deposit	The deposit would take principal above the pool cap
6015	ArithmeticOverflow	deposit , withdraw , balance updates , init_pool , initialize_rent_recovery , close_account , claim_buyback , complete_buyback	A checked operation overflowed: on principal, vault amount or sequence, on the registry size or the close nonce, or in the buyback on the floor, total_burned or sequence
6016	InvalidKeyLength	register_key	The encapsulation key is not 1,216 bytes
6017	InvalidEnvelopeLength	transfer	An envelope is not 1,208 bytes
6018	InvalidCommitment	open_account , deposit , transfer , accept , withdraw	A commitment is not a valid Ristretto point, or a curve operation failed
6019	MissingRangeProof	transfer , withdraw	The instruction is first in its transaction, so no proof precedes it

Code	Name	Raised by	Meaning for the client
6020	InvalidProofProgram	transfer , withdraw	The preceding instruction is not the ZK ElGamal Proof program
6021	InvalidProofType	transfer , withdraw	The preceding proof is not U128 for a transfer or U64 for a withdrawal
6022	ProofCpiUnsupported	transfer , withdraw	The instruction was invoked through CPI; call it at top level
6023	DuplicateNote	attest , accept	A Note appears twice in the batch
6024	InvalidProofData	transfer , withdraw	The proof payload does not decode
6025	ProofContextMismatch	transfer , withdraw	The proof's commitments or bit lengths do not match this balance transition
6026	ProofContextStateUnsupported	transfer , withdraw	The proof instruction lists accounts; supply the proof inline
6027	IdentityProofContextUnsupported	transfer , withdraw	A commitment in the statement is the identity point
6028	InvalidNote	attest , accept	A Note is not at its canonical

Code	Name	Raised by	Meaning for the client
			address
6029	InvalidRecipient	transfer , accept	The payee KeyRecord is not the payee's, or the signer is not the Note's recipient
6030	InvalidRentDestination	accept	The account paired with a Note is not its sender
6031	AttestationRequired	accept	The pool requires attestation and the Note is not attested
6032	NoteAlreadyReturned	attest , return_note	The Note has already been returned
6033	ReturnNotAuthorized	return_note	The signer is neither the recipient nor the sender after the timeout
6034	EmptyBatch	attest	No Notes were supplied
6035	InvalidBatch	accept	The remaining accounts are empty or not in (Note, sender) pairs
6036	AccountNotWritable	attest , accept	A Note or rent destination was not passed as writable

Code	Name	Raised by	Meaning for the client
6037	InitialBalanceDerivationFailed	open_account	The initial blinding derivation exhausted its retry counter
6038	BuybackNotConfigured	claim_buyback , complete_buyback	Buyback token mint and keeper have not been configured
6039	InvalidBuybackConfiguration	configure_buyback	Buyback configuration requires a keeper and a positive minimum output ratio
6040	InvalidBuybackMint	configure_buyback , claim_buyback , complete_buyback	The configured buyback token mint or token program does not match
6041	UnsupportedBuybackMintExtension	configure_buyback , claim_buyback , complete_buyback	Only metadata pointer and token metadata extensions are supported for the buyback mint
6042	InvalidProtocolFeeAccount	claim_buyback , complete_buyback	The protocol fee account is not its canonical associated token account

Code	Name	Raised by	Meaning for the client
6043	InvalidBuybackAccount	claim_buyback , complete_buyback	A buyback token account is not the expected associated token account
6044	InvalidBuybackTransaction	claim_buyback , complete_buyback	Buyback requires one top-level claim, a swap step and the final matching completion
6045	BuybackAlreadyPending	claim_buyback	A buyback claim is already pending
6046	InvalidBuybackAmount	claim_buyback	The buyback claim amount is zero or exceeds available fees
6047	BuybackOutputTooSmall	claim_buyback , complete_buyback	The bought token amount is below the on-chain bound
6048	BuybackNotPending	complete_buyback	No buyback claim is pending
6049	BuybackInputNotSpent	complete_buyback	The swap did not spend the whole claimed input amount
6050	BuybackSequenceMismatch	claim_buyback	The buyback claim sequence has already been used or is not current
6051	InvalidZeroBalanceProof	close_account	The zero-balance proof is

Code	Name	Raised by	Meaning for the client
			invalid or does not match this account
6052	StaleRentRecoveryNonce	open_account , close_account	The rent recovery nonce has changed; refresh and rebuild the transaction
6053	AuditorKeyInUse	close_key_record	A current pool auditor key record cannot be closed
6054	SequenceRangeExhausted	open_account , deposit , transfer , accept , withdraw	The account sequence generation has reached its engineering bound
6055	InvalidRentRecoveryRegistry	initialize_rent_recovery	The rent recovery registry bootstrap contains an invalid pool

For users, the errors fall into three groups. `StaleSequence` and `StaleRentRecoveryNonce` are the normal signals to refresh state and rebuild; neither means value was lost. `AccountFrozen` , `AttestationRequired` and `ReturnNotAuthorized` report the pool's policy or administration in effect at that slot. The proof errors (6019 to 6027) and the length errors mean the transaction was not assembled the way the program binds it, and are fixed in the builder, not by retrying. The buyback errors (6038 to 6050) arise only in the keeper's buyback transaction; no payment instruction raises them. The rent recovery errors (6051 to 6055) arise in opening and closing accounts and keys, in the registry's creation, and at the end of a sequence range.

6 Transactions

A Hindsight transfer is one Solana transaction. It carries the native range-proof instruction and the `transfer` instruction that depends on it, the two X-Wing envelopes, the new commitment and the sender's new self-record. Everything the payment needs lands atomically or not at all. This chapter covers the transaction format that makes that possible, the byte and compute budget of a transfer as measured, the adjacency rule between the proof and the instruction it guards, the resource limits the builders set, and how a client builds, signs, submits and confirms a transaction, including after a lost response or an expired blockhash.

6.1 Why v1 transactions

A legacy Solana transaction is limited to 1,232 bytes. A transfer's two envelopes alone are 2,416 bytes, and the U128 range-proof instruction adds 1,001. Under the legacy limit a transfer would have to be split across several transactions, with the proof verified into a separate account and the envelopes staged in a buffer before the final instruction could run.

Version 1 transactions (SIMD-0296 and SIMD-0385), live on mainnet since epoch 1035 on 2026-09-15, raise the limit to 4,096 bytes. A complete transfer fits in one, so the proof, the debit, the Note and the envelope publication are a single atomic state change. The program is built for this path: it accepts its range proof only inline in the immediately preceding instruction.

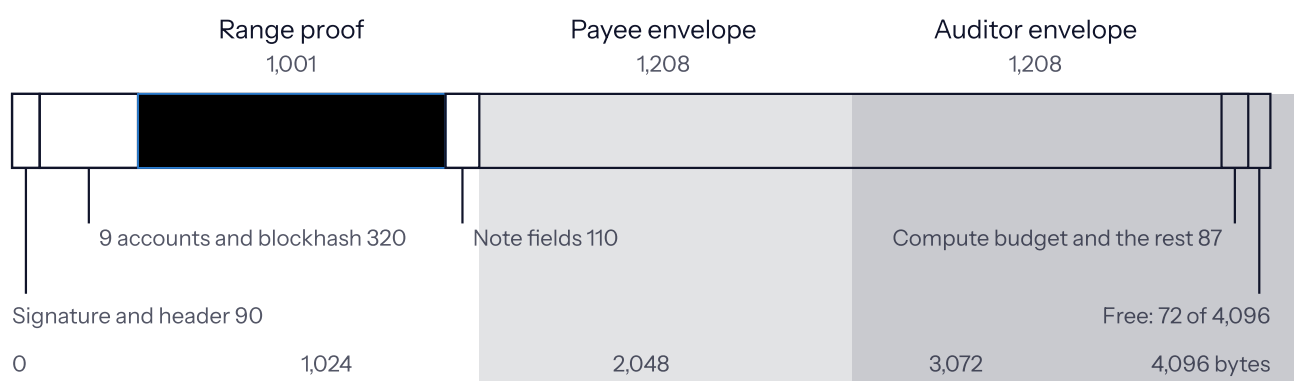
The v1 format differs from legacy and v0 in four ways that matter here:

Property	Legacy and v0	v1
Size limit	1,232 bytes	4,096 bytes
Layout	count-prefixed signatures, then message	message first, then a fixed number of 64-byte signatures with no count prefix
Compute limit, priority fee	Compute Budget instructions inside the instruction list	fields of the message configuration
Address lookup tables	allowed (v0)	not supported; every address is a static key

The format's maxima are 64 addresses, 64 instructions and 12 signatures per transaction. Every Hindsight instruction is built as a v1 transaction, including the ones that would fit a legacy packet, so clients and indexers handle a single format. The buyback transaction is the one exception: the crank builds `claim_buyback`, the swap and `complete_buyback` as a v0 transaction, compressed with the swap route's address lookup tables, and refuses one whose wire encoding exceeds 1,232 bytes. `register_key` benefits directly: its 1,216-byte encapsulation key makes a 1,485-byte transaction, which fits one v1 transaction and would not fit one legacy transaction.

6.2 The byte budget of one transfer

On a local Agave 4.3.0 validator with the program deployed and native proof verification enabled, one complete transfer measured **4,024 bytes** on the wire against the 4,096-byte limit, leaving 72 bytes. The figure divides it into the parts the site's facts module uses: the brief's itemized estimates, plus one part for what those estimates leave out, so the parts sum to the measurement.



One transfer as a v1 transaction: 4,024 of 4,096 bytes, measured on a local Agave 4.3.0 validator (one run). Bar length is to scale against the 4,096-byte limit. The two X-Wing envelopes take 2,416 bytes, the U128 range-proof instruction 1,001 (highlighted). The smaller parts are the brief's estimates; the part labelled "compute budget and the rest" is the difference between the estimates and the measurement.

Part	Bytes	Basis
Signature and header	90	estimate
9 accounts and blockhash	320	estimate
Range proof (736-byte proof, 264-byte context, 1-byte discriminator)	1,001	exact
Note fields (commitment, sequence, self-record)	110	estimate
Payee envelope (1,120-byte X-Wing ciphertext, 88-byte AEAD)	1,208	exact
Auditor envelope	1,208	exact
Compute budget and the rest	87	measurement less the estimates
Total	4,024	measured

6.2.1 Exact wire layout

The SDK compiles transactions with the maintained `@solana/kit` v1 message codec. Following that encoding field by field reproduces the measured length exactly. A v1 message is a fixed prefix, the static addresses, the configuration values, one 4-byte header per instruction, then each instruction's account indices and data; the signatures follow the message.

Offset	Length	Field
0	1	Version byte, 0x81 (v1)
1	3	Message header: 1 signer, 0 read-only signers, 8 read-only non-signers
4	4	Configuration mask: compute-unit limit and loaded-account-data limit present
8	32	Recent blockhash
40	1	Instruction count: 2
41	1	Static address count: 11
42	352	11 addresses of 32 bytes
394	4	Compute-unit limit, u32
398	4	Loaded-account-data limit, u32
402	4	Instruction 0 header: proof program index, 0 accounts, 1,001 data bytes
406	4	Instruction 1 header: Hindsight program index, 9 accounts, 2,540 data bytes
410	1	Proof discriminator, 7 (VerifyBatchedRangeProofU128)
411	256	Proof context: eight commitment slots
667	8	Proof context: eight bit lengths
675	736	Range proof
1411	9	transfer account indices
1420	8	transfer discriminator
1428	8	expected_sequence
1436	32	commitment (C_{note})
1468	4	Payee envelope length, 1,208
1472	1,120	Payee X-Wing ciphertext
2592	88	Payee AEAD ciphertext and tag
2680	4	Auditor envelope length, 1,208

Offset	Length	Field
2684	1,120	Auditor X-Wing ciphertext
3804	88	Auditor AEAD ciphertext and tag
3892	68	self_record
3960	64	Owner's Ed25519 signature
4024		End

Grouped the same way as the estimate, the exact encoding is:

Group	Exact	Estimate
Signature, version, header, mask, counts, configuration, instruction headers, account indices	99	90
11 addresses and the blockhash	384	320
Range-proof instruction data	1,001	1,001
transfer fields: discriminator, sequence, commitment, two length prefixes, self-record	124	110
Two envelopes	2,416	2,416
Not itemized		87
Total	4,024	4,024

The largest difference is the address list. The estimate counted nine account keys; the encoded message carries eleven, because the Hindsight program and the ZK ElGamal Proof program are themselves static addresses:

Role	Addresses
Writable signer, fee payer	sender wallet
Writable	sender ConfAccount, new Note
Read-only	Pool, payee ConfAccount, payee KeyRecord, auditor KeyRecord, instructions sysvar, System program, Hindsight program, ZK ElGamal Proof program

The two envelopes and the proof are fixed-size, and the remaining fields are fixed-width. Every transfer therefore serializes to the same length, whatever the amount, memo or recipient.

A withdrawal is smaller: ten static addresses (the protocol fee account among them), the 937-byte U64 proof instruction and 92 bytes of `withdraw` data, for **1,479 bytes** measured.

6.3 Compute units

The same run measured **226,570 CU** for the transfer, under the explicit 300,000 CU limit the builder set. The native U128 verification is a fixed 200,000 CU, so the Hindsight `transfer` instruction and the transaction's other work account for the remaining 26,570. That remainder covers the instructions-sysvar reads, Ristretto validation and subtraction (an addition or subtraction costs 521 CU), SHA-256 over the 2,416 envelope bytes, the Note account's creation and Anchor's account checks. The withdrawal measured **140,575 CU**, of which the native U64 verification is 111,000; its own instruction adds a scalar multiplication (2,208 CU), a subtraction and the two SPL token transfers, to the destination and to the protocol fee account.

That run executed these instructions:

Instruction	Wire bytes	Consumed CU
<code>init_pool</code>	591	34,593
<code>register_key</code>	1,485	7,043
<code>open_account</code>	432	18,708 to 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>accept</code> , the sender's own returned Note	367	9,347
<code>withdraw</code>	1,479	140,575
<code>freeze</code>	257	5,793
<code>thaw</code>	257	5,792
<code>close_account</code>	370	18,163
<code>close_key_record</code>	257	7,621

Each row is one transaction with a single signature. The ranges for `register_key` and `open_account` come from different wallets: creating a PDA searches for its canonical bump, and the number of attempts depends on the address. The returned-Note `accept` is 32 bytes smaller than an ordinary one because the rent destination is the signer itself, whose address is already in the message.

Batches grow linearly. The auditor service's attestation batches measured 290 bytes and 9,179 CU for two Notes and 356 bytes and 15,267 CU for four, which is 33 bytes per added Note: a 32-byte address and a 1-byte index. An `accept` adds 66 bytes per Note from a new sender: two addresses and two indices. Under the 64-address maximum, an `attest` can name up to 61 Notes beside its three fixed addresses, and an `accept` up to 30 Notes from distinct senders beside its four. Notes from a sender already in the batch add one address instead of two.

6.4 Proof adjacency

`transfer` requires the instruction directly before it, at the top level of the same transaction, to be the native `VerifyBatchedRangeProofU128` with context $[C_{note}, C_{bal} - C_{note}]$ and bit lengths $[64, 64]$;

`withdraw` requires `VerifyBatchedRangeProofU64` with $[C_{bal} - a \cdot G]$ and $[64]$. The program finds that instruction through the instructions sysvar at index $i - 1$, where i is its own index, and compares its context with commitments computed from the account's stored balance. Nothing may sit between them.

The v1 format keeps this simple. In legacy and v0 transactions, compute-budget settings are themselves instructions, and a builder must place them before the proof so they do not separate it from the instruction it guards. In v1 the compute-unit limit, the loaded-account-data limit and any priority fee are message configuration fields, not instructions, so a transfer message contains exactly two instructions: the proof at index 0 and `transfer` at index 1.

The SDK enforces the same rule before anything is signed. For a transfer it recomputes $C_{bal} - C_{note}$ from the account snapshot it was given, requires the proof's context to equal $[C_{note}, C_{bal} - C_{note}]$ with bit lengths $[64, 64]$ and zeroed unused slots, rejects identity commitments in active slots, and emits the proof instruction (discriminator, context, proof) immediately before the `transfer` instruction. A proof built against any other balance fails in the builder, not on chain.

Both instructions must also be invoked directly. A program that wraps `transfer` or `withdraw` in a CPI is rejected with `ProofCpiUnsupported`, so the proof-and-instruction pair always appears as two adjacent top-level instructions that anyone reading the transaction can check.

6.5 Compute budget and loaded account data

A v1 transaction that leaves its compute-unit limit unset is budgeted zero compute units, and one that leaves its loaded-account-data limit unset is budgeted zero bytes of account data. The legacy defaults do not apply. The SDK builder therefore takes both values explicitly and refuses to build without them:

Setting	v1 field	Builder bounds	Value the services use
Compute-unit limit	<code>computeUnitLimit</code> (<code>u32</code>)	1 to 1,400,000	300,000; the payroll executor accepts at most 300,000
Loaded-account-data limit	<code>loadedAccountsDataSizeLimit</code> (<code>u32</code>)	1 byte to 64 MiB	8 MiB
Priority fee	<code>priorityFeeLamports</code> (<code>u64</code>)	not set by the builder	none

The compute limit is sized for a transfer: 300,000 CU covers the measured 226,570 with room for the cost variation of PDA creation, and is the per-transfer target the protocol was designed against. The loaded-account-data limit counts every account the transaction loads, including the executable program and its `ProgramData`, not only the accounts the instruction writes. A budget sized for a

transaction's own small accounts fails before execution, because the deployed program's ProgramData alone is larger than 256 KiB. The payer-side executor and the auditor's attestation builder both set 8 MiB.

In v1 a priority fee is a total amount in lamports for the whole transaction, carried as its own configuration field, not a price per compute unit. The fee is paid by the fee payer in SOL like any Solana transaction fee.

6.6 Address lookup tables

Payment transactions use no address lookup tables; the v1 format does not support them. Every address is a static key in the message. For the transfer that is the 11 addresses listed above, 352 bytes of the 4,024. Static keys also mean every account the program touches is visible in the transaction itself, which the indexer and the auditor rely on when they decode a transfer. Only the v0 buyback transaction loads lookup tables, those of its swap route.

6.7 Building a transaction

A client builds every Hindsight transaction from public chain state. The SDK's

`createTransactionBuilder(context)` takes a `TransactionBuilderContext` : the program ID, the loaded-account-data limit, and snapshots of the Pool, the relevant ConfAccounts and Notes, decoded from raw account bytes.

1. **Decode.** The decoders accept an account only if its owner is the program, its size is the exact allocation (Pool 238, ConfAccount 190, Note 235, KeyRecord 1,265 bytes) and its first 8 bytes are the expected discriminator.
2. **Derive.** The builder re-derives every PDA (Pool and vault from the mint, the ConfAccount from pool and owner, the KeyRecords from owner and epoch, the Note from pool, sender and sequence) and rejects snapshots that do not match.
3. **Check.** It rejects a frozen account, an account from another pool, a sequence different from the snapshot, a duplicate Note in a batch, a Note that does not belong to the accepting owner, a returned Note in an attestation batch, a zero deposit or withdrawal, and a withdrawal to the vault.
4. **Encode.** It writes the instruction data in the program's Borsh layout and, for `transfer` and `withdraw`, prepends the proof instruction after checking its context.
5. **Compile.** It creates a version 1 message, sets the fee payer, sets the lifetime from a recent blockhash and its `lastValidBlockHeight`, sets the compute-unit and loaded-account-data limits, compiles, and encodes the wire bytes.
6. **Bound.** It rejects any result over 4,096 bytes.

The result, a `BuiltTransaction`, carries the message bytes, the complete wire bytes with zeroed signature placeholders, the serialized size, the required signers and the two resource limits. Building never submits anything. Openings, memos, proof witnesses and self-record keys stay in the client; only the finished proof and ciphertexts enter the transaction.

6.8 Signing

The owner's wallet signs the transaction bytes. For `transfer` and `withdraw` the owner is both the fee payer and the only signer. The wallet must support transaction version 1: through Wallet Standard that is the `solana:signTransaction` feature with version 1 listed in `supportedTransactionVersions`. The SDK passes raw transaction bytes to the wallet, so the v1 message never goes through a legacy transaction class.

`buildAndSign` checks what the wallet returns before accepting it:

Check	Purpose
The message bytes are unchanged	The wallet signed exactly what was built, including the proof and envelopes
The wire encoding decodes canonically with the official decoder	The bytes the client will submit are a well-formed v1 transaction
The only signer is the expected wallet	No extra signature is required or added
The Ed25519 signature verifies strictly	Noncanonical and small-order encodings are rejected

Supported wallet transports are browser-injected wallets, Wallet Standard wallets without USB transport, and mobile deep links; the SDK includes no USB hardware-wallet integration. Signing a transaction is separate from the sign-in message a wallet signs to open an API session. Neither signature is ever used as key material for X-Wing keys or self-records.

6.9 Submitting and confirming

The client sends the signed wire bytes to its configured Solana RPC: directly on a local validator, and through the API's restricted RPC relay (`POST /v1/rpc`) on a cluster whose endpoint carries a provider key. That relay forwards the signed bytes unchanged; the envelope relay carries only opaque envelope bytes. The returned signature identifies the transaction; it is not evidence that the transaction executed.

Confirmation is read from chain state at a stated commitment level:

- For a transfer, the expected Note exists at its PDA, derived from the pool, the sender and the sequence, with the expected sender, original recipient, sequence, commitment, envelope hash and key epochs.
- For any balance update, the ConfAccount's sequence has advanced and its commitment and self-record are the ones the client built.
- The transaction status of the signature is confirmed or finalized. Transaction reads request `maxSupportedTransactionVersion: 1`.

The hosted API serves finalized projections, which can trail the confirmed state. A client never builds a new transfer because a projection is late; it observes the expected Note or the signature instead.

6.10 Blockhash expiry and resending

A transaction is valid until the cluster's block height passes the `lastValidBlockHeight` returned with its blockhash. Within that window the client can resend the same signed bytes as often as it needs: identical bytes carry the identical signature, and the runtime processes a signature at most once.

The payer-side executor uses this as a fixed procedure:

1. Before the first submission, persist the signed wire bytes, the signature, the blockhash, its `lastValidBlockHeight` and the hash of the message, durably and encrypted.
2. Submit. A lost or failed RPC response is treated as unknown, never as failure.
3. Observe: look for the expected Note at its PDA and query the signature status, polling at a fixed interval (250 ms by default, 40 polls per attempt).
4. If nothing has landed and the current block height is still at or below `lastValidBlockHeight`, resend the same persisted bytes.
5. If the block height has passed `lastValidBlockHeight` and the transaction has not landed, build a new attempt with a fresh blockhash and a new signature (three signed attempts per row by default). The attempt keeps the same Note PDA, sequence, commitment, note opening, envelopes and self-record; only the range proof is regenerated, over the same context, before signing.
6. If the source account's sequence, commitment, self-record, key epoch or frozen state changed in a way the plan does not explain, pause instead of rebuilding.

The program makes a duplicate payment impossible by construction. A transfer creates its Note at `["note", pool, sender, expected_sequence]` and requires the sender's account to be at exactly that sequence. Once one attempt lands, the account's sequence has advanced and the Note address is taken, so every other attempt for the same step fails with `StaleSequence` or at the Note's creation, whichever is checked first. A retried transfer therefore lands at most once.

7 Flows

This chapter follows value through the system step by step: onboarding, deposit, transfer, attestation, acceptance, return and recovery, withdrawal, closing an account, and a payroll batch. Each flow is a numbered sequence that names who signs each step and what changes on chain. The cryptographic work (commitments, envelopes, range proofs, self-records) happens in the client or in the auditor's own domain. The program sees commitments, ciphertexts and proofs. The hosted services see public chain data and opaque envelope bytes.

7.1 Actors and what they sign

Actor	On-chain signatures	Keys held off chain
Payer (employer, buyer, treasury)	<code>register_key</code> , <code>open_account</code> , <code>deposit</code> , <code>transfer</code> , <code>withdraw</code> , <code>close_account</code> , <code>close_key_record</code> ; <code>return_note</code> after the timeout; <code>accept</code> of its own returned notes	Ed25519 wallet; X- Wing keypair; AES- 256 self-record key
Payee (employee, supplier)	<code>register_key</code> , <code>open_account</code> , <code>accept</code> , <code>return_note</code> , <code>withdraw</code> , <code>close_account</code> , <code>close_key_record</code>	The same three
Auditor	<code>attest</code> with the pool's auditor key; <code>register_key</code> for each X-Wing epoch	Ed25519 attestation key; X- Wing epoch keys in its custody
Pool admin	<code>init_pool</code> and the one-time <code>initialize_rent_recovery</code> and <code>configure_buyback</code> with the upgrade authority; <code>freeze</code> and <code>thaw</code> with the freeze authority	Multisig-held signers
Buyback keeper	<code>claim_buyback</code> and <code>complete_buyback</code> , in one transaction with the swap	Ed25519 keeper key, held by the buyback crank
Relay and indexer	nothing	none

Every on-chain step is a single-signer transaction unless noted. The wallet that signs a balance-changing instruction is the account owner and pays the fee. A wallet also signs one off-chain message, the sign-in challenge that opens an API session. That signature authorizes a session only; it is never used to derive an encryption key.

7.2 Pool setup

Before anyone onboards, the pool exists and its auditor key is on chain.

1. **Pool admin signs** `initialize_rent_recovery` once per deployment, with the payer and the program's upgrade authority, listing every existing Pool. The chain gains the `RentRecoveryRegistry` at `["rent-recovery"]` with close nonce 0 and the auditor key of each listed pool.
2. **Pool admin signs** `init_pool` with the payer and the program's upgrade authority. The chain gains the Pool at `["pool", mint]` with the auditor identity, its KEM epoch, the freeze authority and the full

policy, and an empty vault at `["vault", pool]` whose token authority is the Pool. The registry gains the address of the auditor's KeyRecord.

3. **Auditor signs `register_key`** for the epoch named in the Pool. The chain gains the auditor's KeyRecord at `["kem", auditor, epoch]`. Every transfer references it, so it must exist before the first transfer.

7.3 Onboarding

Payer and payee onboard the same way. Keys belong to the wallet and accounts to the pool, so a wallet registers a key once and can open accounts in several pools with it.

1. **Client generates keys.** It creates an X-Wing keypair from fresh entropy, whose private part is a 32-byte seed, and a non-extractable AES-256-GCM key for self-records. Both stay in the client's durable keystore. Neither comes from a wallet signature.
2. **Wallet signs a sign-in challenge.** The client requests a one-use challenge bound to the wallet, the page origin, the service, the cluster, the program and an expiry, signs it with the wallet's message-signing feature and receives an HttpOnly session cookie and a CSRF token. The session authorizes wallet-scoped reads and relay uploads. Nothing changes on chain.
3. **Owner signs `register_key(epoch, key)`**. The chain gains a KeyRecord at `["kem", owner, epoch]` holding the 1,216-byte encapsulation key. The owner pays its rent.
4. **Client prepares the initial self-record.** It reads the registry's close nonce n , derives the public initial blinding r_0 from the pool, the owner and the key epoch, computes $C_0 = r_0 \cdot H$, and seals the opening $(0, r_0)$ into a 68-byte self-record bound to the pool, the account address, the initial sequence $s_0 = n \cdot 2^{32}$ and C_0 . Until the first close in the deployment, $n = 0$ and $s_0 = 0$.
5. **Owner signs `open_account(self_record, n)`**. The chain gains a ConfAccount at `["acct", pool, owner]` with $C_{bal} = C_0$, sequence s_0 , the self-record, `key_epoch` set to the registered epoch, and `frozen = false`. The owner pays its rent. If a close has advanced the nonce in the meantime, the instruction fails with `StaleRentRecoveryNonce` and the client reseals for the new value.
6. **Client checks.** It reads the account, opens the self-record and confirms the opening reproduces C_{bal} .

After onboarding, a payer can deposit and a payee can be paid: any transfer to this account seals its payee envelope to the KeyRecord at the account's `key_epoch`.

7.4 Deposit

A deposit moves a public amount a of USDC into the pool. With the ramp fee $f = \lfloor a \cdot \text{ramp_fee_bps} / 10\,000 \rfloor$ (5 basis points), $a - f$ goes into the vault and onto the owner's commitment and f into the protocol fee account.

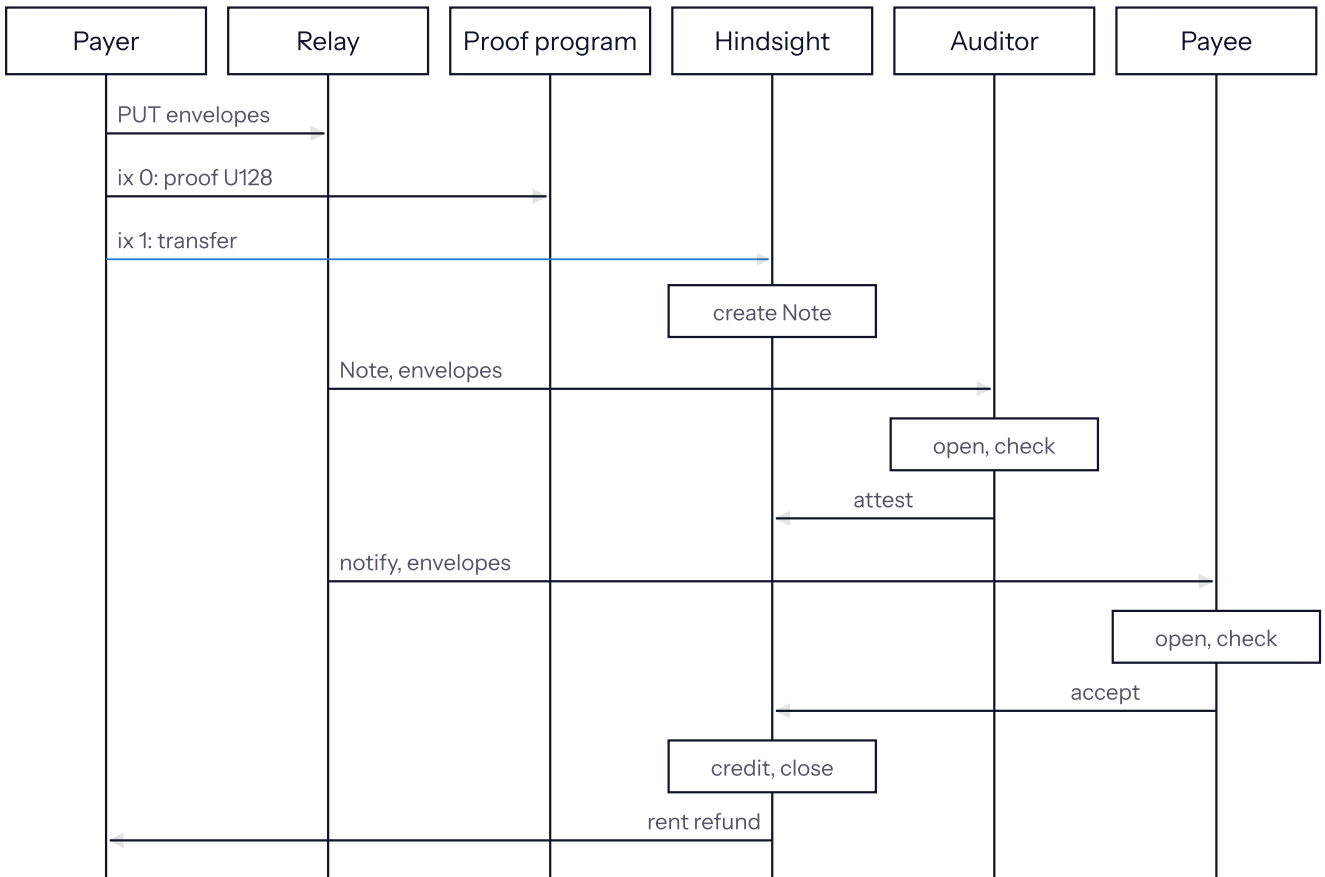
1. **Client reads its state.** It reads the ConfAccount (sequence s , C_{bal} , self-record), opens the self-record to (b, ρ) and checks it against C_{bal} .

2. **Client prepares the new state.** It reads the ramp fee from the pool's policy. The new opening is $(b + a - f, \rho)$, since a deposit keeps the blinding, and the new commitment is $C'_{bal} = C_{bal} + (a - f) \cdot G$. It seals a self-record for sequence $s + 1$ and C'_{bal} .
3. **Owner signs** `deposit(a, s, self_record)` and submits it.
4. **Chain changes.** The program checks the frozen flag, the sequence, $0 < a \leq \text{deposit_cap}$ and $\text{principal} + (a - f) \leq \text{pool_cap}$. It transfers $a - f$ from the owner's token account to the vault and f to the protocol fee account, sets C'_{bal} , adds $a - f$ to `principal`, increments the sequence to $s + 1$ and stores the new self-record.
5. **Client confirms.** It reads the account at sequence $s + 1$ and opens the self-record to $b + a - f$.

The amount, the owner and the vault balance are public, as they are for any SPL transfer. Once a note with a secret blinding has entered or left the account, its balance is known only to its owner.

7.5 Transfer

A transfer moves a hidden amount v from the payer's balance into a Note for the payee. The diagram shows it together with the attestation and acceptance that follow.



Transfer, attestation and acceptance. The payer uploads the envelope pair to the relay, then submits one v1 transaction: the native U128 range proof at index 0 and transfer at index 1 (highlighted). The program debits the payer and creates the Note. The auditor service opens its envelope, checks the opening against C_{note} and attests. The payee opens its envelope, checks it and accepts; the program credits the payee, closes the Note and refunds its rent to the payer.

Payer client, before signing:

1. **Resolve the payee and the auditor.** Read the payee's ConfAccount (owner wallet, `key_epoch`, not frozen) and the payee's KeyRecord at that epoch; read the Pool's `auditor` and `auditor_kem_epoch` and the auditor's KeyRecord. Key lookups through the API are hints. The client compares each hinted key byte for byte with the raw account it reads from the chain, after checking the owner, the epoch and the PDA.
2. **Read its own state.** Read its ConfAccount (sequence s , C_{bal}) and open its self-record to (b, ρ) .
3. **Commit to the amount.** Choose $v \leq b$, sample a fresh blinding r from a CSPRNG, pad the memo to 32 bytes, and form $C_{note} = v \cdot G + r \cdot H$.
4. **Seal the two envelopes.** Encrypt the 72-byte opening $v \parallel r \parallel memo$ to the payee's X-Wing key (role payee) and to the auditor's (role auditor). Each envelope's authenticated data binds the pool, the sender, the payee wallet, the sequence s , that recipient's key epoch and C_{note} . The pair hash is $h = \text{SHA256}(E_{payee} \parallel E_{auditor})$.
5. **Prove.** Compute the remaining opening $(b - v, \rho - r)$ and $C'_{bal} = C_{bal} - C_{note}$, and generate a U128 range proof over $[C_{note}, C'_{bal}]$ in a browser Worker.

6. **Seal its self-record** for sequence $s + 1$ and C'_{bal} .
7. **Upload the envelopes.** Send the pair to the relay at `PUT /v1/envelopes/{h}` with the session and an idempotency key. The relay checks both lengths and the hash and holds the pair as pending; it does not serve it until a Note on chain names this sender and this hash.
8. **Owner signs and submits** the v1 transaction `[VerifyBatchedRangeProofU128, transfer]`.

Chain:

9. The ZK ElGamal Proof program verifies the proof.
10. `transfer` checks the sender's frozen flag and sequence, the recipient's frozen flag, the envelope lengths, the commitment encoding and the proof context. It sets $C_{bal} = C'_{bal}$, increments the sender's sequence to $s + 1$, stores the self-record, and creates the Note at `["note", pool, payer, s]` with the commitment, the envelope hash, the creation slot and both key epochs. The payer pays the Note's rent.

After the transaction:

11. The indexer records the transaction, including the envelopes from its instruction data. The relay promotes the pending pair once it matches the chain Note's sender and envelope hash. The payee's notification stream carries a finalized reference to the new Note.

The transfer makes public that the payer paid the payee at that slot, and publishes the commitment and two ciphertexts. The amount and the memo stay sealed: on chain C_{note} is perfectly hiding, and in the envelopes they are protected by X-Wing and AES-256-GCM. Each party sees this much of one transfer:

Party	Sees	Can open
Any chain observer	sender, payee, slot, C_{note} , both envelopes, the sender's new commitment and self-record	nothing
Relay and indexer	the same, plus the envelope pair by hash	nothing
Auditor	the same	the auditor envelope: v, r , memo
Payee	the same	the payee envelope: v, r , memo
Payer	everything it built	its own opening and self-record

7.6 Attestation

When `attest_required` is set, a payee can accept a note only after the auditor has attested it. The auditor service runs this for every new Note in its pools.

1. **Discover.** The service reads new transfers from the indexer's confirmed feed. It treats the feed, and the relay, as untrusted discovery sources.
2. **Inspect the chain.** It fetches the Note, the Pool and the KeyRecords itself at confirmed commitment, and checks program ownership, discriminators, the Note's PDA, pool, sender, recipients, commitment, both epochs and that the Pool's current auditor is its own key.
3. **Check the envelopes.** It recomputes $\text{SHA256}(E_{\text{payee}} \parallel E_{\text{auditor}})$ and requires it to equal `envelope_hash`.
4. **Open.** Inside its custody it opens the auditor envelope with the X-Wing key of the Note's `auditor_kem_epoch`, using the payee wallet, sender, sequence, commitment and pool as authenticated data, and checks $v \cdot G + r \cdot H = C_{\text{note}}$. A Note that fails to open, or whose opening does not match, is quarantined and never attested.
5. **Record.** The amount and memo go into the auditor's ledger, encrypted at rest with AES-256-GCM under an auditor-only key.
6. **Sign the batch.** It inspects each Note again, builds one `attest` transaction naming the batch, signs it with the attestation key, persists the exact signed bytes and then submits them.
7. **Chain changes.** `attest` sets `attested = true` on every Note in the batch. Nothing else changes.

The service tracks each Note as a durable job:

Job state	Meaning
QUEUED	Discovered; not yet opened and checked
VERIFIED	Envelope opened and opening checked against the chain Note
SUBMITTED	Signed attestation batch persisted and sent
CONFIRMED , FINALIZED	The attestation landed at that commitment level
FAILED_TO_OPEN , OPENING_MISMATCH	Envelope content rejected; the Note stays unattested and can be returned
CLOSED	The Note was returned or is gone before attestation; ineligible
UNAVAILABLE , CONFIGURATION_REQUIRED	Transient read failure, or a key or epoch that does not match the Pool
ATTESTED_UNKNOWN	The chain shows <code>attested = true</code> without an observed receipt time

Retries back off at 1, 2, 4, 8, 16 and at most 30 seconds. If a submission's response is lost, the service queries the recorded signature and resends the identical signed bytes while they are valid. The protocol targets an attestation p95 under 10 seconds. A local six-Note run measured a p95 of 4,264 ms from the payer observing its transfer confirmed to the auditor observing its attestation confirmed.

7.7 Accept

The payee checks each incoming Note and accepts the ones that open correctly, in one batch.

1. **Learn of the Note.** The payee's client receives a finalized notification on `GET /v1/events` (resumable with `Last-Event-ID`), or lists its owned Notes. Notifications carry public references only, never amounts.
2. **Fetch and check the envelopes.** It fetches the pair by hash from the relay, or reads it from the transfer's instruction data, and requires $\text{SHA256}(E_{\text{payee}} \parallel E_{\text{auditor}})$ to equal the Note's `envelope_hash`.
3. **Open.** It opens the payee envelope with the key of the Note's `recipient_key_epoch`. Opening checks the recovered (v, r) against C_{note} before returning it; any failure returns one fixed error.
4. **Wait for attestation.** If the pool's `attest_required` is set, it waits until the Note's `attested` flag is true.
5. **Prepare the new state.** It reads its ConfAccount (sequence s , C_{bal}), opens its self-record to (b, ρ) , and computes $(b + \sum v_i, \rho + \sum r_i)$ and $C'_{\text{bal}} = C_{\text{bal}} + \sum_i C_{\text{note},i}$. It seals a self-record for $s + 1$

and C'_{bal} .

6. **Owner signs** `accept(s, self_record)` with each Note followed by its sender's wallet as remaining accounts.

7. **Chain changes.** For each Note the program checks the pool, the canonical address, that the signer is the recipient, that the paired wallet is the sender, and the attestation policy. It sets $C_{bal} = C'_{bal}$, increments the sequence, stores the self-record, and closes every Note, paying its rent to its sender.

A Note that does not open never enters a balance; the payee returns it instead. Because acceptance is batched, a payee with several incoming payments spends one transaction and one sequence step on all of them.

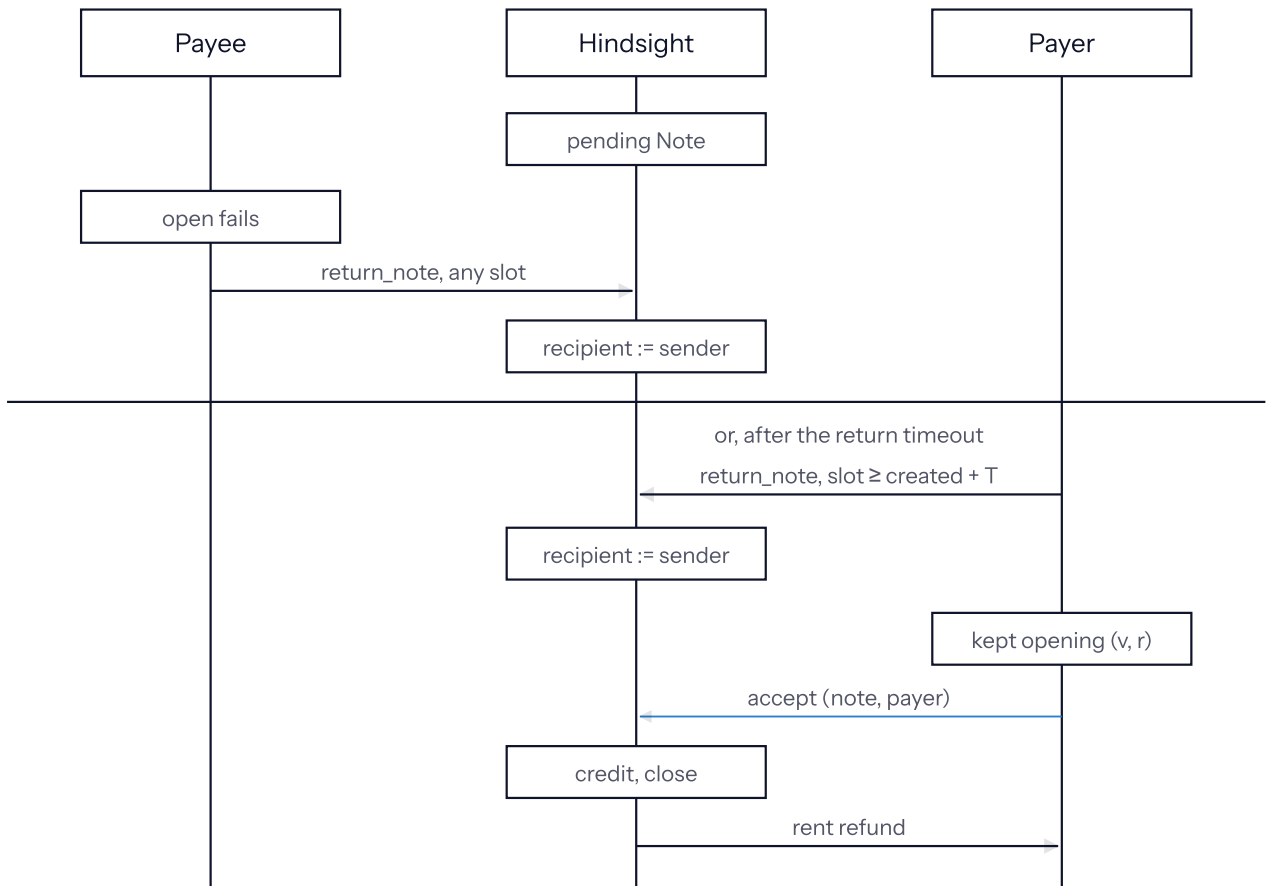
The API reports chain states only; the payee's client combines them with its own local check to decide what to show:

Chain state of the Note	Local result	Shown to the payee as
pending, pool does not require attestation	opening checked	ready to accept
pending, pool requires attestation	opening checked	waiting for the auditor
attested	opening checked	ready to accept
pending or attested	open failed or opening mismatch	failed to open; return it
returned		back with the sender
closed after an <code>accept</code> instruction		accepted

The payer's client does the same for its outgoing Notes: it derives "pending" and "timeout reached" from each Note's `created_slot`, the pool's `return_timeout_slots` and the confirmed slot.

7.8 Return and recovery

A Note that should not stay with its payee goes back to its sender through `return_note`. Returning moves only the right to accept the Note; no balance changes until someone accepts it.



Return and recovery. Above the line, the payee returns a Note whose envelope fails to open, at any slot. Below it, the payer takes back a Note the payee has left unaccepted once the pool's return timeout T has passed. Either way the Note's recipient becomes its sender. The payer then accepts it with the opening it kept from the transfer (highlighted); no attestation is needed for a sender's own returned Note. The program credits the payer, closes the Note and refunds its rent.

7.8.1 Payee return

1. **Payee's client fails to open the Note**, or finds that the opening does not match C_{note} , or the payee declines the payment.
2. **Payee signs** `return_note`. It may do so at any slot, whether or not the Note is attested.
3. **Chain changes**. The Note's `recipient` becomes its `sender`, `returned` becomes true and `attested` becomes false. No balance, sequence or principal changes.

A payee whose account is frozen can still return a Note, since returning changes no balance.

7.8.2 Sender recovery after the return timeout

1. **Payer's client watches its outgoing Notes**. For each pending Note it compares the confirmed slot with `created_slot + return_timeout_slots` from the pool's policy.
2. **Payer signs** `return_note` once the slot has reached that bound.

3. **Chain changes.** The same as a payee return: recipient becomes sender, `returned` true, `attested` false.

A Note can be returned once. The two return paths together mean every unaccepted Note has a way back: the payee can hand it back at any slot, and the sender can reclaim it once the timeout has passed.

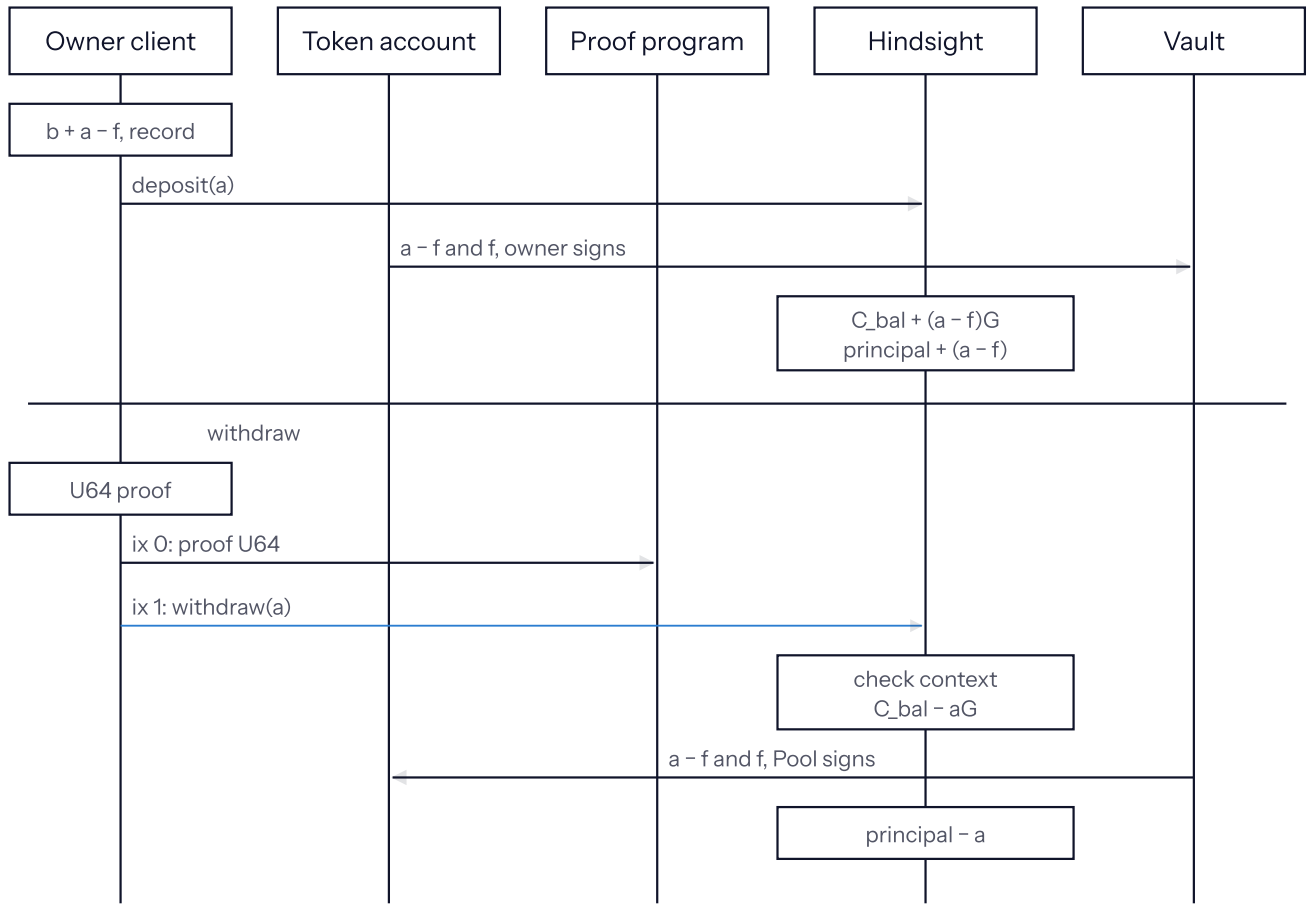
7.8.3 Sender accepts its returned note

1. **Payer recalls the opening.** It uses the Note opening (v, r) it created at transfer time and kept in its client records or payroll journal. It opens no envelope, so recovery works even when the envelopes it sent were unreadable.
2. **Payer prepares the new state.** It opens its current self-record to (b, ρ) , which already excludes v , computes $(b + v, \rho + r)$ and $C'_{bal} = C_{bal} + C_{note}$, and seals a self-record for the next sequence.
3. **Payer signs** `accept` with the pair (Note, its own wallet).
4. **Chain changes.** The policy clause for returned Notes admits it without attestation. The program adds C_{note} to the payer's balance, increments its sequence, closes the Note and returns its rent to the payer.

The payer ends with its balance restored and its rent refunded; it has spent only transaction fees. A sender who seals unreadable envelopes therefore harms only itself: its balance stays debited until it recovers the Note, and the payee is never credited with an amount it cannot open. The validator flow exercised this path: a transfer carrying garbage envelopes stayed unattested, was returned, and was accepted back by its sender.

7.9 Withdraw

A withdrawal subtracts a public amount a from the owner's commitment and pays it out of the vault. The diagram shows it together with the deposit, which is its public counterpart.



Deposit and withdrawal. Above the line, the owner signs `deposit(a)`; the program moves $a - f$ USDC from the owner's token account into the vault and the ramp fee f into the protocol fee account through the SPL Token program, adds $(a - f)G$ to the balance commitment and $a - f$ to the pool's principal. Below it, the owner submits one v1 transaction with the native U64 range proof on $C_{bal} - aG$ at index 0 and withdraw at index 1 (highlighted); the program checks the proof context, subtracts aG , pays $a - f$ from the vault to the owner and f to the protocol fee account with the Pool PDA as token authority, and subtracts a from principal. Both amounts are public.

1. **Client reads its state.** It reads its ConfAccount (sequence s , C_{bal}) and opens its self-record to (b, ρ) .
2. **Client prepares the new state.** It chooses $0 < a \leq b$. The new opening is $(b - a, \rho)$, since a withdrawal keeps the blinding, and the new commitment is $C'_{bal} = C_{bal} - a \cdot G$.
3. **Client proves.** It generates a U64 range proof over $[C'_{bal}]$ with the opening $(b - a, \rho)$, in a browser Worker.
4. **Client seals its self-record** for sequence $s + 1$ and C'_{bal} .
5. **Owner signs and submits** the v1 transaction `[VerifyBatchedRangeProofU64, withdraw(a, s, self_record)]`, naming as destination a token account of the pool's mint, normally the owner's own.
6. **Chain changes.** The native program verifies the proof. `withdraw` checks the frozen flag, the sequence, $a > 0$, that the destination is not the vault, and that the proof context is exactly $[C_{bal} - a \cdot G]$. It sets C'_{bal} , transfers $a - f$ from the vault to the destination and f to the protocol fee account with the Pool PDA as signer, subtracts a from `principal`, increments the sequence and stores the self-record.

7. **Client confirms.** It reads the account at $s + 1$ and opens its self-record to $b - a$.

A full withdrawal is an ordinary withdrawal with $a = b$. The remaining commitment $\rho \cdot H$ is a nonidentity commitment to zero, so its proof can be expressed. After deposits only, ρ is the public initial blinding r_0 ; after any confidential activity it is a secret sum of note blindings.

7.10 Close

An owner whose balance is zero takes back the rent of its ConfAccount and, if it wants, of its KeyRecord.

1. **Client reads its state.** After a full withdrawal it reads its ConfAccount (sequence s , $C_{bal} = \rho \cdot H$), opens its self-record to $(0, \rho)$, and reads the registry's close nonce n .
2. **Client proves.** It draws a fresh nonce k , computes $R = k \cdot H$, the challenge c over the program, pool, account, owner, s , n , C_{bal} and R , and the response $z = k + c\rho$.
3. **Owner signs** `close_account(s, n, R, z)`.
4. **Chain changes.** The program checks the frozen flag, the sequence, the nonce and $z \cdot H = R + c \cdot C_{bal}$. It advances the close nonce to $n + 1$ and closes the ConfAccount, paying every lamport to the owner.
5. **Owner signs** `close_key_record`, optionally. The program checks that no pool names the KeyRecord as its auditor key and closes it, paying every lamport to the owner.

Notes sent from or to the account stay open. To accept a returned or late Note, the owner registers its key again if it closed it and onboards again at the same ConfAccount address, with the nonce current at that time; the new account's sequences start above every sequence of the old one, so its Note addresses never collide with the old account's.

7.11 Payroll batch

A payroll run is many transfers from one payer, prepared together and executed as ordered chains. All of it happens in the payer's browser or in a process the payer controls. The hosted API receives only opaque envelopes and public progress references, never the CSV, an amount, a memo, an opening or a key.

7.11.1 Planning lanes

1. **Parse.** The client parses a CSV with the columns `payee` and `amount`, and optionally `row_id` and `memo`. The amount unit is explicit, either `usdc` (at most six decimals, converted with integer arithmetic) or `base_units` (exact u64 integers). Memos are at most 32 UTF-8 bytes. The parser takes up to 10,000 rows and reports issues by row, field and code, without echoing values.
2. **Choose lanes.** A lane is a ConfAccount that sends transfers in order. Accounts are addressed by `["acct", pool, owner]`, one per owner per pool, so parallel lanes are accounts of distinct payer-controlled wallets.

3. **Distribute.** Rows are dealt round-robin across the lanes, preserving each lane's input order.

7.11.2 Preparing ordered chains

Preparation runs in two phases, so nothing is encrypted until everything has been checked.

1. **Preflight from the chain.** Through an RPC endpoint the payer trusts, the client reads the raw Pool, every lane account, every payee account and every payee and auditor KeyRecord. It checks program ownership, discriminators, exact sizes and PDAs, binds the auditor and its epoch to the actual Pool, compares every hinted X-Wing key with the chain bytes, and rejects frozen accounts. It then opens each lane's self-record, checks it against the lane's commitment, and checks that the lane's balance covers the sum of its rows.
2. **Precompute every step.** For a lane at sequence s with opening (b_0, ρ_0) and rows $v_1, \dots, v_n$, step k samples a fresh r_k and fixes

$$C_{note,k} = v_k G + r_k H, \quad C^{(k)} = C^{(k-1)} - C_{note,k}, \\ (b_k, \rho_k) = (b_{k-1} - v_k, \rho_{k-1} - r_k)$$

with $C^{(0)} = C_{bal}$. Step k runs at sequence $s + k - 1$, so its Note address

["note", pool, lane owner, $s + k - 1$] is known in advance. Its two envelopes and its self-record for sequence $s + k$ and $C^{(k)}$ are sealed now. Range proofs are not: each is generated just before its step is signed.

3. **Seal the plan.** The prepared plan is encrypted with AES-256-GCM under a payer-held key and identified by the SHA-256 digest of its ciphertext. A separately encrypted journal records each step's status.

7.11.3 Executing

1. **Lanes run concurrently; steps within a lane run in order**, because each step spends the sequence and balance the previous one produced.
2. **Before each step**, the executor reads the lane account and the payee account and requires the lane to be at exactly the step's sequence, commitment and self-record, unfrozen, with the payee unfrozen at the expected key epoch.
3. **Each step is one transfer:** generate the U128 proof over $[C_{note,k}, C^{(k)}]$, build, have the lane owner's wallet sign, persist the signed bytes and expiry, submit, and observe the expected Note. Lost responses, resends and blockhash refreshes follow the procedure in the transactions chapter; a refreshed attempt keeps the step's Note address, sequence, commitment, opening, envelopes and self-record.
4. **Anything unexplained pauses the lane.** A lane account is reserved for the run: an `accept` or `deposit` into it changes its sequence and pauses the lane until the payer replans. Other lanes continue.

Failure code	Meaning
ACCOUNT_CHANGED	The lane or payee account is not in the state the plan expects
NOTE_MISMATCH	A Note exists at the step's address with different contents
CHAIN_REJECTED	The transaction failed on chain
SUBMISSION_UNCERTAIN	No landing observed within the polling window
ADAPTER_UNAVAILABLE	The wallet, RPC or storage adapter failed
ATTEMPTS_EXHAUSTED	The step used its signed attempts

7.11.4 Progress and settlement

The journal's `completed` state means every transfer landed. Settlement is later: each payee still opens, checks and accepts its Note. The hosted progress service stores the plan digest, the lane accounts and the ordered Note addresses, and derives accepted and returned counts from finalized chain state, not from client-reported counters. A returned payout is reported as a settlement failure. The payer recovers it with `return_note` and `accept` as above, and creates a replacement payout as a separate, explicit action. The payroll gate exercised this path with a 50-payee batch across two lanes, including an encrypted-journal restart and a deliberately lost RPC response.

7.12 Chain effects at a glance

Flow	Signer	Accounts created	Accounts changed	Sequence	principal	Public amount
Pool setup	admin, auditor	RentRecoveryRegistry (once), Pool, vault, auditor KeyRecord	registry		set to 0	
Onboarding	owner	KeyRecord, ConfAccount		starts at $n \cdot 2^{32}$		
Deposit	owner		ConfAccount, vault	owner +1	$+a$	a
Transfer	payer	Note	payer's ConfAccount	payer +1		none
Attestation	auditor		Notes			none
Accept	payee		payee's ConfAccount; Notes closed	payee +1		none
Return	payee, or payer after the timeout		Note			none
Recovery accept	payer		payer's ConfAccount; Note closed	payer +1		none
Withdraw	owner		ConfAccount, vault	owner +1	$-a$	a
Close	owner		ConfAccount closed, KeyRecord optionally closed; registry nonce +1			none

Flow	Signer	Accounts created	Accounts changed	Sequence	principal	Public amount
Payroll batch	each lane owner	one Note per row	each lane's ConfAccount	lane $+n$		none

8 The auditor

Every Hindsight pool names one auditor. The auditor is the second reader of every payment in its pool: each `transfer` carries an envelope sealed to the auditor's X-Wing key beside the envelope sealed to the payee, and when the pool's policy requires it, a payee can credit a Note only after the auditor has opened its own envelope, checked the opening against the on-chain commitment and recorded that check on chain with `attest`. This chapter describes the auditor's place in the program, the service that performs the checks, the states each check moves through, the key epochs the service holds, and the console through which operators read the result.

8.1 The auditor's place in a pool

`init_pool` fixes the auditor in the `Pool` account through two fields:

Field	Type	Meaning
<code>auditor</code>	<code>Pubkey</code>	Ed25519 attestation key; the only signer <code>attest</code> accepts for this pool
<code>auditor_kem_epoch</code>	<code>u64</code>	Epoch of the auditor's <code>KeyRecord</code> ; every new Note's auditor envelope is sealed to this key

The auditor's encapsulation key is an ordinary `KeyRecord` at seeds `["kem", auditor, epoch]`, written with `register_key` like any payee key. `transfer` binds it to the pool: the `auditor_key_record` account must be derived from `pool.auditor` and `pool.auditor_kem_epoch`, its `owner` must equal `pool.auditor` and its `epoch` must equal `pool.auditor_kem_epoch`, or the instruction fails with `InvalidAuthority`. A payer therefore cannot direct the auditor envelope to a key of its choosing: the transaction names the pool's current auditor key or it does not execute. Each Note copies that epoch into its own `auditor_kem_epoch`, beside `recipient_key_epoch` for the payee.

The auditor's on-chain authority is one boolean. `attest` sets `attested` on Notes of its own pool. The auditor key signs no `transfer`, `accept`, `withdraw`, `freeze` or `thaw`; it cannot change a balance commitment, move USDC out of the vault or block an account. Freezing belongs to a separate key, the pool's `freeze_authority`.

8.2 The attestation gate

The policy field `attest_required` makes the auditor's check a precondition of credit. `accept` evaluates, for every Note in its batch,

$$\neg \text{attest_required} \vee n.\text{attested} \\ \vee (n.\text{returned} \wedge n.\text{recipient} = n.\text{sender})$$

and fails with `AttestationRequired` when the expression is false. With the gate on, an amount enters a payee balance only after two parties have independently opened the same commitment: the payee, whose client checks its opening before it builds `accept`, and the auditor, whose service checks its own envelope before it signs `attest`.

The third clause is the recovery path. `return_note` re-targets a Note to its sender and clears `attested`; the sender then accepts the Note back with the opening it generated itself. Recovering a Note whose envelopes were garbage never waits for the auditor, and a returned Note never carries an earlier attestation into a new balance.

`attest` takes no arguments. Its accounts are the pool and the auditor signer (`has_one = auditor`), followed by the Notes as remaining accounts. For each Note the program requires:

- a non-empty batch (`EmptyBatch`) in which no Note appears twice (`DuplicateNote`);
- a writable account (`AccountNotWritable`) that deserializes as a program-owned `Note`;
- `note.pool` equal to the signing pool (`WrongPool`), and an address equal to `create_program_address(["note", pool, sender, sequence_le, bump])` with the Note's stored canonical bump (`InvalidNote`);
- a Note that has not been returned (`NoteAlreadyReturned`).

It then sets `attested = true`. The batch commits or fails as a whole with its transaction.

An attestation means more than "the auditor saw an envelope". The auditor's check establishes $v_a G + r_a H = C_{note}$ for the opening (v_a, r_a) in its envelope; the payee's check establishes $v_p G + r_p H = C_{note}$ for the opening in its own. Two distinct openings of one commitment would yield the discrete logarithm of H to base G , so while Pedersen binding holds, $v_a = v_p$: the amount the auditor records is the amount the payee can credit. Binding has to hold only while the checks and `accept` execute, which is the split the design rests on: confidentiality must last indefinitely, soundness must hold at verification time.

8.3 The auditor service

The service is a separate process with its own PostgreSQL database, its own HTTP origin, its own session cookie and its own keys. It trusts nothing it receives from the hosted API: the indexer's database is a discovery source, and every fact the service acts on is re-read from the chain over RPC.

A runtime tick performs four steps in order:

1. Confirm that the RPC's genesis hash, runtime version and program account still match the configured cluster.
2. `reconcile` : resolve the receipts of attestation transactions already submitted.
3. `discover` : read candidates from the feed, then open and check their auditor envelopes.
4. `attest` : batch checked Notes per pool, sign and submit.

After a tick, at most one background provenance repair runs. The loop repeats every `AUDITOR_POLL_MS` (250 ms by default), and expired exports, idempotency rows, challenges, sessions and ledger ciphertext are pruned at most once a minute.

8.3.1 Discovering Notes

`PrismaAuditorFeed` reads the hosted indexer's database through its own connection string, `AUDITOR_FEED_DATABASE_URL`, which logs in as `hindsight_auditor_feed` : a role with `SELECT` on nine public chain tables and `default_transaction_read_only` set. Each poll is one repeatable-read transaction drawing on two sources:

- **Confirmed transfers:** successful `TRANSFER` instructions recorded at confirmed commitment above the last finalized slot. They let the auditor work before finality, so attestation latency does not include the finalization delay.
- **Finalized pending snapshots:** Notes present in the latest finalized snapshot with `attested = false` and status `PENDING`, with their stored envelope pair. They cover Notes whose creating transaction the feed no longer holds, after a restart or RPC history pruning.

The feed refuses a checkpoint from another genesis or program, and pauses discovery with `FEED_STALE` when the indexer's last healthy checkpoint is older than 30 seconds, while reconciliation and attestation of checked Notes continue. Pages hold 500 rows; a poll that would exceed 20,000 rows fails with `FEED_LIMIT_EXCEEDED` rather than returning a truncated set. An undecodable transaction, or a snapshot whose stored envelopes fail their hash, is omitted on its own and counted.

A `TRANSFER` is decoded from its canonical 2,540-byte prefix. The program tolerates trailing bytes after it; the indexer, the feed and the service ignore them.

Offset	Length	Field
0	8	Anchor instruction discriminator
8	8	<code>expected_sequence</code> , u64 little-endian
16	32	<code>commitment</code> C_{note} , compressed Ristretto
48	4	payee envelope length, must be 1,208
52	1,208	payee envelope
1,260	4	auditor envelope length, must be 1,208
1,264	1,208	auditor envelope
2,472	68	sender's new self-record

A candidate carries the Note address, the pool, a minimum slot, the expected sender, sequence, commitment, envelope-pair hash, original recipient, both key epochs and both envelopes. The service treats each value as a claim to be confirmed. Jobs are keyed by **SHA-256(cluster : program : genesis : note)**, so one Note address in one deployment maps to exactly one job.

8.3.2 Reading the chain

`inspect` reads the Note at confirmed commitment with `minContextSlot` set to the candidate's slot, takes the pool address from the Note itself (the feed's pool is only a hint), and then reads Note, Pool and the auditor `KeyRecord` together with `getMultipleAccounts` at a context no older than the previous read. It requires:

- program ownership and the Anchor discriminator of each account;
- the Pool and vault PDAs derived from the mint, the Note PDA derived from pool, sender and sequence, and the `KeyRecord` PDA derived from the pool's auditor and the Note's auditor epoch;
- identical Note and Pool contents across the reads, apart from the public principal, which ordinary deposits and withdrawals change;
- `key.owner = pool.auditor` and `key.epoch = note.auditor_kem_epoch` .

The service then compares the chain Note with every claim of the candidate and requires the pool to be one of `AUDITOR_POOL_ADDRESSES` . Last, it checks its own configuration: `pool.auditor` must be its attestation key, `note.auditor_kem_epoch` must equal `pool.auditor_kem_epoch` , and custody must hold that epoch with a public key equal, byte for byte, to the `KeyRecord` 's 1,216-byte encapsulation key. A Note that has been returned, or whose recipient no longer equals its original recipient, is closed instead of checked.

8.3.3 Opening the envelope

The service requires both envelopes to be 1,208 bytes and $\text{SHA-256}(\text{payee} \parallel \text{auditor})$ to equal the Note's `envelope_hash` , then passes the auditor envelope to its custody adapter with this envelope context:

Context field	Value
role	<code>auditor</code> , AAD byte 1
pool	the Note's pool, 32 bytes
sender	the Note's sender wallet, 32 bytes
recipient	the Note's original recipient wallet, 32 bytes
sequence	the Note's sender sequence, u64
keyEpoch	the Note's <code>auditor_kem_epoch</code> , u64
commitment	C_{note} , 32 bytes

The recipient is the payee's wallet, not the payee's `ConfAccount` , in both envelopes; the auditor envelope differs from the payee's in role byte, KEM key and key epoch. X-Wing decapsulation yields the shared secret, HKDF-SHA256 derives the AES-256-GCM key and nonce from it with the context in its `info` , and AES-256-GCM authenticates the 88-byte ciphertext against the same context as associated data, returning the 72-byte plaintext $v \parallel r \parallel \text{memo}$. Any change to the KEM ciphertext, the AEAD ciphertext or the context fails authentication.

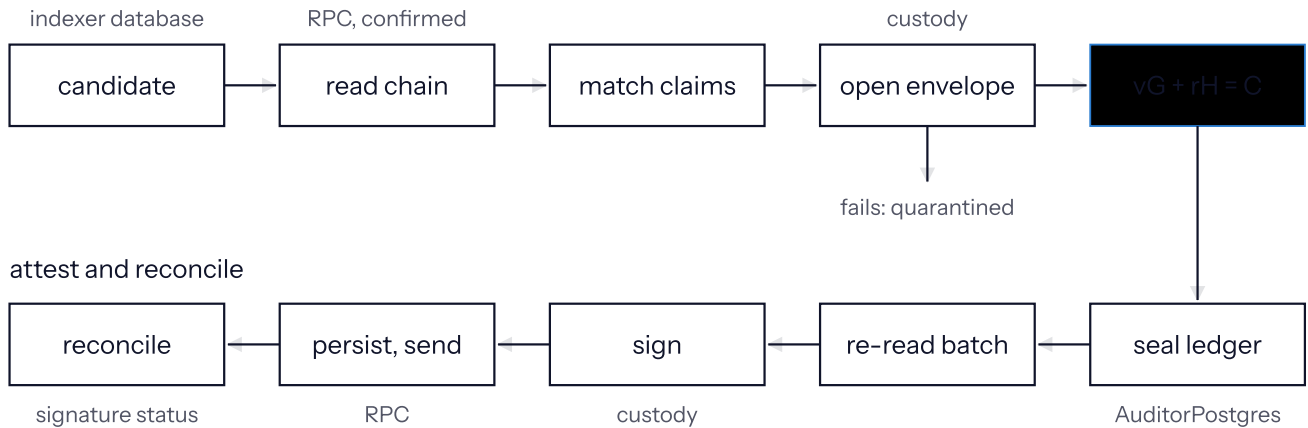
The service then evaluates

$$v G + r H \stackrel{?}{=} C_{note}$$

with G the Ristretto basepoint and H the solana-zk-sdk Pedersen generator. A failed decryption becomes `FAILED_TO_OPEN` ; a decryption whose opening does not reproduce C_{note} becomes `OPENING_MISMATCH` . Both quarantine that Note incarnation: the service never attests it, and the payee, whose client sees the same Note, returns it.

A successful check produces the auditor's private ledger entry `{amountBaseUnits, memoUtf8}` . The service seals it with AES-256-GCM under its ledger key `AUDITOR_LEDGER_KEY_HEX` , a random 12-byte nonce, and the associated data `hindsight:auditor-storage:v1:` followed by the job identifier, so a ciphertext cannot be moved to another row. The job becomes `VERIFIED` together with the ciphertext, the public Note snapshot and the confirmed slot of the check, and the blinding and memo buffers are zeroed. If the chain Note already shows `attested = true` , the job records `ATTESTED_UNKNOWN` instead: the check is complete, but the attestation transaction is not one the service observed.

discover



The attestation pipeline. Discovery turns an untrusted feed candidate into a checked opening; attestation re-reads each checked Note, signs one batch, persists the exact signed bytes before sending them, and reconciles the receipt on later ticks.

8.3.4 Attesting in batches

`attest` runs per configured pool. It selects `VERIFIED` jobs that are due and older than `AUDITOR_BATCH_DELAY_MS` (200 ms by default), up to `AUDITOR_BATCH_SIZE` (8 by default, at most 16), and claims each with a 60-second lease. The batch delay lets Notes that land close together share one transaction.

Before signing, the service re-reads every claimed Note at confirmed commitment, using the slot of its original check as the minimum. A Note that changed incarnation, was returned or was attested in the meantime leaves the batch through the matching transition. A Note that cannot be read is deferred, not discarded: a missing read is not evidence that a checked opening is wrong. The remaining Notes go into one v1 transaction built from the program IDL. The attestation key is fee payer and sole signer, the compute-unit limit is

$$CU(n) = \min(1\,400\,000, 30\,000 + 10\,000\,n), \quad 1 \leq n \leq 16,$$

and the loaded-account-data limit is 8 MiB, which covers the deployed program and its `ProgramData` account in addition to the pool, the Notes and the sysvars. On a local Agave 4.3.0 validator, a two-Note attestation measured 290 bytes and 9,179 CU, and a four-Note attestation 356 bytes and 15,267 CU.

The custody adapter signs only a transaction whose single required signer is the attestation key. The service then writes, in one database transaction, the exact signed wire bytes, the signature and the blockhash's last valid block height to every job of the batch, moving them to `SUBMITTED`. Only after that commit does it call `sendTransaction`. A crash between commit and send leaves a durable receipt that reconciliation can resend or resolve; a crash before the commit leaves `VERIFIED` jobs that a later batch picks up.

8.3.5 Reconciling receipts

`reconcile` runs at the start of every tick, before discovery, over three windows of at most 1,000 rows: submitted receipts, confirmed or attested-unknown receipts, and closed jobs that still hold a landed receipt. It groups rows by signature and reloads each signed cohort, at most 17 rows, so a scan window never splits a batch. Each signature is looked up with `getSignatureStatuses` and history search:

Status	Action
finalized	<code>receipt_finalized</code> to <code>FINALIZED</code> ; signed bytes released
confirmed	<code>receipt_confirmed</code> to <code>CONFIRMED</code> ; signed bytes kept until finality
failed	<code>receipt_failed</code> to <code>QUEUED</code> ; receipt archived, the Note opened and checked again before reuse
missing, blockhash still valid	resend the identical signed bytes
missing, blockhash expired	read the Note at finalized commitment and resolve

A resend requires an intact cohort: the stored bytes decode to one `attest` instruction (plus compute-budget instructions) signed and paid by the attestation key, its Note accounts are exactly the cohort's Notes, and every Note is still pending, unreturned, unattested and the same incarnation. Row locks on the cohort are held while it resends.

Expiry comes from `getEpochInfo` at finalized commitment, which returns block height and slot from the same bank. Once the finalized block height exceeds a receipt's last valid block height, the service reads the Note at a finalized context no older than that bank. An attested Note resolves to `ATTESTED_UNKNOWN` ; a returned Note to `CLOSED` ; an absent Note to `CLOSED` , unless its receipt had already landed; a new incarnation at the same address to `QUEUED` with the old opening discarded; a Note that is still pending to `VERIFIED` through `receipt_dropped` , so it joins the next batch. Every retired receipt moves atomically into the private `AttestationReceipt` table with its signature, wire bytes and timestamps.

Transient failures back off on a schedule persisted across restarts,

$$d_k = \min(30, 2^{\min(k,5)}) \text{ s},$$

that is 1, 2, 4, 8, 16 and then 30 seconds, where k counts the job's consecutive transient failures. Only a validated key mismatch (`AuditorKeyConfigurationMismatch`) consumes the configuration budget: after five such attempts the job moves to `CONFIGURATION_REQUIRED` with a fingerprint of the configuration it failed under (RPC, attestation key, pools and per-pool epoch public keys). Such jobs leave

active batches until the fingerprint changes; the service compares fingerprints at startup, on every change and at most once a minute otherwise.

8.3.6 Recovering provenance

A Note supplied only as a snapshot lacks its creation time and signature. A bounded background repair performs at most one lookup per tick: it pages the Note address's own finalized history (four pages of 64 signatures, eight transaction fetches, a five-second deadline) and accepts only a successful `transfer` at the Note's creation slot with the expected pool, sender, sequence, commitment and envelope hash. It fills missing public fields and never reopens an envelope, changes a status or signs.

8.4 Job states and transitions

`job-state.ts` is the runtime authority over job status. Every creation and every status change goes through its transition functions, which validate the destination, fence the allowed source states in SQL together with the caller's lease, scope and receipt guards, and refuse `VERIFIED` or `SUBMITTED` without ledger ciphertext. A closure can never retire an attestation receipt that has landed.

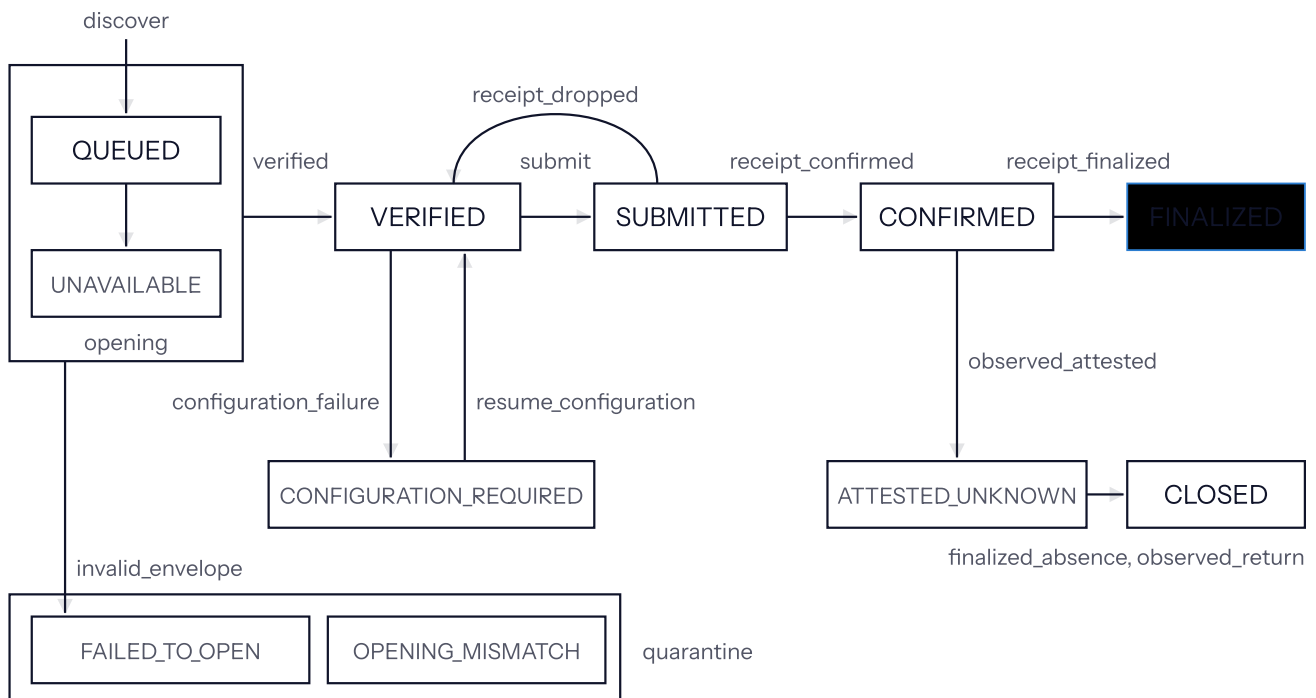
State	Meaning
QUEUED	Discovered; the envelope of this incarnation not yet opened and checked
UNAVAILABLE	Deferred: a read, an envelope or the pool allowlist was unavailable; retried with backoff
VERIFIED	Opened and checked; encrypted ledger entry stored; eligible for a batch
SUBMITTED	Exact signed batch persisted and sent
CONFIRMED	Attestation receipt confirmed; kept until finality
FINALIZED	Attestation receipt finalized; historical and never reopened
ATTESTED_UNKNOWN	The chain Note shows <code>attested = true</code> ; confirmation time not observed
CLOSED	Note absent at a finalized context, or returned; evidence retained
FAILED_TO_OPEN	Envelope did not decapsulate or authenticate
OPENING_MISMATCH	Envelope opened, but $vG + rH \neq C_{note}$
CONFIGURATION_REQUIRED	Key or epoch mismatch after five attempts; waits for a configuration change

In the transitions below, *opening* means `QUEUED` or `UNAVAILABLE` , *receipt* means `SUBMITTED` , `CONFIRMED` or `ATTESTED_UNKNOWN` , and *mutable* means every state except `FINALIZED` .

Event	From	To	Evidence
discover	no job	QUEUED	An untrusted candidate creates work only
verified	opening	VERIFIED	Incarnation, envelopes an opening checked independent
discovery_deferred	opening	UNAVAILABLE	Transient rea or envelope availability failure
invalid_envelope	opening	FAILED_TO_OPEN or OPENING_MISMATCH	Deterministic content quarantine
submit	VERIFIED	SUBMITTED	Exact signed batch persisted before sending
opening_required	VERIFIED	QUEUED	Ledger ciphertext pruned; reopen before signing
identity_contradiction	VERIFIED	UNAVAILABLE	Observed mismatch; opening cleared and checked again
excluded_pool	VERIFIED	UNAVAILABLE	Pool outside this worker's allowlist; evidence kept

Event	From	To	Evidence
configuration_failure	opening, VERIFIED	same state; CONFIGURATION_REQUIRED after five	Validated attestation key or epoch mismatch
resume_configuration	CONFIGURATION_REQUIRED	VERIFIED with ciphertext, else QUEUED	Configuration fingerprint changed
receipt_confirmed	receipt, CLOSED with landed receipt	CONFIRMED	Successful receipt observed
receipt_finalized	receipt, CLOSED with landed receipt	FINALIZED	Successful receipt finalized
receipt_failed	receipt, CLOSED with landed receipt	QUEUED	Failed receipt archived; reopen and check before reuse
receipt_dropped	receipt, CLOSED with landed receipt	VERIFIED with ciphertext, else QUEUED	Expired, missing receipt and a matching pending Note at a recent finalized banl
observed_attested	opening, VERIFIED , receipt, CLOSED	ATTESTED_UNKNOWN	Chain Note has attester = true
finalized_absence	mutable, no landed receipt	CLOSED	Note absent at a sufficiently recent finalized context
observed_return	mutable, no landed receipt	CLOSED	Note returne or re-

Event	From	To	Evidence
			targeted; ineligible for attestation
replacement	mutable	QUEUED , UNAVAILABLE or CLOSED	New incarnation a the same address; old opening neve inherited
retry	mutable	same state	RPC or transport failure, slot lag, missing finality evidence; backoff and lease release



The auditor job state machine. The upper row is the path of an honest Note from discovery to a finalized attestation receipt. CLOSED is reachable from every state except FINALIZED; the table above lists every transition, including receipt_failed, replacement and the configuration loop.

The console counts attestations by one evidence rule. A job is attested when it holds a successful `CONFIRMED` or `FINALIZED` receipt with a signature and an observed time, or when it is `CONFIRMED`, `FINALIZED`, `ATTESTED_UNKNOWN` or `CLOSED` with a public Note snapshot whose `attested` is `true`. The first clause proves the attestation independently of the latest account snapshot, so a Note that was attested, then accepted and closed, still counts.

8.5 Key epochs

Auditor keys are organised by epoch. An epoch is a `u64` naming one X-Wing key pair; its public half is the `KeyRecord` at `["kem", auditor, epoch]`, written once by `register_key` and never modified. The pool points at one epoch through `auditor_kem_epoch`, each Note copies that epoch at creation, and the service opens every Note with the key of the Note's own epoch. That `KeyRecord`'s address is listed in the `RentRecoveryRegistry`, by `init_pool` or, for a pool that predates the registry, by `initialize_rent_recovery`, so `close_key_record` refuses it (`AuditorKeyInUse`) and the key every transfer of the pool names stays on chain. Custody is keyed by the pair (pool, epoch).

The software custody adapter, `SoftwareAuditorHsm`, holds an Ed25519 attestation seed and, for each pool and epoch, a 32-byte X-Wing seed with its 1,216-byte public key and an activation time. In the deployed service these arrive as the secrets `AUDITOR_ATTESTATION_SEED_HEX` and `AUDITOR_XWING_EPOCHS_JSON`. The adapter derives every key pair from its seed, rejects the custody set if a derived public key differs from a listed one or an epoch appears twice, wipes seed buffers after derivation and wipes all key material on shutdown. The `AuditorHsm` interface exposes exactly the two operations the worker needs, `open(pool, epoch, envelope, context)` and `sign(transaction)`, so a hardware-backed adapter replaces the whole implementation without changes to the worker.

The policy field `auditor_rotation_interval_seconds`, positive and fixed at `init_pool`, is the schedule the console reports against. For every epoch in custody the console shows its status, `rotationDueAt` and the number of outstanding Notes sealed to it:

Status	Condition
<code>active</code>	The pool's current auditor and epoch; activation time plus the rotation interval lies ahead
<code>rotation_due</code>	The pool's current epoch; activation time plus the rotation interval has passed
<code>retiring</code>	Not the current epoch; Notes sealed to it still lack attestation evidence
<code>retired</code>	Not the current epoch; no outstanding Notes

A pool's auditor epoch is set by its `init_pool` configuration and holds for the pool's life, so every Note in the pool is sealed to a key the auditor has held from the start. The `auditorRotation` call reports exactly that state, `{state: "configuration_required", epoch: null, unsigned: true}`, and never produces

a transaction. Because each Note stores the epoch it was sealed to, the seeds the service retains keep every Note readable to the auditor.

8.6 The console

The console is the operator's view of one or more pools. Fourteen calls serve it, all on the auditor origin under `/v1/auditor` :

Call	Method and path	Role
<code>auditorChallenge</code>	<code>POST /v1/auditor/auth/challenge</code>	public
<code>auditorVerify</code>	<code>POST /v1/auditor/auth/verify</code>	public
<code>auditorSession</code>	<code>GET /v1/auditor/auth/session</code>	any operator
<code>auditorLogout</code>	<code>POST /v1/auditor/auth/logout</code>	any operator
<code>auditorQueue</code>	<code>GET /v1/auditor/pools/:pool/queue</code>	any operator
<code>auditorHistory</code>	<code>GET /v1/auditor/pools/:pool/attestations</code>	any operator
<code>auditorMetrics</code>	<code>GET /v1/auditor/pools/:pool/metrics</code>	any operator
<code>auditorEpochs</code>	<code>GET /v1/auditor/pools/:pool/key-epochs</code>	any operator
<code>auditorRotation</code>	<code>POST /v1/auditor/pools/:pool/key-rotations</code>	MANAGER
<code>auditorExports</code>	<code>GET /v1/auditor/pools/:pool/exports</code>	any operator
<code>createAuditorExport</code>	<code>POST /v1/auditor/pools/:pool/exports</code>	MANAGER
<code>auditorExport</code>	<code>GET /v1/auditor/exports/:id</code>	any operator
<code>auditorExportContent</code>	<code>GET /v1/auditor/exports/:id/content</code>	any operator
<code>createDisclosure</code>	<code>POST /v1/auditor/pools/:pool/disclosures</code>	MANAGER

Operator roles are `VIEWER` , `ATTESTER` and `MANAGER` , granted per pool. Viewers and attesters read their scoped data; managers also create exports, selective-disclosure reports and rotation requests. The wallets listed in `AUDITOR_MANAGER_WALLETS` receive manager scopes for the configured pools at startup.

8.6.1 Queue and attested notes

The queue lists jobs that are neither `CLOSED` nor `FINALIZED`; the attestation history lists every job of the pool. Each row joins the public Note snapshot to the private check:

Field	Content
<code>note</code>	Address, pool, sender, recipient, original recipient, sequence, commitment, envelope hash, epochs, slots, state
<code>check</code>	<code>awaiting_check</code> , <code>valid</code> , <code>attested</code> , <code>failed_to_open</code> , <code>opening_mismatch</code> or <code>configuration_required</code>
<code>amountBaseUnits</code> , <code>memoUtf8</code>	Decrypted from the ledger entry; null when no ciphertext is retained
<code>keyEpoch</code> , <code>queuedAt</code> , <code>attestedAt</code> , <code>latencyMs</code>	Epoch of the check and timing of the attestation

Listings page by cursor (1 to 100 rows, 50 by default) and filter by date range and by account. An account filter verifies one confirmed `ConfAccount` of the pool, resolves its owner once and matches Notes whose sender or original recipient is that owner. A decrypted ledger entry must hold a canonical u64 amount and at most 32 UTF-8 bytes of memo; anything else omits the row instead of overriding the public Note.

`auditorMetrics` reports, over the last hour, the pending, failed-to-open and attested counts, the current count of `CONFIGURATION_REQUIRED` jobs regardless of age, and the nearest-rank 95th percentile of the time from first indexer observation to observed attestation confirmation, beside the 10,000 ms target.

8.6.2 Exports and their coverage

`createAuditorExport` produces a `csv` or `ledger` (JSON) artifact for a pool, optionally narrowed by date and account; `createDisclosure` produces a `selective_disclosure` report. Creation requires an `Idempotency-Key` of 8 to 128 characters. The service derives the retry identity as an HMAC over cluster, program, genesis, operator wallet, pool, operation and key, plus a digest of the request body: the same key with the same body replays the stored result, the same key with a different body fails with `IDEMPOTENCY_CONFLICT`.

An export reads the pool in 200-row cursor batches inside one repeatable-read transaction, within explicit bounds of 20,000 scanned candidates, 20,000 records and 16 MiB of content; a narrower date or account range serves a larger pool. The content is sealed with AES-256-GCM under the ledger key, with the export identifier in the associated data, kept for `AUDITOR_EXPORT_TTL_MS` (one hour by default), and served only by `auditorExportContent`, with `Cache-Control: no-store`, after the operator's scope has been checked against the export's pool.

CSV line	Content
----------	---------

1	<code>hindsight-auditor-export</code> , format version <code>2</code> , pool, kind, <code>omittedCount</code> , <code>unclassifiedCount</code>
---	--

2	<code>note,sender,recipient,amount_base_units,memo,status</code>
---	--

3	One quoted record per Note onward
---	-----------------------------------

Every CSV field is quoted, embedded quotes are doubled, and a value beginning with `=` , `+` , `@` or `-` after leading whitespace is prefixed with an apostrophe so spreadsheet software does not evaluate it. A JSON artifact is `{version: 2, pool, kind, coverage, records}` , each record holding the public Note, the amount in base units, the memo and the job status.

Coverage states what an artifact leaves out. `omittedCount` counts known omissions inside the requested filters: jobs whose public provenance is incomplete, jobs whose ledger ciphertext expired under `AUDITOR_LEDGER_RETENTION_DAYS` , and rows that failed validation. `unclassifiedCount` , for account-filtered requests, counts jobs whose participants could not be classified for that account. Both counts are frozen inside each new artifact's encrypted bytes and are returned by the listing, status and content calls with `includeCoverage=true` , which also adds coverage to queue and history listings. They bound omissions among the jobs the service holds, not chain history it never observed.

8.6.3 Selective disclosure

A selective-disclosure report is an export narrowed to one account, meant to be shared outside the console with a party entitled to that account's records. It uses the same encrypted storage and download calls as an export. Its coverage carries only `omittedCount` ; the pool-wide `unclassifiedCount` describes other accounts' activity and is withheld from both the artifact and its metadata.

8.7 Operator sessions

Operators sign in with their wallet. `auditorChallenge` issues a one-use challenge bound to the operator wallet, the request Origin, the auditor service audience, the role `AUDITOR` , cluster, program, genesis hash, pool, a random nonce and an exact expiry (`AUDITOR_CHALLENGE_TTL_MS` , two minutes by default). Issuance is the same for any syntactically valid wallet and pool; whether the wallet is an operator is decided only after `auditorVerify` has checked a strict Ed25519 signature over the reconstructed message. Authorization then requires an enabled `Operator` , an unrevoked `OperatorScope` for that wallet and pool with role `VIEWER` , `ATTESTER` or `MANAGER` , and the pool in the service's configured list.

A verified challenge creates a session: a 32-byte random token `t` set as the `hindsight_auditor_session` cookie with `Path=/v1/auditor` , `HttpOnly` , `SameSite=Lax` , `Secure` under HTTPS and a lifetime of `AUDITOR_SESSION_TTL_MS` (one hour by default). The database stores only

$\text{HMAC}_{k_s}(\text{hindsight:auditor-session:v1:} \parallel t)$ under the service's session key k_s . The CSRF token returned in the response metadata is $\text{HMAC}_{k_s}(\text{hindsight:auditor-csrf:v1:} \parallel t)$, compared in constant time on every mutation, logout included.

Every request must carry an Origin listed in `AUDITOR_ORIGINS` ; the service refuses to start if that list overlaps the hosted application's origins, so a request from the payment app is rejected even when it carries a valid auditor cookie. Every authenticated request rechecks the session's audience, program, genesis hash and origin and re-reads the operator's enablement and pool scope, so a revoked scope takes effect on the next call. Mutating requests are limited to 120 per minute per client network identity, and a client holds at most ten outstanding challenges.

9 Settlement and fees

Hindsight settles in two layers. USDC itself sits in one token account per pool, the vault, whose balance anyone can read. Claims on that USDC are Pedersen commitments: one per confidential account, one per Note in flight. The program moves the pool's tokens only at the two edges where amounts are public, `deposit` and `withdraw` ; everything between them is commitment arithmetic. This chapter describes where value lives, the invariant that ties the two layers together, the caps and fees a pool's policy sets, the buyback that spends the fees, the rent its accounts lock and return, and how USDC base units enter the arithmetic.

9.1 What settles where

Layer	Where	Visible on chain	Changed by
USDC tokens	Vault: classic SPL Token account at <code>["vault", pool]</code> , authority the Pool PDA	Token amount	<code>deposit</code> (in, $a - f$), <code>withdraw</code> (out, a)
Protocol fees	Protocol fee account: the associated token account of the PDA <code>["protocol"]</code>	Token amount	<code>deposit</code> and <code>withdraw</code> (in, f), <code>claim_buyback</code> (out, the claim a)
Bought \$HIND	Protocol token account: the associated token account of the PDA <code>["protocol"]</code> for the \$HIND mint	Token amount	the swap (in, o), <code>complete_buyback</code> (burned, o)
Burned total	<code>BuybackConfig.total_burned</code> , u128	Value	<code>complete_buyback</code>
Principal	<code>Pool.principal</code> , u64	Value	<code>deposit</code> ($+(a - f)$), <code>withdraw</code> ($-a$)
Balances	<code>ConfAccount.balance_commitment</code> and <code>self_record</code>	Commitment and ciphertext; value hidden	Owner only: <code>deposit</code> , <code>transfer</code> , <code>accept</code> , <code>withdraw</code> ; closed at zero by <code>close_account</code>
Value in flight	<code>Note.commitment</code>	Commitment; value hidden	Created by <code>transfer</code> , closed by <code>accept</code>

The vault is a classic SPL Token account; no Token-2022 extension takes part. `deposit` transfers from a token account whose owner is the signer and whose mint is the pool's mint. `withdraw` transfers out, signed by the Pool PDA with seeds `["pool", mint, bump]`, to any token account of the pool's mint other than the vault itself (`InvalidDestination`). No other instruction touches the vault.

A payment settles when its Note is accepted. `transfer` debits the payer's commitment at once and parks the value in the Note; `accept` closes the Note into the payee's commitment. Until then the Note can travel back: the payee may `return_note` it at any time, and the payer may after `return_timeout_slots` slots have passed since `created_slot`. A transfer signature is evidence that a Note exists, not that a payment settled; settled USDC leaves the pool only through `withdraw`.

9.2 Commitment arithmetic

A balance or Note holds a commitment $C = vG + rH$ over Ristretto255, with v the amount in base units, r a scalar blinding, G the Ristretto basepoint and H the solana-zk-sdk Pedersen generator. The program updates commitments with the curve25519 syscalls and never learns v or r ; the owner tracks the opening (v, r) in its client and in its encrypted self-record.

Instruction	Public input	Commitment update	Opening update	Proof
<code>open_account</code>	none	$C_0 = r_0H$	$(0, r_0), r_0$ public	none
<code>deposit</code>	a	$C' = C + (a - f)G$	$(v + a - f, r)$	none
<code>transfer</code>	C_n	$C' = C - C_n$	$(v - v_n, r - r_n)$	U128 over C_n and C'
<code>accept</code>	$C_{n_1}, \dots, C_{n_k}$	$C' = C + \sum_i C_{n_i}$	$(v + \sum_i v_{n_i}, r + \sum_i r_{n_i})$	none
<code>withdraw</code>	a	$C' = C - aG$	$(v - a, r)$	U64 over C'
<code>close_account</code>	none	account closed; requires $C = rH$	$(0, r)$	Schnorr proof of knowledge of r

Scalar arithmetic is modulo the Ristretto group order. The initial blinding is derived, not chosen:

$$r_0 = \text{SHA-256}(\text{hindsight:initial-balance-blinding:v1} \parallel \text{pool} \parallel \text{owner} \parallel \text{epoch}_{\text{le64}} \parallel \text{retry}_{\text{le32}})$$

with the high four bits of the last byte cleared and the counter advanced if the result is zero. The opening of a known-zero balance hides nothing and is public by construction. It keeps an empty balance from being the identity point, which the native proof format cannot represent, so an account that only deposited can still withdraw in full: after `deposit( $a$ )` and `withdraw( $a$ )` the commitment is r_0H , a valid 64-bit range statement for zero. Every Note uses a fresh random blinding from a CSPRNG, which restores hiding as soon as a confidential amount enters the balance.

The two proofs state exactly the conditions that keep value from being created:

$$\begin{aligned} \text{transfer} : & \quad v_n \in [0, 2^{64}) \wedge v - v_n \in [0, 2^{64}), \\ \text{withdraw} : & \quad v - a \in [0, 2^{64}). \end{aligned}$$

`transfer` requires that the immediately preceding top-level instruction be `VerifyBatchedRangeProofU128` on the native ZK ElGamal Proof program with commitments exactly $[C_n, C - C_n]$, bit lengths $[64, 64]$ and every unused context slot zero; `withdraw` requires `VerifyBatchedRangeProofU64` on $[C - aG]$ with bit length 64. Each state change also requires `expected_sequence` to equal the account's `sequence` (`StateSequence`) and increments it, so a proof built against one balance cannot be applied to another.

9.3 The solvency invariant

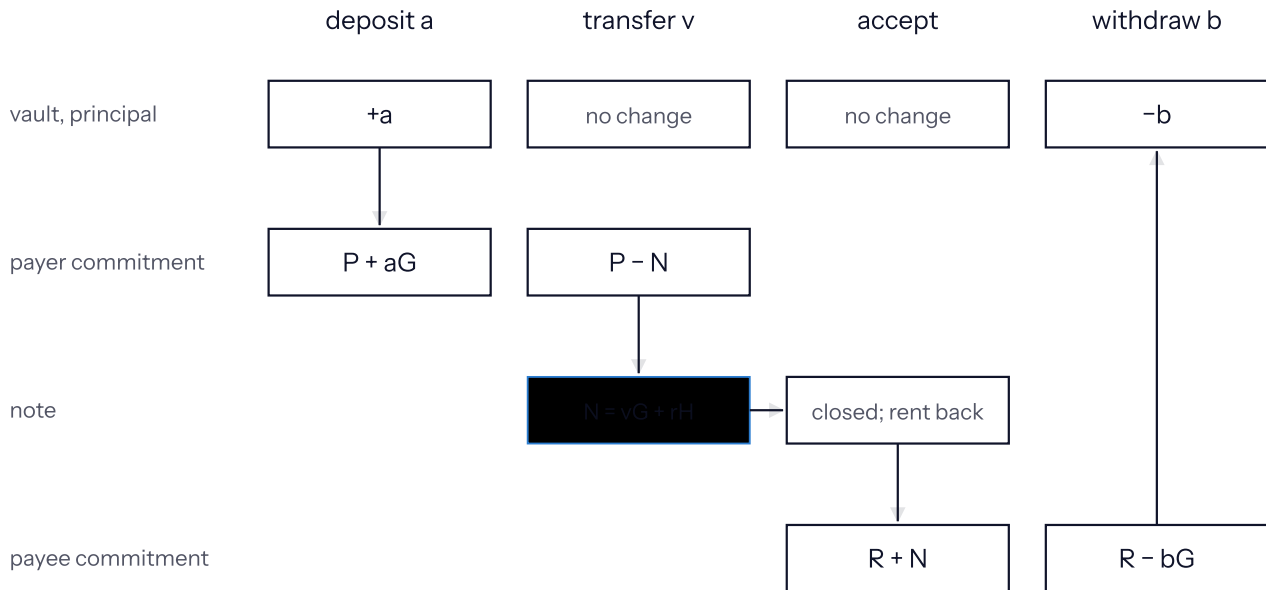
Let $\mathcal{A}$ be the pool's confidential accounts and $\mathcal{N}$ its outstanding Notes. The program maintains

$$\sum_{i \in \mathcal{A}} v_i + \sum_{n \in \mathcal{N}} v_n = \text{Pool.principal} \leq \text{vault.amount}.$$

Instruction	Effect on the left side	<code>principal</code>	<code>vault.amount</code>
<code>deposit(a)</code>	one balance $+a$	$+a$	$+a$
<code>transfer</code>	payer $-v_n$, new Note $+v_n$	unchanged	unchanged
<code>attest</code>	none	unchanged	unchanged
<code>accept</code>	Notes $-\sum v_{n_i}$, payee $+\sum v_{n_i}$	unchanged	unchanged
<code>return_note</code>	Note re-targeted, value unchanged	unchanged	unchanged
<code>withdraw(a)</code>	one balance $-a$	$-a$	$-a$
<code>freeze</code> , <code>thaw</code>	none	unchanged	unchanged
<code>close_account</code>	one balance of 0 leaves $\mathcal{A}$	unchanged	unchanged

`deposit` and `withdraw` update `principal` with checked u64 arithmetic in the same instruction as the token transfer, so both commit or neither does. The equality rests on Pedersen binding and on the soundness of the range proofs and their native verifier: an owner cannot open its commitment to a value larger than the credits it received, and the proofs forbid negative values that would wrap around the group order. The inequality allows for USDC sent to the vault directly, outside `deposit`. Such tokens are public surplus, `vault.amount` $-$ `principal`: they create no confidential credit, do not count against the pool cap, and cannot be withdrawn without a valid balance proof. Because `principal` is a u64 bounded by the pool cap, every honest balance stays below 2^{64} across accept batches.

The local validator flow checks both sides after every transition: the openings of all balances and outstanding Notes sum to `principal`, and the vault backs it.



Money and commitments across one cycle. P and R are the payer's and payee's balance commitments, N the Note's commitment to v . USDC moves only at deposit and withdraw, where the amounts a and b are public; transfer and accept move only commitments, and the vault and principal do not change.

9.4 Deposit and pool caps

Two policy fields, both u64 in USDC base units, bound what a pool holds:

deposit(a) requires $0 < a \leq \text{deposit_cap}$
 and $\text{principal} + (a - f) \leq \text{pool_cap}$,

where f is the ramp fee on a : the fee never enters the vault, so it never counts against the cap.

The first condition fails with `DepositCapExceeded` (a zero deposit included), the second with `PoolCapExceeded`. `init_pool` rejects a policy with **deposit_cap** > **pool_cap** (`InvalidPolicy`). The pool cap is measured against `principal`, not against the vault's token amount, so a donation to the vault cannot block legitimate deposits. Withdrawals are not limited by either field. The runtime configuration publishes `depositCapBaseUnits`, `poolCapBaseUnits` and the current `principalBaseUnits`, read from one RPC observation of the Pool account, so a client can show remaining capacity before building a deposit.

9.5 Fees

A pool's fee terms are two policy fields, set at `init_pool` and stored in the Pool account beside its caps:

Offset in Pool	Length	Field	Unit
176	1	attest_required	bool
177	8	return_timeout_slots	slots
185	8	deposit_cap	USDC base units
193	8	pool_cap	USDC base units
201	2	ramp_fee_bps	basis points, u16
203	8	flat_transfer_fee	USDC base units, u64
211	8	auditor_rotation_interval_seconds	seconds
219	2	admin_multisig_threshold	signers, u16
221	8	admin_timelock_seconds	seconds

The placement follows from what is public. A transfer's amount is hidden, so a fee proportional to it would publish the amount: a fee $f = v \cdot b/10^4$ visible on chain reveals $v = 10^4 f/b$. Fees therefore attach where amounts are already public, or do not depend on the amount at all.

- **Ramp fee.** `ramp_fee_bps` applies to deposits and withdrawals, whose amounts a appear in instruction data. At b basis points the fee is $f = \lfloor a \cdot b/10^4 \rfloor$ base units, rounded down so the participant keeps the rounding, and it discloses nothing beyond a , which is public already. Pools are created at $b = 5$; the program accepts $0 \leq b \leq 100$.
- **Transfer fee.** `flat_transfer_fee` is a fixed amount f_t per Note. Every transfer in a pool pays the same f_t , so the fee is independent of v and carries no information about it.

The flat transfer fee is zero, and `PoolPolicyConfig::resolve` refuses any other value with `FeeCustodyNotConfigured`: a Note carries exactly the value its payer committed. The ramp fee moves with the tokens:

	Owner's tokens	Vault	Protocol fee account	Commitment	principal
<code>deposit(a)</code>	$-a$	$+(a - f)$	$+f$	$+(a - f)G$	$+(a - f)$
<code>withdraw(a)</code>	$+(a - f)$	$-a$	$+f$	$-aG$	$-a$

At 5 basis points a deposit of 10,000 USDC credits 9,995 USDC and a withdrawal of 1,000 USDC pays out 999.50 USDC. Fees never enter the vault, so the solvency invariant below holds unchanged. Clients learn

the fee from the pool's policy on chain and apply it to the balance opening: after a deposit the opening is $(v + a - f, r)$.

9.5.1 The protocol fee account and \$HIND

The protocol's share is all of the ramp fee. It is paid in the same instruction that charges it, into the associated token account, for the pool's mint, of the program address with seeds `["protocol"]`. No key signs for that address, and `deposit` and `withdraw` require the account by its derived address, so every fee lands there and nowhere else.

The fees collected there have one use: buying \$HIND, the project's token, on the market and burning it. \$HIND has a fixed supply of 1,000,000,000, all of it issued in one public launch on equal terms, with no presale, no team or investor allocation, no airdrop, no reserved pool and no vesting. Payments never need it: a deposit, a transfer, an accept and a withdrawal move only USDC, so the fee is the one link between use of the pool and the token, and the token's supply only shrinks as the pool is used.

The program buys and burns in one transaction. Its terms sit in the `BuybackConfig`, a global account at `["buyback"]` that the program's upgrade authority creates once with `configure_buyback` and that no instruction edits afterwards: the \$HIND mint and its token program, one keeper key, and a minimum-output ratio n/d counted in \$HIND base units per USDC base unit. Only that keeper can crank, and every buyback transaction runs the same three steps:

1. **Claim.** `claim_buyback(amount, minimum_output, expected_sequence)` checks the keeper's signature, every account against the configuration, the shape of the transaction, and that `expected_sequence` equals the configuration's `sequence`. For a claim of a base units it requires $0 < a \leq$ the fee account's balance and a minimum output m with $m \geq \lceil a \cdot n/d \rceil$ and $m > 0$. It then moves exactly a from the protocol fee account to the keeper's associated USDC account, signed by the protocol PDA, and records the keeper account's balance before the transfer and the protocol token account's balance before the swap.
2. **Swap.** One or more swap instructions, Jupiter's on mainnet-beta, spend the claimed USDC from the keeper's account and deliver \$HIND to the protocol token account: the associated token account of the protocol PDA for the \$HIND mint, under that mint's own token program. The keeper never holds the bought tokens.
3. **Complete.** `complete_buyback`, the last instruction, requires the keeper's USDC balance to be back at its value before the claim, so the whole claim was spent, and the protocol token account to have received an amount $o \geq m$ since the claim. It burns all of o , signed by the protocol PDA, adds o to `total_burned` and increments `sequence`.

Both instructions read the transaction through the `instructions` sysvar and require exactly one claim and one completion at top level, with identical account lists, at least one instruction between them and nothing after the completion. A failed check, swap or burn rolls back the whole transaction, the claimed fees included, so the fee account pays out only in a transaction that burns at least the minimum. No other instruction moves tokens out of the fee account: there is no withdrawal, no sweep and no other recipient.

	claim_buyback	swap	complete_buyback
protocol fee account	-a	no change	no change
keeper USDC account	+a	-a	back to its start
protocol token account	no change	+o	-o, burned
BuybackConfig	pending: a, m start balances	no change	total_burned + o sequence + 1

One buyback transaction. Columns are its instructions in order, rows the accounts they change. `claim_buyback` moves a USDC from the protocol fee account to the keeper's account and records the claim; the `swap` spends all of a and delivers o \$HIND to the protocol token account; `complete_buyback` checks that the keeper's balance is back at its start and that o is at least m , burns all of o and adds it to `total_burned`. A failure at any step undoes every column.

	Protocol fee account	Keeper's USDC account	Protocol token account	\$HIND supply	total_burned
claim_buyback	$-a$	$+a$			
swap		$-a$	$+o$		
complete_buyback			$-o$	$-o$	$+o$
Whole transaction	$-a$	0	0	$-o$	$+o$

Since $o \geq m \geq \lceil a \cdot n/d \rceil$, every USDC base unit claimed from the fee account burns at least n/d base units of \$HIND. The keeper's side is the buyback crank, `npm run buyback`, a scheduled service described in the Infrastructure chapter: it claims the fee account's whole balance or a set amount, takes an exact-input Jupiter quote whose minimum output meets the floor, and passes that minimum as `minimum_output`. On localnet and devnet a co-signing stand-in market takes Jupiter's place and faces the same program checks.

9.6 Rent

Every account the program creates holds a rent-exempt deposit in SOL. For an allocation of s bytes the deposit is

$$\text{deposit}(s) = (s + 128) \times \text{the cluster's lamports per byte}$$

read live from the Rent sysvar, and every participant account's deposit returns to its payer when the account closes. The program's actual allocations, which include the 8-byte Anchor discriminator:

Account	Allocation	Paid by	Returned
Pool	238 B	<code>init_pool</code> payer	not closed
Vault	165 B	<code>init_pool</code> payer	not closed
KeyRecord	1,265 B	owner, at <code>register_key</code>	to the owner, by <code>close_key_record</code>
ConfAccount	190 B	owner, at <code>open_account</code>	to the owner, by <code>close_account</code>
Note	235 B	sender, at <code>transfer</code>	to the original sender, when <code>accept</code> closes it
BuybackConfig	209 B	payer, at <code>configure_buyback</code>	not closed
RentRecoveryRegistry	21 B, plus 32 B per protected auditor key	payer, at <code>initialize_rent_recovery</code> ; growth by the <code>init_pool</code> payer	not closed

A payer funds each Note only while it is outstanding. `accept` takes its remaining accounts as ordered pairs (Note, rent destination) and requires the destination to equal `note.sender` (`InvalidRentDestination`), so one batch can close Notes from many payers and refund each of them. The rule also holds for returned Notes: when a payer accepts its own returned Note, the rent comes back to the same wallet that paid it.

A participant funds its `KeyRecord` and `ConfAccount` when it onboards and takes both deposits back when it leaves. After a full withdrawal its balance commitment is rH , and `close_account`, signed by the owner, checks a Schnorr proof of knowledge of r , which shows the balance is zero without revealing r , and moves every lamport of the `ConfAccount` to the owner. `close_key_record`, also owner-signed, moves every lamport of the `KeyRecord` to the owner, unless a pool names that `KeyRecord` as its auditor key (`AuditorKeyInUse`). Neither touches the vault, `principal` or a Note: Notes still open keep their rent

destinations, and the owner can reopen its account at the same address to accept them. Pool, vault, the global `BuybackConfig` and the global `RentRecoveryRegistry` have no closing instruction and keep their deposits for the life of the deployment.

Transaction fees are separate and also in SOL. For deposits, transfers, accepts and withdrawals the account owner is sole signer and fee payer; for attestations the auditor key pays, and for the buyback the keeper.

9.7 USDC decimals and rounding

`init_pool` requires the mint to have six decimals (`InvalidMint`), and the runtime configuration re-reads the mint account and reports its decimals. On mainnet-beta the deployment configuration selects Circle's native USDC mint `EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v` ; on devnet, Circle's devnet mint. Every amount in the protocol is an unsigned 64-bit integer of base units,

$$1 \text{ USDC} = 10^6 \text{ base units}, \quad 0 \leq v \leq 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615,$$

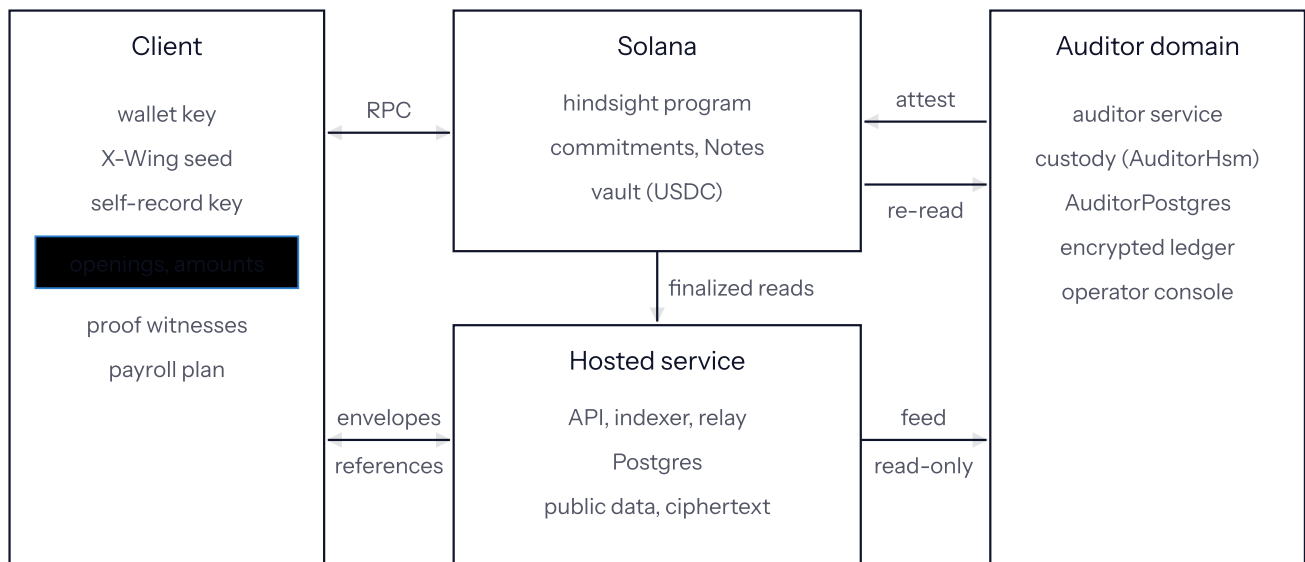
with no 48-bit cap and no split into halves, because no amount is ever recovered by solving a discrete logarithm. The program maps a public amount a to the scalar whose low eight bytes are a in little-endian order and computes aG ; the self-record and the envelope plaintext carry amounts as eight little-endian bytes.

No step rounds. Deposits, withdrawals, transfers and accepts are exact integer operations on base units, the buyback claims and burns exact amounts too, and commitment arithmetic is exact in the scalar field because honest values stay below 2^{64} , far below the group order. The SDK holds amounts as `bigint` , and the API carries them as canonical decimal strings, never as floating-point numbers. The payroll CSV parser makes the unit explicit: with `amountUnit: "usdc"` it accepts at most six decimal places and converts $w.f$ to $w \cdot 10^6 + f$ by integer arithmetic, rejecting any value with more precision instead of rounding it; with `amountUnit: "base_units"` it accepts exact u64 integers.

10 Security design

Hindsight's security design starts from what each party can see and do, and assigns every guarantee to the mechanism that provides it. Four adversaries shape it: an archive of the chain recorded today and read later with a quantum computer, a sender who builds a malicious transfer, a relay or indexer that is curious about what it stores, and an observer on the network between clients and services. This chapter takes each in turn, then describes the mechanisms that run across all of them: domain separation, wallet sessions, key custody, and the pool's administrative controls.

Adversary	Capability	Protection
Chain archive read with a quantum computer	Every byte ever posted on chain; Shor's algorithm against discrete logarithms	Perfectly hiding commitments; no decryption handles; openings inside ML-KEM-768 envelopes; AES-256
Malicious sender	Arbitrary transfers from its own account, arbitrary envelope bytes	Range proofs bound to the exact transition; credit only on <code>accept</code> after the payee's and, by policy, the auditor's check
Curious relay or indexer	Everything the hosted service stores and serves	It stores only public chain data and opaque ciphertext; clients and the auditor verify every byte against the chain
Network observer	Traffic between browsers, services and RPC	Plaintext amounts never leave the client; sessions bound to origin, audience and chain; HTTPS-only cookies



Trust boundaries. Plaintext amounts and openings exist only in the client and in the auditor domain. The chain holds commitments and ciphertext; the hosted service holds public chain data and opaque envelopes, and the auditor reads it through a read-only feed and re-reads every fact from the chain before it opens or signs anything.

10.1 A chain archive read with a quantum computer

Everything posted to Solana is copied by validators and archival RPC providers, so an adversary can record it today at the cost of storage and attack it whenever a cryptographically relevant quantum computer exists. Hindsight's on-chain footprint is chosen so that such an archive contains no amount that a quantum computer can recover.

On chain	Protection	Against Shor
Balance and Note commitments $C = vG + rH$	Perfect hiding	Unaffected: no computation recovers v
Range proofs (Bulletproofs)	Zero-knowledge	The design treats them as revealing nothing beyond the commitments
Payee and auditor envelopes, 1,208 bytes each	X-Wing (X25519 + ML-KEM-768), AES-256-GCM	ML-KEM-768 holds; AES-256 retains a 128-bit margin against Grover's algorithm
Self-records, 68 bytes each	AES-256-GCM under a client key never posted	As above
Decryption handles	None exist	Nothing to attack

Commitments. A Pedersen commitment with a uniformly random blinding is perfectly hiding: for every pair of amounts v, v' and every group element X ,

$$\Pr_r [vG + rH = X] = \Pr_r [v'G + rH = X] = \frac{1}{\ell},$$

where ℓ is the order of the Ristretto group. A commitment is a uniformly random group element whatever the amount, and no amount of computation, classical or quantum, extracts v from it. The program never stores anything else derived from v .

No decryption handles. Token-2022 confidential transfers attach ElGamal handles $D = rP$ to the same commitment, with public keys stored on chain; one discrete logarithm per key decrypts every amount ever sent to it, and the mint's auditor key exposes every amount of the mint. Hindsight stores zero handles. Openings travel only inside envelopes, so a discrete-logarithm oracle applied to the archive learns nothing about amounts.

Envelopes. Each opening reaches the payee and the auditor inside an X-Wing envelope: a 1,120-byte hybrid KEM ciphertext (1,088 bytes of ML-KEM-768, 32 of X25519) followed by 88 bytes of AES-256-GCM over $v \parallel r \parallel \text{memo}$. X-Wing hashes both component secrets, with the X25519 ciphertext and public key, into one 32-byte shared secret, so the envelope stays confidential as long as either ML-KEM-768 (FIPS 203) or X25519 holds. Against a quantum adversary, ML-KEM-768 carries the guarantee and X25519 is the classical hedge.

Key rules. Three rules keep this footprint sound in practice. Post-quantum keys come from fresh entropy, never from a wallet signature: Shor's algorithm applied to an Ed25519 wallet key recovers its scalar, and a decryption key derived from that key's signatures would fall with it. Every Pedersen opening of a Note is sampled from a CSPRNG and never reused. Envelopes have one fixed size and memos are padded to 32 bytes, so the length of a ciphertext says nothing about its content. The only public opening is the initial blinding r_0 of a known-zero balance, derived from a hash of pool, owner and epoch; it hides nothing because there is nothing to hide, and the first fresh Note opening added to or subtracted from the balance restores full hiding.

What the archive holds in the clear. Deposit and withdrawal amounts are public by construction, since USDC crosses the vault boundary there, and so are senders, recipients and timing. Pedersen binding, Bulletproof soundness and Ed25519 do their work when each transaction is verified; what they protect is settled at that moment, and no amount in the archive depends on them afterwards.

10.2 A malicious sender

A sender controls its own account and every byte of its transactions. The program constrains what those bytes can achieve.

It cannot spend more than it holds. `transfer` accepts a commitment C_n only together with a native proof that $v_n \in [0, 2^{64})$ and $v - v_n \in [0, 2^{64})$ for the sender's current balance. `verify_range_context` binds that proof to the exact transition:

- the proof instruction is the immediately preceding top-level instruction, owned by the ZK ElGamal Proof program (`InvalidProofProgram`), of type `VerifyBatchedRangeProofU128` (`InvalidProofType`), with no accounts, so no context-state account substitutes for verification (`ProofContextStateUnsupported`);
- `transfer` itself is a top-level Hindsight instruction, not a CPI from a wrapper (`ProofCpiUnsupported`);
- the context's commitments are exactly $[C_n, C - C_n]$ with bit lengths $[64, 64]$ and every unused slot zero (`ProofContextMismatch`), and neither commitment is the identity (`IdentityProofContextUnsupported`);
- C_n is a valid Ristretto point (`InvalidCommitment`).

If the native proof fails, the transaction aborts before `transfer` runs. `withdraw` applies the same rules with `VerifyBatchedRangeProofU64` over $C - aG$.

It cannot replay or race. Every balance change requires `expected_sequence` to equal the account's `sequence` (`StaleSequence`) and increments it. The Note PDA is derived from `["note", pool, sender, sequence]`, so a second transfer at the same sequence cannot create a second Note. Only the owner changes its own commitment: incoming value waits in Notes, so no third party's transaction can invalidate the proof a sender has just built.

It cannot choose the auditor's key. The auditor `KeyRecord` in `transfer` is derived from the pool's `auditor` and `auditor_kem_epoch` and checked against both (`InvalidAuthority`); the payee `KeyRecord` must belong to the recipient account's owner and match its `key_epoch` (`InvalidRecipient` , `InvalidKeyEpoch`). Envelopes must be exactly 1,208 bytes (`InvalidEnvelopeLength`), and the Note stores `SHA-256(payee || auditor)`, which fixes the bytes both readers will open.

What it writes into an envelope is its own loss. The next section explains why the program needs no proof of an envelope's contents.

10.3 Why envelopes need no proof

Proving inside a SNARK that an ML-KEM envelope encrypts the opening of C_n would cost an estimated 10^5 or more constraints of client proving for every transfer. Hindsight replaces that proof with a rule about credit: no account is ever credited with a commitment its owner has not opened. A balance increases only through `accept`, which requires:

1. the signature of the Note's current recipient, who is the owner of the receiving account (`InvalidRecipient`);
2. a client that has decrypted the payee envelope under its AAD and checked $vG + rH = C_n$ before building the instruction;
3. when `attest_required` is set, `attested = true`, meaning the auditor has performed the same check on its own envelope and recorded it on chain.

Consider a sender who posts a Note whose envelopes do not open, or open to a value that does not match. The payee's client fails the check and calls `return_note`, which re-targets the Note to its sender. The auditor's service classifies the Note `FAILED_TO_OPEN` or `OPENING_MISMATCH` and never attests it. The payee's balance never changed. The sender's balance was debited by C_n when it sent the Note, and that value comes back only when the sender accepts the returned Note with the opening it generated itself; the Note's rent stays locked until then. A garbage envelope therefore costs only the party that wrote it, and it is returned.

The case of two envelopes that open to different amounts reduces to the same rule. If the payee's opening and the auditor's opening both reproduce C_n , binding forces them to be equal; if either fails, the Note is not credited. The auditor cannot be shown one amount while the payee is credited another, under the same binding assumption that already protects solvency.

A sender who withholds envelopes from the relay gains nothing either: the envelopes are in the `transfer` instruction data on chain, and the indexer stores them from there. A Note that is never accepted can be recovered by its sender once `return_timeout_slots` have passed after `created_slot`; until then the payee alone decides.

10.4 A curious relay or indexer

The hosted service runs the indexer, the envelope relay, the read API, wallet sessions and notifications. The design gives it nothing confidential to look at and nothing it could alter undetected.

What it stores. The hosted schema holds public chain metadata, opaque envelope pairs, the already-public deposit and withdrawal amounts, hashed challenges and sessions, payroll routing references and notifications. It has no column for a confidential amount, an opening, a memo, a payroll CSV or a private key. Indexed instruction data keeps only known Hindsight instruction prefixes and opaque ciphertext; proof bytes, memo instructions, program logs and trailing calldata are dropped. Request logs carry a request identifier, route, status and duration, never headers, bodies, signatures or keys.

What it cannot alter. Every reader verifies what the service hands it:

- A payee checks the envelope pair it downloads against the Note's on-chain `envelope_hash` before opening it, and the AEAD check rejects any envelope altered after hashing.
- Transaction builders take Pool, `ConfAccount`, `KeyRecord` and Note bytes decoded from chain accounts, and the payroll batcher reads them through a payer-configured RPC endpoint, checking program ownership, discriminators, sizes, PDAs and every X-Wing public key byte before it seals anything. A hosted API is not a trust anchor for the program ID, the RPC endpoint, the auditor or any key.
- The auditor uses the indexer only as a list of addresses to look at, through a SELECT-only database role, and re-reads Note, Pool and `KeyRecord` by RPC before opening and again before signing.

Withholding gains it nothing. A relay that refuses to serve envelopes cannot redirect funds: the envelopes are recoverable from the transaction, and an unaccepted Note stays recoverable by its sender after the timeout. Uploads made before the Note exists are held as bounded, expiring pending records

and are served only after a finalized Note confirms that the uploader is its sender and that the bytes match its hash.

What the RPC relay sees. On keyed clusters the API forwards browser JSON-RPC calls to the provider, so the provider key stays on the server. It forwards only an allowlist of read methods plus `sendTransaction` and `simulateTransaction`, accepts a `sendTransaction` only if it decodes to a transaction of at most 4,096 bytes that invokes the Hindsight program and otherwise only the compute-budget, Lighthouse, ZK ElGamal Proof and associated-token programs, and logs nothing but the method name. Everything it handles is public or about to be.

10.5 Network observers

Nothing a client sends to a Hindsight service contains a plaintext amount, opening, memo or key. Signed transactions are public once submitted; envelope uploads are ciphertext that will appear on chain; reads and notifications carry public references. Range-proof witnesses never leave the browser: proofs are generated in a Worker from a hash-pinned WASM module.

Deployed services accept only HTTPS browser origins. Session cookies are `HttpOnly`, `SameSite=Lax` and `Secure` whenever the service audience is HTTPS. CORS admits only the configured origins, every protected request (GET and SSE included) must carry an allowed `Origin` equal to the one its session was created from, and every mutation carries a CSRF token. Client identity for rate limits is the socket peer unless both `TRUSTED_PROXY_CIDRS` and `CLIENT_IP_HEADER` are configured, in which case forwarding headers are read right to left only while each hop is a trusted proxy; IPv6 clients are grouped by /64, and no raw address is written to a database or log.

10.6 Domain separation

Every key derivation, MAC and associated-data string in Hindsight starts with a versioned label naming its purpose, so a value produced for one purpose never verifies for another.

Label	Primitive	Binds
<code>hindsight:envelope:v0:</code>	AES-256-GCM AAD	Role, pool, sender, original recipient, sequence, key epoch, commitment
<code>hindsight:envelope:aes256gcm:key-nonce:v0:</code>	HKDF-SHA256 info	Envelope key and nonce to the AAD
<code>hindsight:self-record:v1:</code>	AES-256-GCM AAD	Pool, account, sequence, balance commitment
<code>hindsight:initial-balance-blinding:v1</code>	SHA-256	Pool, owner, key epoch
<code>hindsight:close-account:v1</code>	SHA-256	Zero-balance proof challenge to program, pool, account, owner, sequence, close nonce, balance commitment and nonce point
<code>hindsight:session-challenge-ticket:v1</code>	HMAC-SHA256	Challenge body to wallet, origin, audience, role, cluster, program, genesis
<code>hindsight:challenge-client:v1</code>	HMAC-SHA256	Quota identity to service audience and client network
<code>hindsight:session-token:v1:</code>	SHA-256	Stored session token hash
<code>hindsight:csrf:v1:</code>	HMAC-SHA256 keyed by the session token	CSRF token to service audience and role
<code>hindsight:relay:key:v1: ,</code> <code>hindsight:relay:body:v1:</code>	HMAC-SHA256	Relay retry key and request digest
<code>hindsight:payroll-progress:key:v1: ,</code> <code>hindsight:payroll-progress:body:v1:</code>	HMAC-SHA256	Payroll progress retry key and request digest
<code>hindsight:access-request:key:v1: ,</code> <code>hindsight:access-request:body:v1:</code>	HMAC-SHA256	Access-request retry key and request digest
<code>hindsight:payroll-storage:v1:</code>	AES-256-GCM AAD	Plan or journal to its local plan ID and digest

Label	Primitive	Binds
<code>hindsight:auditor-storage:v1:</code>	AES-256-GCM AAD	Ledger or export ciphertext to its row ID
<code>hindsight:auditor-session:v1:</code> , <code>hindsight:auditor-csrf:v1:</code>	HMAC-SHA256	Auditor session hash and CSRF token

10.6.1 Envelope binding

The envelope AAD is 167 bytes:

Offset	Length	Field
0	22	ASCII <code>hindsight:envelope:v0:</code>
22	1	Role: 0 payee, 1 auditor
23	32	Pool
55	32	Sender wallet
87	32	Recipient: the original payee wallet, in both envelopes
119	8	Sender sequence, u64 little-endian
127	8	Recipient's key epoch, u64 little-endian
135	32	Note commitment C_n

The AES key and nonce come from the KEM shared secret ss , the recipient's public key pk and the KEM ciphertext ct :

$$\begin{aligned}
 k \parallel \nu &= \text{HKDF-SHA256}(\text{IKM} = ss, \text{salt} = \text{SHA-256}(pk \parallel ct), \\
 &\quad \text{info} = \text{hindsight:envelope:aes256gcm:key-nonce:v0:} \\
 &\quad \parallel \text{aad}, L = 44),
 \end{aligned}$$

with k the first 32 bytes and ν the last 12. Each envelope encapsulates afresh, so each key and nonce pair is used once, and no nonce is transmitted. Because the AAD names the role, an auditor envelope cannot be presented as a payee envelope; because it names pool, sender, sequence and commitment, an envelope cannot be moved to another Note. The self-record applies the same idea to balances: its 129-byte AAD binds pool, account, sequence and the balance commitment, so a record copied to another account or another sequence fails to open, and opening checks the recovered value against the commitment before returning it.

10.6.2 Scoped relay keys

Retry keys protect writes against duplicate submission without letting one client probe another's. The relay stores $\text{HMAC}_K(\text{hindsight}:\text{relay}:\text{key}:\text{v1}:\parallel \text{key})$ and $\text{HMAC}_K(\text{hindsight}:\text{relay}:\text{body}:\text{v1}:\parallel [\text{note}, \text{hash}])$ under the API's `RELAY_HMAC_KEY`, in rows scoped to the uploading wallet, cluster, program and genesis hash. An identical retry replays the stored result; the same key with a different body fails with `IDEMPOTENCY_CONFLICT`. Payroll progress uses the same construction under its own labels; access requests use their own key, `ACCESS_REQUEST_HMAC_KEY`. Raw retry keys are never stored.

10.7 Wallet sessions

A wallet session authorizes reads and uploads scoped to one wallet. It never derives a key. The challenge message says so in its second line.

`authChallenge` returns an opaque 96-character ticket, the base64url encoding of 72 bytes:

Offset	Length	Field
0	1	Version, 1
1	6	Issue time, milliseconds, big-endian
7	4	Lifetime, milliseconds, big-endian
11	1	Flags: role (wallet, auditor, admin) and pool presence
12	32	Pool, or zero
44	12	Random nonce
56	16	Truncated HMAC-SHA256 tag

The tag covers the first 56 bytes together with the wallet, origin, service audience, role, cluster, program and genesis hash, so the server can reconstruct and authenticate a challenge without a database row. The wallet signs a human-readable message listing the same fields:

Hindsight session authorization v1

This signature authorizes a session only. Never derive encryption keys from it.

Wallet: <wallet address>

Origin: <browser origin>

Audience: <service audience>

Role: <WALLET, AUDITOR or ADMIN>

Cluster: <cluster>

Program: <program ID>

Genesis: <genesis hash>

Pool: <pool address or →>

Challenge: <ticket>

Nonce: <nonce hash>

Issued at: <ISO 8601 time>

Expires at: <ISO 8601 time>

`authVerify` checks the signature with strict Ed25519: canonical encodings, public key and nonce point neither of small order nor outside the prime-order subgroup, ZIP-215 relaxations disabled. It checks the ticket's tag and expiry (issue times up to five seconds in the future are tolerated for replica clock skew; expiry is exact), and then consumes the challenge. A unique ticket hash on the session row makes each challenge one-use, even after the challenge cache evicts it or the session expires.

The session token is 32 random bytes; the database stores

SHA-256(hindsight:session-token:v1: || t). The token travels as the

`hindsight_wallet_session` cookie with `Path=/v1`, `HttpOnly`, `SameSite=Lax` and `Secure`, for

`SESSION_TTL_SECONDS` (one hour by default, at most one day). The CSRF token is

HMAC_t(hindsight:csrf:v1: || audience || : || role), returned in response metadata and required

as `X-CSRF-Token` on every mutation, compared in constant time. Expiry and revocation are checked on every protected read and on every poll of a notification stream, so a stream ends when its session does.

Each client network may hold ten outstanding challenges; the challenge cache keeps at most

`AUTH_CHALLENGE_MAX_OUTSTANDING` rows (1,000 by default) and evicts the oldest to admit a new client rather than refusing service.

The auditor runs the same construction with its own database, HMAC key, cookie name and path, and audience, and authorizes operators only after the signature is verified.

10.8 Key custody

Key	Held by	Purpose
Wallet key, Ed25519	User's wallet, browser-injected or mobile deep link, no USB transport	Signs transactions and session challenges
X-Wing seed, 32 bytes	Client durable storage, from fresh entropy	Opens payee envelopes
Self-record key, AES-256	Client WebCrypto key, exported only into the user's passphrase-encrypted backup	Seals and opens the balance opening
Payroll storage key, AES-256	Payer-controlled	Seals payroll plans and journals
Range-proof witnesses	Client Worker memory	Never stored or sent
Attestation seed and X-Wing epoch seeds	Auditor custody (AuditorHsm)	Sign attest ; open auditor envelopes
Ledger key, session HMAC key	Auditor service only, required to differ	Encrypt ledger and exports; sessions
Relay and access-request HMAC keys	API service only	Retry keys and digests
Keeper key, Ed25519	Buyback crank only, as BUYBACK_KEEPER_KEYPAIR_JSON or a private keypair file	Signs and pays the buyback transaction: claim_buyback , complete_buyback
Upgrade, admin and freeze authorities	Pool governance	init_pool , initialize_rent_recovery , configure_buyback , freeze , thaw

Client keys never reach a Hindsight service. The SDK generates the X-Wing seed and self-record key on the device and offers no path that uploads them; the hosted API never generates proofs or opens an envelope or self-record.

The auditor's keys live in its own process and its own storage. Its secrets are configured only on the auditor service and never on the API or the web application. Its private database is separate from the hosted one, and its read access to the hosted database uses a role with SELECT on nine chain tables only. At startup it verifies its private migrations before it loads custody, contacts the chain or serves a request. Its ledger and exports are encrypted at rest under a key that never enters either database, and the configured retention period removes private ciphertext while keeping public attestation metadata. The software custody adapter derives every key from its seed, checks each derived public key against the registered one, wipes seed and key buffers when they are no longer needed, and sits behind the `AuditorHsm` interface (`open` and `sign`) that a hardware-backed adapter replaces as a whole.

10.9 The admin authority, freeze and policy controls

Bootstrapping. `init_pool` must be signed by the program's upgrade authority, checked through the program's actual ProgramData account (`InvalidBootstrapAuthority`), so no other signer can claim the canonical pool for a mint. The Pool PDA is `["pool", mint]`, one pool per mint per deployment. The mint must have six decimals (`InvalidMint`), the auditor and freeze authority must be set (`InvalidAuthority`), and every policy field must be supplied explicitly (`ConfigurationRequired`); the program never fills in a default.

Policy. The policy is fixed at `init_pool` and no instruction rewrites it. Its values are checked once: `deposit_cap` at most `pool_cap`, `return_timeout_slots` at most $2^{32} - 1$, a positive rotation interval and multisig threshold (`InvalidPolicy`), and fee fields as described in the settlement chapter.

Buyback. The protocol fee account's token owner is the PDA `["protocol"]`, and the program signs for it only in `claim_buyback` and `complete_buyback`. `configure_buyback` takes the upgrade authority's signature through the same ProgramData check as `init_pool` (`InvalidBootstrapAuthority`) and creates the BuybackConfig once; no instruction edits it. Only the configured keeper can claim (`InvalidAuthority`), a claim's minimum output is at least the floor fixed at configuration (`BuybackOutputTooSmall`), and a claim takes effect only in a transaction whose final `complete_buyback` finds the whole claim spent and burns everything bought (`InvalidBuybackTransaction`, `BuybackInputNotSpent`). The bought tokens arrive in the protocol's own token account, and no instruction withdraws or sweeps either account.

Rent recovery. `initialize_rent_recovery` takes the upgrade authority's signature through the same ProgramData check (`InvalidBootstrapAuthority`), creates the RentRecoveryRegistry once, and accepts as remaining accounts only program-owned Pools at their canonical addresses (`InvalidRentRecoveryRegistry`). Every pool's auditor KeyRecord is listed there, by that bootstrap or by `init_pool`, and `close_key_record` refuses a listed key (`AuditorKeyInUse`). `close_account` and `close_key_record` require the owner's signature and pay only that owner; a zero-balance proof binds the account, its owner, its sequence and the global close nonce, which every close consumes (`InvalidZeroBalanceProof`, `StaleRentRecoveryNonce`).

Authorities. The Pool stores `admin_authority` and `freeze_authority` as signer addresses intended to be controlled by an external multisig with a timelock; `admin_multisig_threshold` and

`admin_timelock_seconds` record that arrangement's terms. The program checks the signature of the configured address; the threshold and delay are enforced by the multisig program that holds it.

Freeze. The freeze authority signs `freeze` and `thaw`, which set and clear `ConfAccount.frozen`. A frozen account cannot deposit, transfer, accept, withdraw or close, and no transfer can target it (`AccountFrozen`). Freezing changes no balance, reads no amount and moves no USDC. `return_note` stays available, because a return changes no balance: a payee whose account is frozen can still send a Note back, and a payer can recover a Note addressed to a frozen account once the return timeout has passed.

11 Infrastructure

The on-chain program holds every balance and decides every payment; the services around it make the chain usable from a browser without being trusted with anything confidential. Four services run off chain: `web`, the application the user's browser loads; `api`, which indexes the program, relays envelopes, authenticates wallets, streams notifications and relays RPC calls; `auditor`, which opens auditor envelopes and attests; and the buyback crank, which spends the protocol fees on \$HIND for the program to burn. The `api` and `auditor` each own one PostgreSQL database; the crank needs none. This chapter describes what each service does, what it stores, and how the deployment is assembled.

Service	Runs	Database	Holds
<code>web</code>	The payment app; the auditor console is built from the same <code>web/</code> source	none	Nothing; keys and openings live in the user's browser
<code>api</code>	HTTP API, in-process indexer, envelope relay, wallet sessions, notifications, payroll progress, access requests, RPC relay	<code>Postgres</code>	Public chain projections, opaque envelopes, hashed sessions, notifications, contact requests
<code>auditor</code>	Attestation worker and console API	<code>AuditorPostgres</code> , plus read-only access to <code>Postgres</code>	Jobs, encrypted ledger entries, receipts, encrypted exports, operator scopes
Buyback crank	<code>npm run buyback</code> on a schedule: one claim, swap and burn transaction per run	none	Nothing; the keeper key is a secret variable

11.1 The indexer

The indexer turns the program's accounts and transactions into projections the API can serve. It runs inside the `api` process, signs nothing and cannot decrypt an envelope or a self-record.

Finalized projections. Each cycle reads the program's accounts at finalized commitment. Pools, `ConfAccount` s, `KeyRecord` s and Notes are decoded from the deployed account layout, including each pool's current policy and each account's real 64-bit sequence. A pool's public principal and its vault's token balance come from one `getMultipleAccounts` response, so both values describe the same bank. Transactions are fetched by signature, up to `INDEXER_MAX_SIGNATURES_PER_SYNC` (2,000) per cycle, every `INDEXER_POLL_INTERVAL_MS` (3,000 ms). Confirmed observations, where the RPC supplies them, are recorded as hints: they let the auditor start work early, but they never credit a balance or change a finalized Note.

Atomic cycles. The checkpoint, chain events, envelopes, public deposit and withdrawal history and projections commit in one PostgreSQL transaction under an advisory lock scoped to the program. Expected genesis hash and monotonic slots fence a delayed worker out of the database, and transactions within a slot are ordered by the canonical block's signature order. A confirmed observation is declared orphaned only when an available finalized block shows it absent; missing RPC history proves nothing.

Coverage. RPC providers prune history, and the indexer reports what it could not see instead of inventing it. Pruned history, unavailable transactions, backfill limits, snapshot divergence and unknown CPI outcomes become durable `IndexerGap` rows (`HISTORY_PRUNED` , `TRANSACTION_UNAVAILABLE` , `BACKFILL_LIMIT` , `SNAPSHOT_DIVERGENCE` , `CPI_OUTCOME_UNAVAILABLE`). A snapshot-only `KeyRecord` or Note keeps a null registration or creation signature and timestamp. A Note missing from a newer snapshot is marked `onChainPresent = false` and is never called accepted without an accepting instruction. Notes and envelopes indexed before pruning remain available. `GET /v1/indexer/coverage` returns:

Field	Content
<code>status</code>	<code>healthy</code> , <code>partial</code> or <code>unavailable</code>
<code>programId</code> , <code>genesisHash</code>	Identity of the indexed program and cluster
<code>finalizedSlot</code> , <code>snapshotSlot</code>	Latest finalized cycle and account snapshot
<code>historyComplete</code>	False while any gap is recorded
<code>lastHealthyAt</code>	Time of the last healthy cycle
<code>gaps</code> , <code>gapsTruncated</code>	Up to 100 slot ranges with their reasons

Global history can be partial while a newly created pool has complete history of its own.

Freshness. Every read runs inside a transaction that first checks the indexer: unavailable RPC, a genesis hash that differs from the configured cluster, or a checkpoint older than 30 seconds fails the read with `UNAVAILABLE` rather than serving stale state.

Data boundary. Indexed instruction data keeps known Hindsight instruction prefixes, public account addresses, public signers and opaque ciphertext. Proof bytes, memo instructions, program logs and

trailing calldata are dropped. Notifications are written per wallet with kinds `NEW_NOTE` , `ATTESTED` , `ACCEPTED` and `RETURNED` , a public UUID and a private monotonic sequence that orders delivery.

11.2 The API

The API is one Node service in `APP_MODE=chain` , bound to the port the platform injects. It refuses to start in production in any other mode, requires HTTPS browser origins, and serves only the hosted domain of the contract; auditor routes are never proxied through it.

11.2.1 Runtime configuration

`GET /v1/runtime` (`runtimeConfig`) is the first call every client makes. It is decoded from chain accounts at the time of the request, never from fixtures:

Field	Content
<code>status</code>	<code>ok</code> , or <code>waiting</code> while the program is not deployed or a configured pool is not initialized
<code>cluster</code> , <code>genesisHash</code> , <code>programId</code>	Cluster identity, checked against the known genesis hash of devnet or mainnet-beta
<code>rpcUrl</code> , <code>rpcRelay</code>	The API's RPC relay endpoint and <code>true</code> on keyed clusters
<code>rentRecovery</code>	<code>status</code> (<code>not_initialized</code> or <code>active</code>), the <code>RentRecoveryRegistry</code> address, <code>closeNonce</code> and <code>initialAccountSequence</code> , the sequence a <code>ConfAccount</code> opened now starts at
<code>buyback</code>	<code>status</code> (<code>not_configured</code> or <code>active</code>), the <code>BuybackConfig</code> address, the \$HIND mint, its token program, the keeper, <code>minimumOutputNumerator</code> , <code>minimumOutputDenominator</code> and <code>totalBurnedBaseUnits</code>
<code>pools[]</code>	Each pool's address, mint, vault, <code>principalBaseUnits</code> , protocol fee account and its balance (<code>protocolFeeAccount</code> , <code>protocolFeeBalanceBaseUnits</code>), auditor attestation key and KEM epoch, and full policy
<code>pools[].mint</code>	Mint address and decimals, read from the mint account
<code>pools[].auditorKeyRecord</code>	Address, owner, epoch and the 1,216-byte encapsulation key payers seal to
<code>features</code>	<code>walletSessions</code> , <code>relay</code> , <code>notifications</code>

The API verifies before answering that the Pool and vault PDAs derive from the mint, the vault is a classic SPL token account owned by the pool for that mint, and the auditor `KeyRecord` belongs to the pool's auditor at the pool's epoch. A pool whose ramp fee lies within 0 to 100 basis points is reported `active` with its protocol fee account; a pool with a nonzero flat transfer fee is reported `configuration_required` . Before the `BuybackConfig` exists, `buyback` is `not_configured` , with null mint, program, keeper and ratio and a zero total; once set it is `active` and reports the configuration and the on-chain `total_burned` , in base units. The provider URL never appears in the response.

11.2.2 Reads

Public reads describe pools and keys: `pools` , `pool` , `vault` , `poolAuditorKey` , `currentKey` , `keyHistory` , `keySnapshot` , `indexerCoverage` , and `envelopes` (`GET /v1/envelopes/:hash`), which serves a stored pair only when a finalized Note carries that hash. Wallet-scoped reads require a session and return only the caller's own data: `onboarding` , `accounts` , `account` , `publicHistory` (already-

public deposit and withdrawal amounts), `inbox`, `outbox`, `note`, `noteSnapshots`, `transaction` and `notifications`. Accounts come back as commitments and encrypted self-records; the client opens its own balance. A closed `ConfAccount` is omitted from `accounts` and onboarding state and returns `NOT_FOUND` from `account`, and `currentKey`, `keySnapshot` and `keyHistory` omit closed `KeyRecords`; Notes, ramps and events indexed before a close remain.

11.2.3 Envelope relay

`PUT /v1/envelopes/:hash ( relayEnvelopes )` accepts a payer's envelope pair for a Note it is about to create, so the payee can fetch it by hash without archival RPC history. The relay requires a wallet session, CSRF token and `Idempotency-Key`; it checks that both envelopes are exactly 1,208 bytes and that `SHA-256(payee || auditor)` equals the path hash. If the finalized Note already exists, its sender must be the session's wallet and its `envelope_hash` the uploaded hash. Otherwise the upload becomes a pending record: at most `RELAY_MAX_PENDING_PER_WALLET` (32) per wallet and `RELAY_MAX_PENDING_TOTAL` (1,000) in all, expiring after `RELAY_PENDING_TTL_SECONDS` (900 s). Every five seconds a reconciliation pass promotes pending uploads whose finalized Note matches sender and hash and discards the rest. Retry receipts live for `RELAY_IDEMPOTENCY_TTL_SECONDS` (86,400 s). An upload receipt confirms storage of ciphertext, not settlement; signed transactions go from the wallet to the chain, never through the relay.

11.2.4 Sessions and notifications

Wallet sessions follow the challenge, verify and session flow described in the security chapter: `authChallenge`, `authVerify`, `session`, `logout`. Notifications are available two ways. `GET /v1/notifications` pages them by cursor. `GET /v1/events` streams them as server-sent events: each event carries the notification's UUID as its `id`, the stream polls every second and sends heartbeats, re-authenticates the session on every poll, and resumes from `Last-Event-ID` or a cursor. A wallet may hold five concurrent streams. A notification names a Note and an event kind; the payee's client derives "ready to accept" from its own decryption, and no notification carries an amount, opening or memo.

11.2.5 Payroll progress and access requests

`savePayrollProgress`, `payrollProgress` and `payrollPlans` store and reconcile the public routing of a payroll run (described in the payroll chapter): an encrypted-plan digest, lane accounts and ordered Note addresses, with counts derived from finalized chain facts. `createAccessRequest` persists a contact form with HMAC-protected retry keys, a retention period set by `ACCESS_REQUEST_RETENTION_DAYS` and hourly cleanup; it stores no raw retry key, client address or honeypot value.

11.2.6 The RPC relay

Browsers need an RPC endpoint to read accounts and submit signed transactions, and production endpoints carry a provider key in their URL. `POST /v1/rpc ( rpcRelay )` keeps that key on the server. Runtime configuration points `rpcUrl` at it and sets `rpcRelay: true`.

Aspect	Rule
Methods	<code>getAccountInfo</code> , <code>getMultipleAccounts</code> , <code>getLatestBlockhash</code> , <code>getEpochInfo</code> , <code>getSignatureStatuses</code> , <code>getBalance</code> , <code>getTokenAccountBalance</code> , <code>getTokenAccountsByOwner</code> , <code>getGenesisHash</code> , <code>getSlot</code> , <code>sendTransaction</code> , <code>simulateTransaction</code> ; at most 10 parameters
Submissions	<code>sendTransaction</code> must carry canonical base64 of at most 4,096 bytes that decodes to a transaction invoking the Hindsight program, with no programs other than compute budget, Lighthouse, the ZK ElGamal Proof program and the associated-token program
Rate	<code>RPC_RELAY_RATE_LIMIT_MAX</code> requests per client per minute (120 by default)
Cache	Identical read requests share one in-flight call and a cached result for <code>RPC_RELAY_READ_CACHE_MS</code> (1,000 ms by default, at most 2,000 ms), up to 512 entries; submissions are never cached
Upstream	10-second timeout, no redirects, 8 MiB response limit; failures return one fixed JSON-RPC error
Logging	The method name only; request bodies, parameters and provider diagnostics are never written

The Lighthouse allowance exists because wallets such as Phantom append its guard instructions to transactions they sign.

11.3 The auditor service

The auditor runs as its own service from the same source tree, started with `npm run auditor:start`. Its worker and console API are described in the auditor chapter; its deployment properties are these:

- **Own database.** `AUDITOR_DATABASE_URL` points at `AuditorPostgres` , which holds jobs, ledger ciphertext, receipts, exports, idempotency records, challenges, sessions and operator scopes. No other service has credentials for it.
- **Read-only feed.** On every start the API creates or updates the role `hindsight_auditor_feed` with the password `AUDITOR_FEED_PASSWORD` , grants it SELECT on nine chain tables (`IndexedTransaction` , `IndexerCheckpoint` , `IndexerGap` , `Pool` , `ConfAccount` , `KeyRecord` , `Note` , `EnvelopeSet` , `ChainEvent`) and sets `default_transaction_read_only` for it. The auditor's `AUDITOR_FEED_DATABASE_URL` logs in as that role and never uses the API's writer credential.
- **Custody from secrets.** The attestation seed and per-epoch X-Wing seeds arrive as `AUDITOR_ATTESTATION_SEED_HEX` and `AUDITOR_XWING_EPOCHS_JSON` through the software custody adapter; an HSM adapter replaces it behind the `AuditorHsm` interface.

- **Identity checks.** On devnet and mainnet-beta the service requires HTTPS for its console origins, its audience and its RPC URL, requires the configured genesis hash to equal the cluster's known genesis, and requires its console origins to differ from the payment app's.
- **Operators.** Wallets listed in `AUDITOR_MANAGER_WALLETS` receive manager scopes for the configured pools at startup.

The auditor pays attestation fees in SOL from its attestation key; program initialization funds that key, and its balance is public.

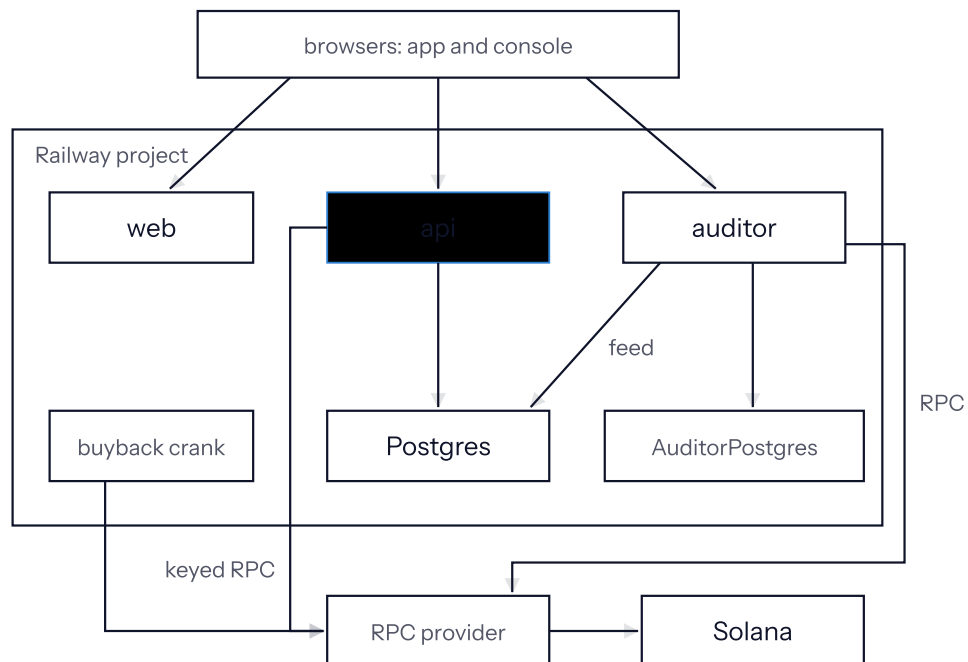
11.4 The buyback crank

The crank is the keeper's side of the buyback. It is one command, `npm run buyback`, run from `backend/` with the keeper key, and each run builds, signs and submits at most one buyback transaction:

1. **Read.** It checks the cluster's genesis hash, reads the `BuybackConfig`, and stops unless the keeper key and the `$HIND` mint match it and no claim is pending. It claims the protocol fee account's whole balance, or the amount given with `--amount`, and refuses a claim above that balance or below `BUYBACK_MIN_CLAIM_BASE_UNITS`.
2. **Quote.** With the Jupiter adapter on mainnet-beta it requests an exact-input quote from `https://lite-api.jup.ag/swap/v1/quote` with `maxAccounts=48` and the configured slippage, refuses a quote whose minimum output is below the on-chain floor, and fetches the route's instructions from `swap-instructions` with the protocol token account as their destination. On localnet and devnet, which have no Jupiter, the stand-in adapter replaces the swap with two exact token payments, the claimed USDC from the keeper to a local market maker and the bought tokens from the market maker into the protocol token account, and the market maker co-signs.
3. **Build.** It assembles one v0 transaction: the compute-unit limit and price, idempotent creation of the keeper's USDC account and the protocol token account, the route's setup instructions, `claim_buyback` with the quote's minimum output and the configuration's `sequence`, the swap instructions, and `complete_buyback` last. It compresses the message with the route's address lookup tables and refuses wire bytes over 1,232. `--dry-run` prints the planned public instructions, accounts and bounds without signing.
4. **Send once.** It signs with the keeper key, sends the bytes and polls that one signature to finality, so a lost response never leads to a second claim. It then reads the finalized transaction, sums the burns from the protocol token account, and requires them to reach the claim's minimum output.

The crank reaches the chain through its own RPC endpoint, not through the API's RPC relay. The keeper needs SOL for transaction fees, the priority fee and associated-account rent, and no USDC or `$HIND` inventory. Neither API startup nor local development starts the crank; it runs only when invoked.

11.5 Deployment



Deployment on Railway. Browsers load web and call api and auditor directly on their own origins. api owns Postgres; auditor owns AuditorPostgres and reads Postgres through a read-only feed role. The buyback crank runs on a schedule, has no database and sends its transaction through its own keyed RPC. Only the backend services hold the provider's keyed RPC URL; browsers reach the chain through the api's RPC relay.

The deployment runs on Railway: four services, the buyback crank among them as a cron service, and two PostgreSQL databases in one project.

Service	Source root	Build	Start	Health check
web	web/	npm ci && npm run build	npm start (next start on PORT)	—
api	backend/	npm run build (Railpack, npm ci --include=dev)	npm start	/health
auditor	backend/	npm run build	npm run auditor:start	/health
Buyback crank (cron)	backend/	npm ci --include=dev && npm run build	npm run buyback , on an interval such as every 15 minutes	—
Postgres	PostgreSQL template			
AuditorPostgres	PostgreSQL template			

Build without the database. Railway's build phase cannot reach its private network, so the build only generates code: `npm run build` runs `prisma generate` for both the hosted and the auditor Prisma clients, compiles the contract and SDK packages, and compiles the services. No migration runs at build time.

Migrations at start. `npm start` for the API runs `prisma migrate deploy` against `Postgres` , then provisions the feed role, then starts the server. `npm run auditor:start` runs `prisma migrate deploy` with the auditor's own Prisma configuration against `AuditorPostgres` , then starts the worker and console; the runtime itself refuses to start if any private migration is pending, failed or altered. Both services bind the `PORT` Railway injects and never hardcode one.

The crank's schedule. The buyback crank is a separate Railway cron service built from `backend/` . It has no database and no migration step, runs as one instance, and serves one fee mint per job, so several pool mints take one job each. It receives only the crank's variables and the keeper key, supplied through the owner's custody process.

Health. `/health` answers HTTP 200 with `{"status":"ok"}` when the service is ready, and HTTP 200 with `{"status":"waiting"}` while the program is not yet deployed or a configured pool is not yet initialized, so the platform keeps the service running through initialization. Both backend services restart on failure, up to ten times, with a 120-second health-check timeout.

Variables. Cross-service values use Railway reference variables, and secrets are set only on the service that needs them.

Service	Variable	Value
api	DATABASE_URL	<code>\${{Postgres.DATABASE_URL}}</code>
api	SOLANA_RPC_URL	The provider's keyed HTTPS URL (see SOLANA_RPC_URL)
api	SOLANA_CLUSTER , HINDSIGHT_PROGRAM_ID , APP_POOL_ADDRESSES	Public chain identity from program initialization
api	SERVICE_AUDIENCE , ALLOWED_ORIGINS	The API's own HTTPS origin; the web exact HTTPS origin
api	RELAY_HMAC_KEY , ACCESS_REQUEST_HMAC_KEY , AUDITOR_FEED_PASSWORD	Independent 32-byte random secret
auditor	AUDITOR_DATABASE_URL	<code>\${{AuditorPostgres.DATABASE_URL}}</code>
auditor	AUDITOR_FEED_DATABASE_URL	The feed role's login, built from <code>\${{api.AUDITOR_FEED_PASSWORD}}</code> a <code>Postgres</code> host, port and database references
auditor	AUDITOR_RPC_URL	The provider's keyed HTTPS URL (see SOLANA_RPC_URL)
auditor	AUDITOR_ATTESTATION_SEED_HEX , AUDITOR_XWING_EPOCHS_JSON	Custody seeds (secrets)
auditor	AUDITOR_LEDGER_KEY_HEX , AUDITOR_SESSION_HMAC_KEY_HEX	Independent 32-byte random secret
auditor	AUDITOR_SERVICE_AUDIENCE , AUDITOR_ORIGINS , AUDITOR_MANAGER_WALLETS	Its own HTTPS origin, the console's or manager wallets
web	NEXT_PUBLIC_API_ORIGIN	<code>https://\${{api.RAILWAY_PUBLIC_DOMAIN}}</code>
Buyback crank	BUYBACK_CLUSTER , BUYBACK_ADAPTER	<code>mainnet-beta</code> with <code>jupiter</code> ; <code>local devnet</code> with the <code>local</code> stand-in
Buyback crank	BUYBACK_RPC_URL	The provider's keyed HTTPS URL (see SOLANA_RPC_URL); falls back to <code>RPC_URL</code> or <code>SOLANA_RPC_URL</code>
Buyback crank	BUYBACK_KEEPER_KEYPAIR_JSON	The keeper's 64-byte keypair as a JSON array (secret); <code>BUYBACK_KEEPER_KEYPAIR_FILE</code> names a private keypair file instead

Service	Variable	Value
Buyback crank	<code>BUYBACK_TOKEN_MINT</code> , <code>BUYBACK_FEE_MINT</code>	The \$HIND mint, equal to the on-chain configuration; the six-decimal USDC falling back to <code>USDC_MINT</code>
Buyback crank	<code>BUYBACK_SLIPPAGE_BPS</code> , <code>BUYBACK_MIN_CLAIM_BASE_UNITS</code>	An explicit slippage from 0 to 1,000 bps; the smallest claim, in USDC base units
Buyback crank	<code>BUYBACK_PROGRAM_ID</code> , <code>BUYBACK_COMPUTE_UNIT_PRICE_MICROLAMPORTS</code> , <code>BUYBACK_JUPITER_API_KEY</code>	Optional: the program ID, the compute price in micro-lamports (1,000 by default), and the Jupiter provider key (secret)

The browser receives API origins and nothing else; it learns cluster, program, pools and the RPC route from runtime configuration.

Program deployment and initialization. Two operator commands bring the chain side up, run from an operator machine that holds the deployer, program and admin keypairs; none of those keys is placed on Railway. `program:deploy` first checks that the cluster's genesis hash matches and that every runtime feature the program needs is active: v1 transactions, SBPF v3, the Curve25519 syscalls, loaded-account-data budgeting and the re-enabled ZK ElGamal Proof program. It reports the cost before uploading, deploys through a retained buffer so an interrupted upload resumes, and verifies that the deployed ProgramData bytes hash to the reviewed local artifact. `program:init` then bootstraps the RentRecoveryRegistry once with `initialize_rent_recovery` , signed by the current upgrade authority and listing every canonical Pool the program already has, creates the pool for the cluster's USDC mint with the policy from `deploy/pool.<cluster>.json` , creates the vault and the protocol fee account, registers the auditor's `KeyRecord` , funds the auditor's fee balance, and prints the public values (program, genesis hash, pool, mint, vault, protocol fee account, auditor and epoch) that go into the services' variables. Rerunning it skips what exists and stops on any mismatch rather than overwriting an immutable account. A separate, one-time `program:init -- --buyback` , signed by the current upgrade authority, sends `configure_buyback` with `HINDSIGHT_BUYBACK_TOKEN_MINT` , `HINDSIGHT_BUYBACK_KEEPER` , `HINDSIGHT_BUYBACK_MIN_OUTPUT_NUMERATOR` and `HINDSIGHT_BUYBACK_MIN_OUTPUT_DENOMINATOR` (or the matching flags), creates the protocol token account under the mint's own token program, and prints only public values; an identical rerun skips, and a mismatch stops. The services discover the initialized accounts on their next request, and `/health` moves from `waiting` to `ok` without a redeploy.

Upload boundary. When the backend root is uploaded directly, an anchored `.railwayignore` excludes dependencies, every environment file except the template, caches and keypair files, build output, validator ledgers and test artifacts, keeping build sources, migrations and the packaged runtime assets.

12 Payroll

A payroll run is many transfers from one employer to many payees, prepared together and executed as ordered chains. Hindsight's batcher runs entirely on the payer side, in the employer's browser or in a process the employer controls: it parses the CSV, plans lanes, computes every commitment, opening, envelope and self-record ahead of time, encrypts the plan and its execution journal, and submits, retries and reconciles. The hosted service learns only public routing: which accounts send which Notes, in which order.

Stays with the payer	Reaches the hosted service
The CSV, salaries and memos	An opaque digest of the encrypted plan
Lane plan, openings, envelopes before submission, self-records	Lane account addresses and their starting sequences
Proof witnesses and wallet signatures	The ordered Note addresses of each lane
Plan and journal encryption keys	Counts derived from finalized chain facts

12.1 The payee CSV

`parsePayrollCsv(text, { amountUnit })` reads a strict CSV with these columns, in any order:

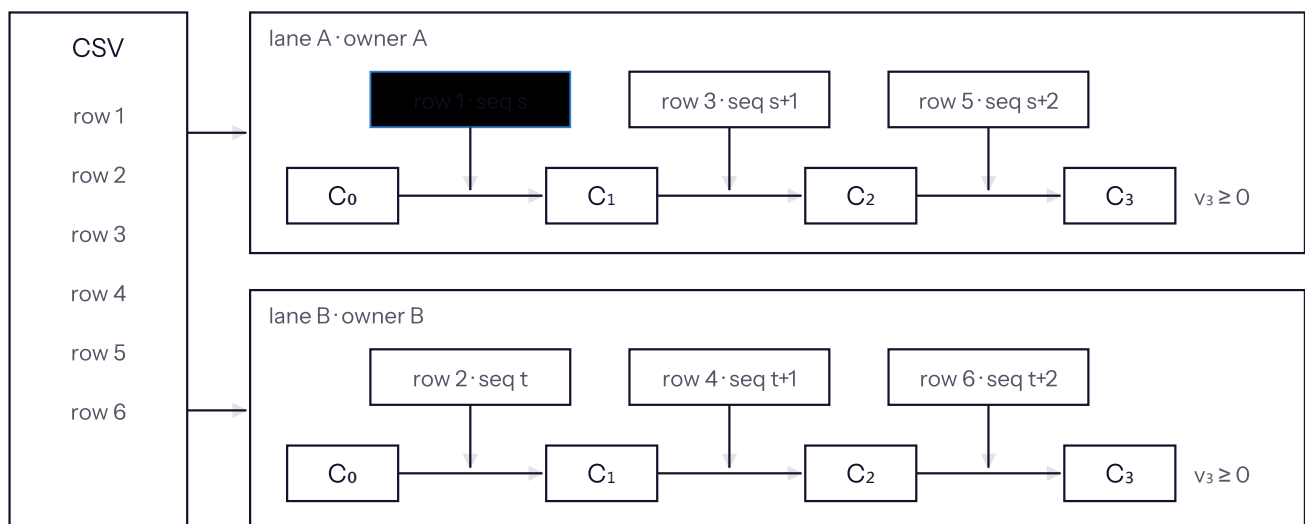
Column	Required	Rule
<code>payee</code>	yes	The payee's wallet address
<code>amount</code>	yes	With <code>amountUnit: "usdc"</code> , at most six decimal places; with <code>"base_units"</code> , an exact u64 integer; zero is allowed
<code>row_id</code>	no	1 to 96 characters of <code>[A-Za-z0-9_-]</code> , unique; defaults to <code>row_1</code> , <code>row_2</code> , ...
<code>memo</code>	no	At most 32 UTF-8 bytes, no NUL

The unit is mandatory, so no file is ever interpreted in the wrong scale; a `usdc` amount $w.f$ becomes $w \cdot 10^6 + f$ base units by integer arithmetic, and any value with more precision is rejected rather than rounded. Unknown or duplicate headers, malformed quoting, mismatched cell counts, duplicate row IDs, invalid addresses and overflowing amounts are rejected; quoted commas, escaped quotes, CRLF or LF line endings and multiline quoted memos are accepted. The parser stops at 10,000 rows and four million input code units. Issues are reported as a row number, a field and a fixed code (`invalid_format` , `required` , `invalid` , `duplicate` , `too_long` , `too_many_rows`) without echoing the rejected value. Repeated payee addresses are allowed: each row is its own payment intent.

12.2 Lanes

A `ConfAccount` lives at `["acct", pool, owner]` , so one owner has one account per pool, and every transfer from that account consumes its next sequence number. Transfers from one account are therefore strictly ordered: each needs the commitment the previous one produced. Parallelism comes from several accounts, which means several payer-controlled owner wallets: one employer runs a payroll over L lanes, each a separately funded `ConfAccount` .

`planPayroll({ rows, laneAccounts })` distributes rows round-robin and keeps input order within each lane: row i (counting from zero) goes to lane $i \bmod L$. Lane accounts must be distinct and no more numerous than rows. The plan receives a random 16-byte local identifier. Planning does not assume the lanes are funded; preparation checks each lane's actual balance.



Lanes and ordered chains. Rows are dealt round-robin to lanes, each a `ConfAccount` of a different payer-controlled owner. Within a lane, every Note's sequence, PDA and commitment and every intermediate balance commitment C_0 to C_3 are computed before the first transaction is signed; lanes run in parallel, steps within a lane in order.

12.3 Ordered chains, computed ahead

`preparePayroll` turns the plan into a complete, executable chain per lane. For a lane whose account holds commitment C_0 with opening (v_0, r_0) at sequence s_0 , and whose rows carry amounts $a_1, \dots, a_k$, step j computes

$$N_j = a_j G + \rho_j H, \quad C_j = C_{j-1} - N_j, \quad (v_j, r_j) = (v_{j-1} - a_j, r_{j-1} - \rho_j),$$

with a fresh blinding ρ_j from a CSPRNG, the Note address `PDA("note", pool, owner,  $s_0 + j - 1$ )`, the payee and auditor envelopes sealed with that sequence and N_j in their AAD, and the self-record the account will hold afterwards, sealed for sequence $s_0 + j$ and commitment C_j . The chain is valid only if

$$v_k = v_0 - \sum_{j=1}^k a_j \geq 0,$$

which preparation checks against the lane's verified starting balance. Every prepared step records its row, lane and chain index, sequence, Note address, recipient account, both key epochs, the commitments and self-records before and after, the Note commitment and opening, both envelopes and their pair hash.

Preparation runs in two phases, and nothing is sealed until the first finishes. The preflight reads every account the batch depends on through an explicit `chainAccounts` reader, typically `createPayrollAccountRpc({ rpcUrl })`, which calls `getAccountInfo` at confirmed commitment on an RPC endpoint the payer configured, without credentials or redirects. It decodes the Pool, every source and payee `ConfAccount` and every payee and auditor `KeyRecord`, and checks program ownership, discriminators, exact sizes and PDAs. It binds the auditor and its epoch to the actual Pool, so a forged auditor hint is rejected even when the attacker has registered a real `KeyRecord`; compares every hinted X-Wing public key byte for byte with the chain; checks payee epochs and frozen flags; and opens each source self-record under its account, sequence and commitment to confirm the supplied starting opening. Only then does the second phase generate Notes, envelopes and self-records, so a bad final row cannot cause earlier rows to be sealed. Payee lookups during preparation send only public wallet addresses.

12.4 Encrypted plans and journals

The prepared plan contains every salary and opening of the run, so it never exists unencrypted outside payer memory. `sealPayrollPlan` encrypts it once with AES-256-GCM under a payer-controlled 256-bit key, a fresh 12-byte nonce and the associated data `hindsight:payroll-storage:v1:plan:<localId>`. Its digest is **SHA-256** of the randomized ciphertext, so the digest that reaches the hosted service reveals nothing about the CSV it came from. The sealed plan is immutable for the life of the run.

Execution state lives in a separate journal, sealed after every change with the associated data

`hindsight:payroll-storage:v1:journal:<localId>:<planDigest>`, which binds each journal to exactly one plan. A journal records, per lane and step, the row ID, Note address, status, attempt count, the signed attempt (wire bytes, signature, blockhash, last valid block height, message hash), earlier signatures and a failure code.

Journal	Values
---------	--------

Run state	<code>ready</code> , <code>running</code> , <code>paused</code> , <code>completed</code> , <code>failed</code>
-----------	--

Step status	<code>planned</code> , <code>signed</code> , <code>submitted</code> , <code>confirmed</code> , <code>finalized</code> , <code>paused</code> , <code>failed</code>
-------------	---

Failure code	<code>ACCOUNT_CHANGED</code> , <code>NOTE_MISMATCH</code> , <code>CHAIN_REJECTED</code> , <code>SUBMISSION_UNCERTAIN</code> , <code>ADAPTER_UNAVAILABLE</code> , <code>ATTEMPTS_EXHAUSTED</code>
--------------	--

The storage key belongs to the payer; `generateSelfRecordKey()` can create an independent non-extractable one. Opening a journal requires the plan's identifier and digest, and a mismatch fails.

12.5 Execution, retries and reconciliation

`executePayroll({ prepared, journal, adapter })` runs lanes concurrently and the steps of each lane in order. The payer supplies the adapter: chain reads (`readAccount`, `readNote`, `transactionStatus`, `latestBlockhash`, `blockHeight`), a wallet `sign`, an RPC `submit`, and `saveJournal`, which must persist the encrypted journal atomically. For each step the executor re-reads the lane account, generates the range proof, builds the v1 transaction, has the wallet sign it, verifies the returned wire (one signer, the lane owner, a strict Ed25519 signature over the expected message), and saves the signed attempt to the journal before it submits. That durability barrier means a crash at any point leaves the journal knowing every transaction that might have been sent.

Retries reuse what was prepared:

- **Lost response.** A submission whose outcome is unknown is reconciled against the exact expected Note and signature; while its blockhash is valid, the same signed bytes may be resent.
- **Expired blockhash.** An attempt that never landed is rebuilt with a new blockhash and signature, keeping its Note address, sequence, commitment, opening, envelopes and self-record. No retry introduces new randomness, so a payee can never receive two different Notes for one row.
- **Unexpected state.** A sequence, balance commitment, self-record, key epoch or frozen flag that does not match the prepared chain pauses the lane with `ACCOUNT_CHANGED`. Replanning is an explicit payer decision.

The defaults are a 300,000 CU limit, an 8 MiB loaded-account-data limit, 250 ms between observations, 40 observations per attempt and three signed attempts per row; a caller may lower them. A paused run

resumes from the same sealed plan and journal. A journal state of `completed` means every transfer was observed confirmed or finalized; it does not mean every payee has accepted.

12.6 Progress and settlement

Settlement happens when payees accept. To follow it, the payer saves a public progress record with `savePayrollProgress` (`PUT /v1/payroll-plans/:id` , with an `Idempotency-Key`): a plan identifier, the pool, the encrypted-plan digest and, per lane, the account, its starting sequence and its ordered Note addresses, at most 32 lanes and 2,000 Notes. The service checks that the session's wallet owns the first lane, that every account belongs to the pool, and that each listed address is exactly the Note PDA of its lane owner at starting sequence plus position. It accepts no CSV, amount, memo, opening or key.

`payrollProgress` then reports every count from finalized chain facts, never from client-submitted counters:

Level	States and counts
-------	-------------------

Lane	<code>ready</code> , <code>running</code> , <code>blocked</code> (a returned Note, a frozen account or an unexpected sequence), <code>completed</code>
------	--

Plan	<code>planned</code> , <code>running</code> , <code>paused</code> , <code>completed</code> (every Note accepted), <code>failed</code> (any Note returned)
------	---

Counts	<code>totalCount</code> , <code>acceptedCount</code> , <code>returnedCount</code>
--------	---

A returned payout counts as returned, not settled, even though its transfer landed; accepting it back and sending a replacement are explicit payer actions. The batcher was exercised on a local validator with 50 payees over two lanes of 25, including an encrypted-journal restart and a deliberately lost RPC response; every one of the 50 payment transactions measured 4,024 bytes, no payment was duplicated, all 50 Notes were accepted, and the pool's principal was fully withdrawn at the end.

13 Reference

This chapter collects the protocol's constants and sizes, the terms the whitepaper uses, and the standards and specifications the design relies on. Values come from the protocol contract and the site's facts module. The status column follows the contract's own labels: *exact* for fixed encodings, *derived* for values computed from exact ones, *draft* for choices awaiting cryptographic review, *estimate* for the brief's budgets, *target* for goals, and *measured* for one run on a local Agave 4.3.0 validator.

13.1 Parameters

13.1.1 Amounts, commitments and envelopes

Parameter	Value	Unit	Status
Amount width	64	bits	exact
Amount encoding	8	bytes, little-endian	derived
Largest amount	18,446,744,073,709,551,615	USDC base units	exact
USDC decimals	6	digits	exact
Token-2022 amount width, for comparison	48, split 16 + 32	bits	exact
Decryption handles per transfer	0	count	exact
Pedersen commitment	32	bytes, compressed Ristretto255	exact
Blinding scalar	32	bytes	derived
Memo, padded	32	bytes	exact
Opening plaintext ($v \parallel r \parallel \text{memo}$)	72	bytes	derived
X-Wing encapsulation key	1,216	bytes	exact
X-Wing decapsulation key, compact seed	32	bytes	exact
X-Wing ciphertext	1,120	bytes	exact
X-Wing shared secret	32	bytes	exact
AES-256 key	32	bytes	derived
AES-GCM tag	16	bytes	derived
AES-GCM nonce, derived per encapsulation, not transmitted	12	bytes	derived
AEAD ciphertext and tag	88	bytes	exact

Parameter	Value	Unit	Status
Envelope	1,208	bytes	derived
Envelopes per transfer (payee, auditor)	2	count	exact
Envelope pair	2,416	bytes	derived
Envelope-pair hash, SHA-256	32	bytes	exact
Self-record (nonce, ciphertext, tag)	68	bytes	exact
ML-KEM-1024 ciphertext, CNSA-aligned profile	1,568	bytes	exact

13.1.1.2 Transactions

Parameter	Value	Unit	Status
v1 transaction limit	4,096	bytes	exact
Legacy transaction limit	1,232	bytes	exact
Signature, header and config	90	bytes	estimate
9 account keys and blockhash	320	bytes	estimate
U128 proof instruction (736 + 264 + 1)	1,001	bytes	estimate
Commitment, sequence and self-record	110	bytes	estimate
Payee envelope	1,208	bytes	exact
Auditor envelope	1,208	bytes	exact
Sum of estimated parts	3,937	bytes	estimate
Compute budget and remainder	87	bytes	derived from measurement
Transfer, one v1 transaction	4,024	bytes	measured
Transfer compute	226,570	CU	measured
Withdrawal, one v1 transaction	1,479	bytes	measured
Withdrawal compute	140,575	CU	measured
<code>transfer</code> instruction data, canonical prefix	2,540	bytes	exact

13.1.3 Range proofs

[illegible]

Bulletproof size follows $(2 \log_2 n + 9) \times 32$ bytes for an aggregate of n range bits: 23 group and scalar elements for $n = 128$.

13.1.1.4 Compute

Parameter	Value	Unit	Status
U64 proof verification	111,000	CU	exact
U128 proof verification	200,000	CU	exact
U256 proof verification	368,000	CU	exact
Ristretto addition	521	CU	exact
Ristretto multiplication	2,208	CU	exact
<code>init_pool</code>	under 20,000	CU	estimate
<code>register_key</code> , <code>open_account</code>	under 10,000 each	CU	estimate
<code>deposit</code>	about 15,000	CU	estimate
<code>transfer</code> , including the U128 proof	230,000 to 250,000	CU	estimate
<code>attest</code>	about 5,000	CU per Note	estimate
<code>accept</code>	5,000 to 10,000	CU per Note	estimate
<code>return_note</code>	about 5,000	CU	estimate
<code>withdraw</code> , including the U64 proof	about 130,000	CU	estimate
<code>freeze</code> , <code>thaw</code>	under 5,000 each	CU	estimate
Transfer budget	at most 300,000	CU	target
Auditor attestation latency, 95th percentile	under 10	seconds	target
Transfer build on a phone	under 1	second	target

13.1.5 Accounts and rent

Account	Seeds	Allocation	Status
Pool	<code>["pool", mint]</code>	238 bytes, 53 of them policy	exact
Vault	<code>["vault", pool]</code>	165 bytes	exact
KeyRecord	<code>["kem", owner, epoch_le]</code>	1,265 bytes	exact
ConfAccount	<code>["acct", pool, owner]</code>	190 bytes	exact
Note	<code>["note", pool, sender, sequence_le]</code>	235 bytes	exact
BuybackConfig	<code>["buyback"]</code>	209 bytes	exact
RentRecoveryRegistry	<code>["rent-recovery"]</code>	21 bytes, plus 32 per protected auditor key	exact
Protocol fee account	associated token account of the PDA <code>["protocol"]</code> for the pool's mint	165 bytes	exact

Allocations include the 8-byte Anchor discriminator. An account of s bytes holds $(s + 128)$ times the cluster's lamports per byte as rent, read live from the Rent sysvar, and every participant account's rent returns to its payer when it closes: a Note's on `accept`, a ConfAccount's on `close_account`, a KeyRecord's on `close_key_record`. The return timeout is counted in slots and bounded by $2^{32} - 1 = 4\,294\,967\,295$, an engineering limit rather than a policy default.

13.1.1.6 Instructions

Instruction	Discriminator	Signer
<code>init_pool</code>	<code>74e9c7cc739fab24</code>	payer and admin authority (the upgrade authority)
<code>register_key</code>	<code>38084361807a50d5</code>	owner
<code>open_account</code>	<code>65985bf3b024bd65</code>	owner
<code>deposit</code>	<code>f223c68952e1f2b6</code>	owner
<code>transfer</code>	<code>a334c8e78c0345ba</code>	owner (sender)
<code>attest</code>	<code>53947877908b75a0</code>	pool auditor
<code>accept</code>	<code>419646d885066b04</code>	owner (recipient)
<code>return_note</code>	<code>9e58488eb5d10935</code>	payee, or sender after the timeout
<code>withdraw</code>	<code>b712469c946da122</code>	owner
<code>close_account</code>	<code>7dff950e6e224818</code>	owner
<code>close_key_record</code>	<code>755304f9cf37303b</code>	owner
<code>freeze</code>	<code>ff5bcf54fbc2fe3f</code>	freeze authority
<code>thaw</code>	<code>e2f92239bd15b165</code>	freeze authority
<code>initialize_rent_recovery</code>	<code>759246be85359cff</code>	payer and upgrade authority
<code>configure_buyback</code>	<code>ff699821c0dd4a18</code>	payer and upgrade authority
<code>claim_buyback</code>	<code>c7fff3ecc7acc98d</code>	keeper
<code>complete_buyback</code>	<code>d04ca859f26f8d4b</code>	keeper

13.1.1.7 Error codes

Anchor numbers the program's 56 custom errors from 6000 in declaration order, and the program chapter lists every one with the instructions that raise it. Codes 6000 to 6037 cover pools, keys, accounts, notes and proofs; the buyback adds 6038 to 6050 and rent recovery 6051 to 6055:

Code	Name	Message
6038	BuybackNotConfigured	Buyback token mint and keeper have not been configured.
6039	InvalidBuybackConfiguration	Buyback configuration requires a keeper and a positive minimum output ratio.
6040	InvalidBuybackMint	The configured buyback token mint or token program does not match.
6041	UnsupportedBuybackMintExtension	Only metadata pointer and token metadata extensions are supported for the buyback mint.
6042	InvalidProtocolFeeAccount	The protocol fee account is not its canonical associated token account.
6043	InvalidBuybackAccount	A buyback token account is not the expected associated token account.
6044	InvalidBuybackTransaction	Buyback requires one top-level claim, a swap step and the final matching completion.
6045	BuybackAlreadyPending	A buyback claim is already pending.
6046	InvalidBuybackAmount	The buyback claim amount is zero or exceeds available fees.
6047	BuybackOutputTooSmall	The bought token amount is below the on-chain bound.
6048	BuybackNotPending	No buyback claim is pending.
6049	BuybackInputNotSpent	The swap did not spend the whole claimed input amount.
6050	BuybackSequenceMismatch	The buyback claim sequence has already been used or is not current.
6051	InvalidZeroBalanceProof	The zero-balance proof is invalid or does not match this account.
6052	StaleRentRecoveryNonce	The rent recovery nonce has changed; refresh and rebuild the transaction.
6053	AuditorKeyInUse	A current pool auditor key record cannot be closed.

Code	Name	Message
6054	SequenceRangeExhausted	The account sequence generation has reached its engineering bound.
6055	InvalidRentRecoveryRegistry	The rent recovery registry bootstrap contains an invalid pool.

13.1.1.8 Chain

Fact	Value
ZK ElGamal Proof program re-enabled	Epoch 982, slot 424,224,000; feature zkexuyPRdyTVbZqEAREueqL2xvvoBhRgth9xGSc1tMN
v1 transactions on mainnet	Epoch 1035, 2026-09-15
Measured runtime and proof SDK	Agave 4.3.0; solana-zk-sdk 7.0.1

13.1.9 Fees and token

Parameter	Value	Unit
Ramp fee on deposits and withdrawals	5 (the program accepts at most 100)	basis points
Ramp fee on an amount a	$\lfloor a \cdot 5/10\,000 \rfloor$, rounded down	base units
Flat transfer fee	0	base units
Protocol share of the ramp fee	all of it	
Protocol fee account	associated token account of the PDA <code>["protocol"]</code> for the pool's mint	
Ways out of the protocol fee account	<code>claim_buyback</code> only, inside a buyback transaction	
Buyback transaction	<code>claim_buyback</code> , one or more swap instructions, <code>complete_buyback</code> last	
Buyback configuration	BuybackConfig at <code>["buyback"]</code> , created once by the upgrade authority	
Buyback signer	the one keeper named in the BuybackConfig	
Minimum output for a claim of a	$\lceil a \cdot n/d \rceil$, with n/d fixed at configuration	\$HIND base units
Bought \$HIND	all of it burned in the same transaction	
Burn counter	<code>BuybackConfig.total_burned</code> , u128	base units
\$HIND supply	1,000,000,000, fixed	tokens
\$HIND launch	all of it in one public launch, on equal terms	

13.2 Glossary

Accept. The `accept` instruction, signed by a Note's recipient, which adds one or more Note commitments to the recipient's balance commitment, closes the Notes and refunds their rent to their senders.

Account generation. One opening of a `ConfAccount` at its address, from `open_account` to `close_account`, with its own range of 2^{32} sequence values starting at `close_nonce << 32`.

Attestation. The auditor's on-chain record, set by `attest`, that it opened its envelope for a Note and checked the opening against the Note's commitment. Required by `accept` when the pool sets `attest_required`.

Attestation gate. The pool policy `attest_required`, under which no Note is credited before it is attested, except a Note returned to its own sender.

Auditor. The single party per pool, named by `Pool.auditor`, that receives an envelope for every Note, checks it and attests. It holds an Ed25519 attestation key and X-Wing keys by epoch.

Base unit. The smallest USDC amount, 10^{-6} USDC. Every protocol amount is a u64 of base units.

Blinding. The scalar r in a commitment $vG + rH$, drawn fresh from a CSPRNG for every Note.

Buyback. The one transaction that spends protocol fees: `claim_buyback` moves an exact amount from the protocol fee account to the keeper, a swap turns all of it into \$HIND delivered to the protocol token account, and `complete_buyback`, the final instruction, burns everything that arrived.

BuybackConfig. The global account at `["buyback"]`, created once by `configure_buyback`, that fixes the \$HIND mint and its token program, the keeper and the minimum-output ratio, and counts `total_burned`.

Close nonce. `RentRecoveryRegistry.close_nonce`, the u64 count of closed accounts, which every zero-balance proof names, every `close_account` advances and every `open_account` places in the high 32 bits of the new sequence.

Commitment. A Pedersen commitment $C = vG + rH$ over Ristretto255: perfectly hiding, computationally binding, additively homomorphic.

ConfAccount. A confidential account at `["acct", pool, owner]` holding a balance commitment, a self-record, a sequence, a key epoch and a frozen flag; its owner closes it with `close_account` at a zero balance.

Coverage. An explicit statement of what a projection or export does not contain: indexer gaps with `historyComplete`, or an auditor artifact's `omittedCount` and `unclassifiedCount`.

Envelope. A fixed 1,208-byte ciphertext carrying a Note's opening and memo to one recipient: an X-Wing ciphertext followed by AES-256-GCM output, bound by associated data to the Note's pool, sender, payee, sequence, key epoch and commitment.

Epoch. A u64 naming one X-Wing key pair of an owner, registered once as a `KeyRecord`. Notes record the payee's and the auditor's epoch they were sealed to.

Finalized projection. The indexer's view of program accounts built only from finalized commitment; confirmed observations serve as hints and never credit a balance.

Incarnation. One lifetime of a Note at a given address. The auditor never carries an opening from one incarnation to another.

Job. The auditor service's durable record of one Note's check and attestation, moving through the states of the job state machine.

Keeper. The one key, named in the BuybackConfig, that can sign `claim_buyback` and `complete_buyback`; the buyback crank holds it.

KeyRecord. The account at `["kem", owner, epoch]` holding a 1,216-byte X-Wing encapsulation key; its owner closes it with `close_key_record` unless a pool names it as its auditor key.

Lane. One `ConfAccount` of a payer-controlled owner used to send part of a payroll run; lanes run in parallel, transfers within a lane in sequence order.

Note. The account created by `transfer` at `["note", pool, sender, sequence]`, holding the transferred value as a commitment until it is accepted, with the hash of its two envelopes, its creation slot, epochs and `attested` and `returned` flags.

Opening. The pair (v, r) that reproduces a commitment. A payee or the auditor checks $vG + rH = C$ before relying on an amount.

Pool. The account at `["pool", mint]` holding the pool's vault address, auditor, governance authorities, policy and public principal.

Principal. `Pool.principal`, the public sum of deposits minus withdrawals, equal to the sum of all balance and outstanding Note values.

Protocol fee account. The associated token account, for a pool's mint, of the program address `["protocol"]`, into which `deposit` and `withdraw` pay the whole ramp fee and from which only `claim_buyback` moves tokens.

Protocol token account. The associated token account of the program address `["protocol"]` for the \$HIND mint, under that mint's token program, into which the buyback's swap delivers and from which `complete_buyback` burns.

Range proof. A batched Bulletproof, verified by the native ZK ElGamal Proof program, that committed values lie in $[0, 2^{64})$.

Relay. The API's store of envelope pairs by hash, from which payees fetch envelopes without archival RPC history.

RentRecoveryRegistry. The global account at `["rent-recovery"]`, created once by `initialize_rent_recovery`, that holds the close nonce and the auditor KeyRecords that may not close.

Return. The `return_note` instruction, which re-targets a Note to its sender: by the recipient at any time, by the sender after the pool's return timeout.

Selective disclosure. An auditor report covering one account's Notes, shared outside the console.

Self-record. The 68-byte AES-256-GCM record in a `ConfAccount` of the account's own balance opening, sealed under a key only the owner holds.

Sequence. A `ConfAccount`'s u64 counter, required as `expected_sequence` by every balance change and incremented by it within the 32-bit range of its account generation.

Vault. The pool's classic SPL Token account at `["vault", pool]`, owned by the Pool PDA, holding all deposited USDC.

X-Wing. A hybrid key encapsulation mechanism combining X25519 and ML-KEM-768.

Zero-balance proof. The Schnorr proof of knowledge of ρ with $C_{bal} = \rho H$, a nonce point and a response, by which `close_account` checks that a balance commitment opens to zero without learning its blinding.

13.3 References

Curves, commitments and proofs

- H. de Valence, J. Grigg, M. Hamburg, I. Lovecruft, G. Tankersley, F. Valsorda. *The ristretto255 and decaf448 Groups*. RFC 9496, 2023. <https://www.rfc-editor.org/rfc/rfc9496>
- T. P. Pedersen. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*. CRYPTO 1991, LNCS 576.
- B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, G. Maxwell. *Bulletproofs: Short Proofs for Confidential Transactions and More*. IEEE S&P 2018. <https://eprint.iacr.org/2017/1066>
- Solana Program. *zk-elgamal-proof: zk-sdk Pedersen generators* (`pedersen.rs`). <https://github.com/solana-program/zk-elgamal-proof/blob/main/zk-sdk/src/encryption/pedersen.rs>
- Anza. *ZK ElGamal Proof program*. <https://docs.anza.xyz/runtime/zk-elgamal-proof>
- Anza. *Agave zk-elgamal-proof program, compute-unit constants*. <https://github.com/anza-xyz/agave/blob/master/programs/zk-elgamal-proof/src/lib.rs>

Key encapsulation and symmetric cryptography

- D. Connolly, P. Schwabe, B. Westerbaan. *X-Wing: general-purpose hybrid post-quantum KEM*. draft-connolly-cfrg-xwing-kem-11, IETF. <https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/>
- NIST. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard*. August 2024.
- A. Langley, M. Hamburg, S. Turner. *Elliptic Curves for Security (X25519)*. RFC 7748, 2016.
- H. Krawczyk, P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869, 2010.
- M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST SP 800-38D, 2007.
- NIST. *FIPS 197: Advanced Encryption Standard (AES)*.
- NIST. *FIPS 180-4: Secure Hash Standard (SHA-256)* and *FIPS 202: SHA-3 Standard (SHA3-512)*, used to derive the Pedersen generator H).
- S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032, 2017.
- W3C. *Web Cryptography API*. <https://www.w3.org/TR/WebCryptoAPI/>

Solana platform

- Solana. *Larger transaction sizes* (v1 transactions, 4,096 bytes). <https://solana.com/upgrades/larger-transaction-sizes>
- Solana. *Accounts* (rent and account limits). <https://solana.com/docs/core/accounts>
- Anza. *Agave execution budget* (syscall costs). https://github.com/anza-xyz/agave/blob/master/program-runtime/src/execution_budget.rs
- Anza. *Agave feature set*. <https://github.com/anza-xyz/agave/blob/master/feature-set/src/lib.rs>
- Solana Program docs. *Confidential Balances: encryption*. <https://www.solana-program.com/docs/confidential-balances/encryption>

Migration context

- NIST. *IR 8547 (initial public draft): Transition to Post-Quantum Cryptography Standards*. November 2024. <https://csrc.nist.gov/pubs/ir/8547/ipd>
- EU. *Coordinated Implementation Roadmap for the Transition to Post-Quantum Cryptography*. June 2025. <https://digital-strategy.ec.europa.eu/en/library/coordinated-implementation-roadmap-transition-post-quantum-cryptography>