Privium

The omnibus account for Solana traders

Trade Solana. Leave no wallet to follow.

Whitepaper v1.0 · November 2026 · privium.io

Contents

01 Introduction

02 System overview

03 Cryptographic foundations

04 The confidential Ledger

05 Trusted execution and attestation

06 Threshold keys and the Escape

07 The Crossing engine

08 Unlinkability

09 Solana program engineering

10 The Coprocessor and the off-chain stack

11 Selective disclosure and Receipts

12 Economics and $PRV

Chapter 1

Introduction

Privium is the omnibus account for Solana traders. Traders deposit into one shared Pool, hold their balances in an encrypted Ledger, and submit encrypted Intents that a trusted-execution Coprocessor crosses against each other at the oracle mid every batch. Only the net Residual of each pair reaches public liquidity, as one Jupiter swap from the Pool, and each batch ends with a Commit that the Solana program accepts after checking the enclave's signature and the conservation of every asset. This chapter states the problem Privium solves, the structure it borrows from equity brokerage, the two recent pieces of work that make that structure buildable on Solana, the market it serves, and what the rest of this paper contributes.

Every trader is a feed

A Solana wallet is a public, append-only record of everything its owner has done with it. Every swap names the signer, the input and output mints, the amounts and the slot. Indexers decode these records within moments of confirmation, and a whole category of software is built on reading them back. The terminals that dominate Solana spot trading sell wallet tracking as a headline feature: Axiom offers unlimited tracked wallets with per-wallet PnL, win rate, trade history and real-time buy and sell alerts, and its copy-trading engine mirrors a tracked wallet's buys with fixed-SOL or percentage sizing [5]. A trader with a good record therefore publishes, with every fill, a signal that other traders' software is configured to act on.

The consequences compound. A wallet that is followed is front-run by its followers on entry and dumped on by them at exit, because the followers' bots react to the same on-chain event at machine speed and compete for the same thin liquidity. A wallet that is merely visible leaks its size, its holding period and its preferred venues to anyone who wants to price against it. And a swap that is visible before it lands can be sandwiched: a measurement study of Jito bundles over four months of early 2025 found 521,903 sandwich attacks costing victims more than $7.7M, and found users spending more than $2.4M on defensive bundling to avoid them [8]. Those figures are lower bounds by the authors' own method, since they count only canonical three-transaction bundles and only SOL-denominated legs [8].

Traders already pay to escape this. The standard practice among active Solana traders is to rotate burner wallets: fund a fresh address, trade from it until it becomes known, abandon it, repeat. Rotation is a manual, lossy imitation of a property that equity markets give every client by default. It costs transfer fees and attention, it fragments a trader's history across addresses they then cannot prove they own, and it fails the moment the funding path is traced, because each fresh wallet is funded from somewhere.

We can state the problem precisely. Let a trader u submit a sequence of trades $(i_1, i_2, \dots)$. On a public chain the observable trace of u is

$$\mathcal{O}_{\text{pub}}(u) = \big((\mathsf{addr}_u, \pi_j, s_j, x_j, \text{slot}_j)\big)_{j \geq 1}, \tag{1.1}$$

one record per trade, each carrying the same address addr_u. Any function of a trader's strategy that can be estimated from (1.1) is public. The aim of Privium is to replace (1.1) with a trace in which no record carries a trader at all. Under Privium the observer sees, per Crossing k and per pair π,

$$\mathcal{O}^{(k)}_{\text{Priv}} = \big(\rho^{(k)},\ \{R^{(k)}_\pi\}_\pi,\ \{V^{(k)}_a\}_{a \in \mathcal{A}}\big), \tag{1.2}$$

that is, the new Ledger root, one net Residual per pair, and the vault balances that follow from deposits, Exits and routed Residuals. No component of (1.2) is indexed by u. Deposits and Exits remain visible events at the boundary of the Pool, and Chapter 8 measures exactly how much they reveal and how Privium reduces it.

The omnibus account

Equity markets met this problem long ago and settled it with an account structure. In an omnibus account, a broker holds the cash and securities of many clients in a single account at the clearing firm and keeps the per-client records on its own books [7]. US broker-dealer rules treat such an account as one customer of the carrying firm, while the per-client records stay with the broker that maintains it [14]. The market sees the broker. It does not see which client bought, how much, or when.

The omnibus structure carries a second mechanism that matters as much as the first. A broker that holds many clients' order flow can cross opposing orders internally, at or inside the best quote, before anything reaches an exchange, and route only the remainder outward. In 2025, trading reported to the US Trade Reporting Facilities, that is, trading away from the lit exchanges, rose 361 basis points to 50.6% of total consolidated US equity volume, the first year off-exchange trading exceeded half the market [6]. More than half of the volume in the deepest equity market in the world now executes away from the exchanges, much of it against the internal books of the firms that hold the orders.

The structure won for two reasons that carry directly to Solana. Clients receive price improvement, because a crossed fill at the midpoint pays neither the spread nor the price impact of walking a book. Clients receive information protection, because the market sees aggregated, netted flow rather than each client's intent. The omnibus account delivers both from the same fact: many clients' orders meet inside one account before any of them meets the market.

Three things kept this structure off public chains. First, a broker's books are private because they are held by a firm, and a public chain has no private books. Second, a broker's client claims are enforced by law and by the custodian; a protocol needs a claim that is enforced by code even if its operator disappears. Third, internal crossing needs a computation over every client's private order, and that computation must be both hidden from the public and verifiable by it. Each of these was an open engineering problem on Solana until recently. The next section describes the two pieces of work that closed them.

The thesis from two sources

Private balances as a native primitive

Helius's "Solana Privacy Rings" [1] argues that public-ledger transparency blocks ordinary financial use of Solana, and that privacy is the chain's next frontier. It introduces Rings: programmable shielded pools whose balances are encrypted on chain and whose transactions execute natively on Solana and are verified by zero-knowledge proofs [1]. A permissionless default Ring is live on mainnet, encrypting assets and amounts while wallets remain visible, with balances readable only by the owner through a viewing key [1]. Rings accept SOL, SPL tokens and Token-2022 tokens, keep the owner's signature as a requirement for every transaction, and store their encrypted data on Solana's own ledger [1].

The article also specifies Custom Rings, aimed at institutions and targeted for mainnet in Q4 2026 [1]. A Custom Ring can be configured as confidential (counterparties visible, balances encrypted) or anonymous (both encrypted), can grant an auditor key read access under a configured policy, and lets a user disclose all of their activity through a viewing key or a single transaction through a transaction viewing key [1]. Helius lists Jupiter among the integrations in progress [1].

For Privium, Rings supply the boundary of the Pool. A trader can fund a Privium deposit from a Ring and take an Exit into one, so that neither edge of a trader's activity needs to touch a public balance. Rings also fix the vocabulary of selective disclosure on Solana: the viewing key and the per-transaction disclosure key are already familiar objects to wallets and compliance providers, and Privium's Viewing keys and Receipts (Chapter 11) are designed to sit beside them.

The same period restored the lower layer that Solana's confidential token tooling rests on. Token-2022 confidential transfers, which hide balances and transfer amounts using the ZK ElGamal proof program, were disabled on mainnet in June 2025 after researchers found a flaw in that program's Fiat-Shamir transformation; the fix shipped in Agave v2.1.21 in March 2026, and the feature gate was reopened on 17 June 2026 [9]. Chapter 3 treats the twisted ElGamal and sigma-protocol machinery of that stack in full.

Programs over private balances at trading speed

A private balance is not yet a private trading account. Crossing two traders' orders requires reading both orders, comparing them with a price, and writing both balances, all without revealing any of it. Lokhande et al.'s Encifher [2] is a confidentiality coprocessor for Solana that makes this kind of computation practical. It takes the ciphertext-handle and symbolic-execution model of homomorphic coprocessors such as Zama's fhEVM and runs it inside a trusted execution environment instead of a homomorphic evaluator [2]. On-chain programs manipulate only 128-bit ciphertext handles; an off-chain coprocessor in a hardware enclave fetches the ciphertexts from a public data availability layer, decrypts and computes on them inside the enclave, re-encrypts the results, and posts attested, Merkle-anchored outputs back on chain [2].

Two features of Encifher's design are central to Privium. The first is that Solana's account model already supplies the dependency graph of the confidential computation: because every transaction declares its read and write sets, the Sealevel scheduler orders encrypted operations correctly and in parallel without a custom scheduler [2]. The second is that no single enclave holds the only copy of the decryption capability: trust is spread across a threshold-decryption committee, and enclave attestation is checked on chain [2]. Encifher states its trust base plainly as TEE integrity, honest threshold key management and the cloud attestation root [2]. Privium adopts that trust base and states each of its guarantees against it (Chapter 2).

The reported production figures place this approach in the cost range of ordinary trading. A confidential swap settles in one Solana transaction at 157,000 compute units; the AES-256-GCM encryption layer exceeds one million operations per second; threshold decryption takes single-digit to tens of milliseconds; and the system has served more than 5,000 users and 50,000 confidential operations in production, across confidential payments, swaps and a cross-chain bridge [2].

What neither source builds

Rings make a private balance a native object. Encifher makes a program over private balances fast enough for trading. Neither carries the two to the place where a trader's information actually leaks, which is the trading account itself. A Ring hides what a wallet holds, but a trade from that wallet still lands as that wallet's swap, and an Encifher swap is a confidential swap between parties rather than a shared account in which many traders' flow is netted before it meets the market.

Privium composes them into the omnibus account. The Pool is one program-owned account per asset. The Ledger is Encifher-style encrypted state, one ciphertext leaf per trader and asset, anchored by a Merkle root. The Coprocessor is an attested enclave that does what a broker's internalization desk does: it decrypts the batch's Intents, crosses opposing flow at the oracle mid, computes one Residual per pair, and routes it. The Council plays the role that law and the custodian play for a broker's client: it holds threshold shares of the decryption key and runs the Escape if the Coprocessor stops, so that every trader's claim to their last committed balance is enforced without the operator. Rings sit at the boundary, so that deposits and Exits can be unlinked as well.

Why now

The structure needs five platform capabilities at once. Each is now available on Solana mainnet, and the first two arrived in 2026.

Capability Privium needs	What provides it	Status
Encrypted balances and private deposits and exits	Helius Rings; Token-2022 confidential transfers on the ZK ElGamal proof program	Default Ring live on mainnet; Custom Rings targeted for Q4 2026 [1]; confidential transfers re-enabled 17 June 2026 [9]
Computation over encrypted state at trading cost	TEE coprocessor with ciphertext handles and on-chain attestation checks	In production, 157,000 CU per confidential swap [2]
Correct ordering of confidential state transitions	Sealevel's declared read and write sets	Used as the confidential dependency graph [2]
One swap from the Pool across all Solana liquidity	Jupiter Swap API V2, whose Router path `/swap/v2/build` returns instructions for custom transactions	Current API generation [15]
Enclave signatures checked by the program	Solana's native signature verification programs, read by instruction introspection	Chapter 5 and Chapter 9

The table also explains why the structure could not have been built earlier from any single source. Without encrypted balances, an omnibus Pool on a public chain is only a commingled account whose per-trader state must live somewhere off chain and in the clear. Without an attested coprocessor, internal crossing requires either a trusted operator who sees every order or a cryptographic protocol (fully homomorphic encryption or secure multi-party computation) whose latency Encifher's authors judge too high for interactive use [2]. Without a liquidity aggregator that composes into a program's own transaction, the Residual could not be executed from the Pool's vaults as one atomic, enclave-bounded step whose fill the Commit then absorbs. Chapter 9 gives the instruction-level construction of that step.

Market evidence

The venue

Solana is the largest spot venue in crypto by DEX volume. Its decentralized exchanges cleared $197B of spot volume in Q3 2026, up 16.6% from Q2, leading every other chain for the fifth consecutive quarter [3]. In one week of September 2026 Solana processed about 208M DEX trades, against about 190M trades on the NYSE in the same week, a comparison of trade counts rather than dollar value [3]. DefiLlama's DEX dataset records $49.8B of Solana DEX volume over the 30 days to 15 November 2026 and $969.5B over the trailing twelve months [10].

The trade count matters as much as the dollar figure. Internal crossing produces value in proportion to how often opposing flow arrives in the same batch, and that frequency grows with the number of independent orders per unit of time. Chapter 7 defines the crossing rate, and Chapter 12 derives it as a function of order flow and turns it into an expected-cost comparison.

The terminals

The flow is concentrated in trading terminals, and those terminals are large businesses. DefiLlama records about $18.4M of fees and $8.4M of revenue for Axiom over the 30 days to 15 November 2026, and $475.8M of cumulative revenue [4]. Photon reports more than $40B of cumulative volume and 1.3 million users [5]. Axiom's base trading fee is 0.95%, falling to 0.75% at its top tier, and Photon's is 1% flat [5]. These fees are paid on top of the DEX fees and price impact of each fill, which is the cost Privium's internal crossing removes for the matched portion of every batch.

The same terminals are where the wallet-tracking problem is sold as a product. Wallet tracking with PnL and alerts, and copy trading of tracked wallets, are standard features of the leading terminal [5]. Every serious trader on these platforms is simultaneously a user of the tracking tools and a target of them.

The behaviour already exists

The demand for the omnibus account is visible in what traders already do without it. Burner-wallet rotation is the manual form. Defensive bundling, which the IMC '25 study measured at more than $2.4M of user spend over four months [8], is the form traders adopt against ordering attacks. MEV-protection settings in mainstream Solana trading apps, offered through Jito, are a third [8]. Each of these is a trader paying to reduce the information content of their own trace (1.1). Privium removes the trader from the trace instead.

The equity precedent gives the scale of the endpoint. When clients are offered an account that internalizes their flow, a majority of volume moves there: off-exchange trading reached 50.6% of US equity volume in 2025 [6]. Privium does not need that share to be valuable; it needs enough concurrent flow per batch for opposing Intents to meet, and Chapter 7 shows that the value of a Crossing to each participant grows with the flow of the others.

Landscape

Batch matching and private trading each have strong precedents, and Privium borrows from them openly.

CoW Protocol collects signed intents off chain and runs batch auctions in which solvers first look for coincidences of wants inside the batch, matching opposing orders peer to peer without touching on-chain liquidity, and source the remainder from on-chain and off-chain liquidity [11]. Privium's Crossing is the same economic move, with two differences of construction: the batch is computed inside an attested enclave over encrypted Intents rather than by competing solvers over public orders, and settlement is against a shared encrypted Ledger rather than per-order token transfers.

Penumbra batches swaps per trading pair in each block and writes the batch result to public state, with users claiming their pro-rata share of outputs; its sealed-bid batch swap design reveals only the net flow across a pair in each batch [12]. Privium's observable trace (1.2) has the same shape, one net quantity per pair per batch, delivered on Solana and against Solana's existing liquidity.

Renegade is an on-chain dark pool that matches orders with multi-party computation, proves settlement with zero-knowledge proofs, and executes crosses at the midpoint [13]. Privium shares the midpoint crossing rule and the commitment to pre-trade and post-trade privacy, and takes the coprocessor route to the computation, which Encifher has shown to run at Solana transaction cost [2].

Encifher itself runs confidential swaps on Solana [2]. Privium is built on its model and occupies a different place in the stack: not a confidential swap between parties, but the account a trader lives in, with one deposit, the whole token universe that Jupiter routes, internal crossing of everyone's flow, and no wallet to follow.

Contributions of this paper

This paper specifies Privium completely enough to build, audit and price. Its contributions are as follows.

An account structure. We define the omnibus account on Solana: a single Pool program custodying per-asset vaults, an encrypted per-trader Ledger committed as a Merkle root over ciphertext leaves, and a Coprocessor that advances the Ledger in batches. Chapter 2 gives the system and its properties; Chapter 4 gives the Ledger's state transition function.

A conservation theorem enforced on chain. We state and prove that for every asset and every committed root, the sum of Ledger balances equals the vault balance net of committed Exits not yet released and deposits not yet credited, and we show how the Solana program checks the public side of that identity on every Commit without seeing a single balance. Chapter 4 proves the theorem; Chapter 9 implements the check.

An attested Coprocessor with on-chain key binding. We bind the Coprocessor's signing key pkC to its enclave measurement through the report data of a remote attestation quote, register it on chain, and verify every Commit against it. Chapter 5 gives the chain of trust, rotation and upgrade by re-attestation.

A threshold Council and a forced Escape. We give a distributed key generation for a twisted ElGamal key, threshold re-encryption with proofs of correct decryption, proactive resharing, and an Escape protocol that pays every owner their last committed balance when no Commit lands within the liveness window W, with a safety theorem. Chapter 6 covers all of it.

A crossing engine with fairness and cost results. We give the netting rule per pair, clearing at the oracle mid $m_\pi^{(k)}$ under limit constraints, allocation rules with proofs of their fairness properties, an expected-cost inequality between crossing and routing, a Residual routing step bounded by a slippage limit the enclave signs, and an analysis of ordering attacks against batched Residuals versus individual swaps. Chapter 7.

An unlinkability analysis with a formal game. We model a chain observer, a wallet tracker and a timing analyst; measure the anonymity set with min-entropy H_∞ and Shannon entropy H as functions of Pool activity; bound deposit-to-exit, amount and

timing linkage; and prove the unlinkability theorem Privium achieves, including with Ring Exits. Chapter 8.

A complete Solana implementation design. We give account layouts with sizes, PDAs and seeds, every instruction with its checks, compute budgets, address lookup tables, write-lock design for Sealevel parallelism, signature verification by instruction introspection, and the Jupiter composition for the Residual (Chapter 9), and the Coprocessor service, oracle ingestion, indexer and the TypeScript SDK with its one-line call (Chapter 10).

Selective disclosure without a wallet. We define Viewing keys, per-transaction disclosure, and Receipts: enclave-attested statements of a trader's realized PnL over a period, verifiable by anyone and shareable without revealing an address. Chapter 11.

An economic model and a token. We price crossed, routed and direct execution as expected costs, derive the crossing rate's effect on cost, and specify the fee flow: each Commit pays the Crossing's fees to the token's launch wallet, and a crank buys $PRV on the market with them and burns what each purchase delivered. Chapter 12.

Notation and conventions

The notation below is shared by every chapter. Chapters that need further symbols define them where they first appear and keep them local.

Symbol	Meaning
$u \in \mathcal{U}$, $a \in \mathcal{A}$, q	trader, asset, quote asset (USDC by default)
$b_{u,a}^{(k)}$, $V_a^{(k)}$	Ledger balance of u in a after Crossing k; vault balance of a
k, Δ, W	Crossing index; batch interval in slots; liveness window in slots
$i = (u, \pi, s, x, \ell, \tau, \nu)$	Intent: trader, pair $\pi = (a, q)$, side $s \in \{+1, -1\}$, size $x > 0$ in base units, limit ℓ, expiry slot τ, nonce ν
$m_\pi^{(k)}$, $p_\pi^{(k)}$, $R_\pi^{(k)}$	oracle mid, clearing price, Residual for pair π in Crossing k
ϕ	fee rate, 0.10% (10 bps)
$\mathbb{G}$, $\ell_{\mathbb{G}}$, G, H	ristretto255 group, its order, generators
H, AE	hash function; AEAD (AES-256-GCM)
n, t	Council size and threshold
(sk_C, pk_C), $\rho^{(k)}$	Coprocessor signing key pair; Ledger root after Crossing k
$\lambda = 128$, $\mathsf{negl}(\lambda)$	security parameter; negligible function

The letter H denotes the hash function in protocol descriptions and Shannon entropy in the unlinkability analysis; context always fixes which. Amounts are integers in each asset's base units, and prices are quoted in the quote asset per unit of the base asset.

Privium design parameters appear with a recommended value marked "(parameter)" and a one-line reason, in the chapter that owns them. Engineering estimates are marked "est." and derived where they appear. Every other number is cited.

Reading guide

The paper is written for three readers, and each can take a direct path through it.

Reader	Path	What it covers
Investor or product reader	Chapters 1, 2, 7 (first two sections), 12	the structure, the life of a trade, why crossing is cheaper, fees and $PRV
Protocol engineer	Chapters 2, 4, 5, 9, 10	state, attestation, the Solana program, the Coprocessor and SDK
Cryptographer or auditor	Chapters 2, 3, 4, 6, 8, 11	primitives and reductions, conservation, threshold keys and the Escape, unlinkability, Receipts

Chapter 2 is the common entry point: it introduces every actor, follows one trade from deposit to Exit, draws the boundaries between trust domains, and states the six properties the rest of the paper proves. Chapters 3 to 6 build the confidential state and the keys that protect it. Chapters 7 and 8 analyse what Privium does with that state, first for price and then for privacy. Chapters 9 and 10 turn the design into code. Chapters 11 and 12 cover what a trader can prove about themselves and how the protocol is paid.

References

1. "Solana Privacy Rings", Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings
2. N. Lokhande, R. Gupta, R. Chahar, A. Jangra, A. Prajapati, M. Kumari, "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
3. V. Nguyen, "Solana leads all chains in spot DEX volume for fifth straight quarter with $197B in Q3 2026", Crypto Briefing, 2 October 2026. https://cryptobriefing.com/solana-leads-spot-dex-volume-q3-2026/
4. "Axiom: fees and revenue", DefiLlama (fees and revenue API, data to 15 November 2026), accessed 16 November 2026. https://defillama.com/protocol/axiom
5. "Photon vs Axiom: Solana Memecoin Terminal Comparison (2026)", MadeOnSol, 31 May 2026, updated 30 September 2026. https://madeonsol.com/blog/photon-vs-axiom-solana-trading-terminal
6. "2025 U.S. Equities Year in Review", Cboe Global Markets, 26 January 2026. https://www.cboe.com/insights/posts/2025-u-s-equities-year-in-review
7. "Omnibus account", Nasdaq Glossary, accessed 16 November 2026. https://www.nasdaq.com/glossary/o/omnibus-account
8. N. Gerzon, B. Weintraub, J. In, A. Mislove, C. Nita-Rotaru, "Quantifying the Threat of Sandwiching MEV on Jito: A Measurement of Solana's Leading Validator Client", Proceedings of the 2025 ACM Internet Measurement Conference (IMC '25), October 2025. https://doi.org/10.1145/3730567.3764493
9. "Confidential transfers return to Solana mainnet after year-long security pause", Solana Compass, 18 June 2026. https://solanacompass.com/news/confidential-transfers-return-to-solana-mainnet-after-year-long-security-pause
10. "DEX volume: Solana", DefiLlama (DEX overview API, data to 15 November 2026), accessed 16 November 2026. https://defillama.com/dexs/chain/solana
11. "Coincidence of Wants", CoW Protocol documentation, accessed 16 November 2026. https://docs.cow.fi/overview/coincidence-of-wants
12. "Batch Swaps", Penumbra Protocol Specification, accessed 16 November 2026. https://protocol.penumbra.zone/main/dex/swap.html
13. "Dark pool explainer", Renegade documentation, accessed 16 November 2026. https://docs-v2.renegade.fi/concepts/dark-pool-explainer
14. "SEA Rule 15c3-3 and Related Interpretations", FINRA, accessed 16 November 2026. https://www.finra.org/rules-guidance/guidance/interpretations-financial-operational-rules/sea-rule-15c3-3-and-related-interpretations
15. "Swap API", Jupiter Developer Docs, accessed 16 November 2026. https://developers.jup.ag/docs/swap-api

Chapter 2

System overview

This chapter describes Privium as a whole before the following chapters take it apart. It introduces each actor and what that actor wants from the system, follows one trade from deposit to Exit, lays out where every piece of state lives and which trust domain protects it, and states the six properties that the rest of the paper proves: custody, confidentiality, unlinkability, liveness, price fairness and auditability by the owner.

Actors

Privium has five kinds of participant that act on the protocol and two that only observe it. The table gives each one's goal and its means; the prose explains how the design serves the three actors that hold keys, and the observer and verifier are treated in Chapters 8 and 11.

Actor	Wants	Holds	Can do
Trader u	to trade any Jupiter-routable token without publishing a trace; to keep a claim on funds that no operator can revoke	a wallet key and the keys derived from it: an Intent authority key, a viewing root for their own leaves, an escape key	deposit, submit Intents, request Exits, trigger the Escape, issue Viewing keys and Receipts
Coprocessor	to advance the Ledger correctly each Crossing and earn the protocol fee	skC and the HPKE key dkC inside the enclave; the Ledger data key k_L	decrypt Intents and leaves, cross, sign Residual orders, sign and submit Commits
Council member	to keep the Council key safe and to honour Escapes	one share of the Council secret s	publish re-encryption shares with proofs, take part in resharing
Pool program	to keep every vault equal to what the Ledger owes	the vault token accounts, the Ledger root, the Registry	accept or reject Commits, release Exits, run the Escape
Liquidity venues	to fill the Residual at a market price	public pools reachable through Jupiter	execute the Residual swap
Observer	to infer which trader did what	the full public chain	read every account and transaction
Receipt verifier	to check a claim a trader makes about their results	a Receipt and the Registry	verify signatures and inclusion

The **trader** wants two things that pull in different directions on a public chain: privacy of their trading and a claim on their funds that does not depend on anyone's good behaviour. Privium serves the first by never letting a trader's order appear as a transaction of their own, and the second by keeping custody in a program whose only release paths are the Exit, which needs the trader's signature, and the Escape, which needs the Council and pays only the owner.

The **Coprocessor** is an enclave service, run by an operator, that executes the Crossing. Its operator controls the host, the network and the schedule, and is trusted for none of them: the Pool program accepts a Commit only if it carries a signature under the registered key pkC, and that key is bound by remote attestation to a published enclave measurement (Chapter 5). Each Commit pays the protocol fee at rate ϕ on crossed and routed notional to the token's launch wallet, where it buys $PRV on the market and is burned (Chapter 12).

The **Council** is a set of n members holding Shamir shares, with threshold t, of the secret behind the Council key $P_{\mathcal{C}}$, under which the Ledger data key and every leaf's Council handle are encrypted. Its members never see a trader's balance in ordinary operation. They are called on in two situations: when a new Coprocessor enclave must be provisioned with the key, and when the Escape is triggered because no Commit has landed within the liveness window W. In both cases the output of their computation is bound to a destination, the attested enclave or the balance's owner, and is accompanied by proofs of correct decryption (Chapter 6).

Two external services feed the Coprocessor and are trusted only within narrow bounds. The **oracle** supplies the mid $m_\pi^{(k)}$ for each pair; Privium reads Pyth price feeds, which publish a confidence value alongside each price and recommend treating it as a range around the price [3], and the Coprocessor refuses to cross a pair whose confidence interval is too wide relative to its

price (Chapter 7). **Jupiter** supplies the route for the Residual; its Swap API V2 Router path `/swap/v2/build` returns swap instructions for a transaction the integrator composes [4], which is what lets the Residual execute from the Pool inside Privium's own instruction flow (Chapter 9).

State

Privium's state is divided between the chain and the enclave's memory. Every byte that determines a trader's claim is on chain: in accounts, or in the ciphertext diffs that Commits publish as transaction data. The enclave holds plaintext only transiently.

On-chain state

Account	Contents	Written by
Vault (one per asset a)	SPL or Token-2022 token account holding $V_a^{(k)}$, owned by the Pool's program-derived authority	deposits, Exits, the Residual swap, the Escape
Ledger root	Crossing index k, root $\rho^{(k)}$, slot of the last Commit, data availability digest, recent exits roots	Commit
Crossing state	the open Crossing's verified mids, recorded Residual fills and ciphertext diff accumulator	`open_crossing`, `route_residual`, `log_diff`
Registry	admitted measurements; attested keys pk_C, ek_C, Y_E with validity intervals	registration and rotation (Chapter 5)
Council config	n, t, the Council key P_C, member verification keys	DKG and resharing (Chapter 6)
Asset state and deposit queues	committed total and unreleased Exits per asset; deposits not yet credited, in public amounts	deposits, Commits, Exits
Escape state	whether the Escape is active, the open Crossing's fills, and which entries have been paid	Escape instructions

The vault balances are public by nature: they are ordinary token accounts. What Privium hides is the partition of each vault among traders. Deposits and Exits move public amounts across the boundary of the Pool, and are therefore recorded in the deposit queues and exits roots in the clear; the Coprocessor folds deposits into the encrypted Ledger at the next Crossing. Chapter 8 treats the information content of these boundary events and the countermeasures that reduce it, including amount denominations and Ring Exits.

The Ledger

The Ledger is the encrypted mapping $(u, a) \mapsto b_{u,a}^{(k)}$. Each entry is a leaf carrying a Pedersen commitment to the balance, a Council handle, and AEAD ciphertexts for the enclave and the owner, and the Ledger root $\rho^{(k)}$ is the Merkle root over all leaves after Crossing k. Following Encifher, on-chain state refers to encrypted values by short ciphertext handles [1]; Privium publishes the ciphertexts themselves on Solana, hash-chained by the program. Chapter 4 defines the leaf structure, the handle model, the state transition function and the growth of the Ledger with the number of traders and assets.

Off-chain state

The Coprocessor holds three things off chain. The Intent queue is the set of encrypted Intents received for the current Crossing; each is encrypted to an enclave key bound to the attestation (Chapter 3 gives the HPKE construction). The working set is the plaintext of the leaves touched by the current Crossing, held in enclave memory for the duration of one batch. The key state is the epoch keys sk_C, dk_C and y_E, generated inside the enclave and bound to its measurement, and the Ledger data key k_L, which the Council releases only to admitted enclaves (Chapter 5). None of these is needed to recover a trader's funds: if the Coprocessor disappears, the Escape works from on-chain state alone.

The life of a trade

We follow one trade from the moment a trader funds the Pool to the moment the proceeds land at a fresh address. The pair is $\pi = (a, q)$ with a the base asset and q the quote asset.

Deposit

The trader u sends d units of q to the Pool's vault for q with a `deposit` instruction. The instruction transfers the tokens and appends (q, d, note) to a deposit queue, where note seals the trader's record identifier to the Coprocessor's HPKE key, so the public record shows an amount entering the vault and never which leaf it credits. The tokens can come from a public wallet or be withdrawn from a Ring [2]; Chapter 8 compares what each funding path reveals. At the next Crossing the Coprocessor credits the leaf, so that

$$b_{u,q}^{(k)} = b_{u,q}^{(k-1)} + d. \tag{2.1}$$

Intent

To buy x units of a at a price no worse than ℓ, the trader builds the Intent $i = (u, \pi, +1, x, \ell, \tau, \nu)$, with expiry slot τ and a fresh nonce ν, encrypts it to the Coprocessor's attested encryption key, and submits the ciphertext. Submission is off chain to the Coprocessor's endpoint, or on chain as an opaque blob in an account the Coprocessor reads; in both cases the ciphertext reveals neither pair, side, size nor limit. The nonce makes each Intent unique and lets the Ledger reject a replay (Chapter 4).

The listing gives the plaintext form of an Intent as the Coprocessor sees it after decryption, together with the public header of a Commit. The field names follow the shared notation.

```rust
/// Plaintext Intent, visible only inside the enclave.
pub struct Intent {
    pub trader: LeafId,      // u, as a Ledger leaf identifier
    pub pair: PairId,        // pi = (a, q)
    pub side: Side,          // s: Buy = +1, Sell = -1
    pub size: u64,           // x, base units of a
    pub limit: Price,        // l, quote units per base unit
    pub expiry_slot: u64,    // tau
    pub nonce: [u8; 16],     // nu
    pub auth: [u8; 64],      // signature by the trader's authority key A_u
}

#[repr(i8)]
pub enum Side { Buy = 1, Sell = -1 }

/// Public header of a Commit, checked by the Pool program (layout in Chapter 9).
pub struct CommitHeader {
    pub crossing: u64,              // k
    pub prev_root: [u8; 32],        // rho^(k-1)
    pub new_root: [u8; 32],         // rho^(k)
    pub da_digest: [u8; 32],        // accumulator over the ciphertext diff of k
    pub exits_root: [u8; 32],       // Exits listed by this Commit
    pub fills_hash: [u8; 32],       // Residual fills recorded on chain for k
    pub assets: Vec<AssetLine>,     // per asset: total, exits, fees, cuts, commitment delta
}

/// Signed by the enclave before each Residual leg executes.
pub struct ResidualOrder {
    pub crossing: u64,       // k
    pub pair: PairId,        // pi, routed in the direction of sgn R^(k)_pi
    pub amount_in: u64,      // exact input from vault 0
    pub min_out: u64,        // slippage bound checked on the measured fill
}
```

The `auth` field is the signature of the trader's authority key A_u, derived from their wallet (Chapter 10), over the Intent's fields, so that the Coprocessor executes only Intents their owner authorized. `CommitHeader` contains no trader, no balance and no individual size: everything in it is either a root, a hash, a counter or a per-pair aggregate.

Crossing

Once the Coprocessor holds something to settle and Δ slots have passed since the last Commit, it closes the batch and runs one Crossing (Chapter 7 fixes the cutoff, Chapter 10 what counts as work). It decrypts the Intents in the queue, drops any whose expiry τ has passed or whose nonce has been seen, and groups the remainder by pair. For each pair it reads the oracle mid $m_\pi^{(k)}$ and computes a fill $f_i \in [0, x_i]$ for every eligible Intent under the limit constraints and the allocation rule of Chapter 7. The Residual is the signed net of the fills,

$$R_\pi^{(k)} = \sum_{i \in \mathcal{I}_\pi^{(k)}} s_i \, f_i, \tag{2.2}$$

where $\mathcal{I}_\pi^{(k)}$ is the set of eligible Intents on π in Crossing k. The portion of buy flow matched by sell flow, $\min\left(\sum_{s_i=+1} f_i,\ \sum_{s_i=-1} f_i\right)$, is crossed internally at the clearing price $p_\pi^{(k)} = m_\pi^{(k)}$. It pays no DEX fee and moves no market. Only $|R_\pi^{(k)}|$ units of a are routed, in the direction of $\operatorname{sgn} R_\pi^{(k)}$.

If our trader's buy meets a sell of equal or larger size in the same Crossing, the trader is filled entirely at the mid. If the batch is net long, the trader's fill is completed from the Residual swap at the Residual's execution price, and Chapter 7 specifies how that price is blended across the participants on the routed side. In both cases each leaf touched changes by

$$\Delta b_{u,a} = s_i f_i, \qquad \Delta b_{u,q} = -\,s_i f_i \bar{p}_i \;-\; \phi\, f_i \bar{p}_i, \tag{2.3}$$

where $\bar{p}_i$ is the trader's average fill price over the crossed and routed portions and ϕ is the protocol fee rate.

The Coprocessor then builds the Residual swap. It requests a route from Jupiter for $|R_\pi^{(k)}|$ units, fixes a minimum output from the oracle mid and a slippage bound, signs both into a `ResidualOrder`, and submits a `route_residual` transaction in which the Pool's authority is the swap's signer [4]. The program measures the vault deltas, rejects a fill outside the signed bound, and records the fill on chain for the Crossing. The Residual of every pair executes from the Pool's vaults; no trader's wallet is involved.

Commit

The Coprocessor re-encrypts every touched leaf with its decoys, publishes the new ciphertexts as `log_diff` transaction data, computes $\rho^{(k)}$, and signs the Commit message:

$$\sigma^{(k)} = \mathsf{Sign}_{sk_C}\left(M_k\right), \qquad M_k = \texttt{privium/commit/v1} \,\|\, \mathsf{id}_{\mathrm{prog}} \,\|\, e \,\|\, k \,\|\, \rho^{(k-1)} \,\|\, \rho^{(k)} \,\|\, \mathsf{body}_k, \tag{2.4}$$

with body_k the header fields above (Chapter 4 defines M_k, Chapter 9 lays it out). The Pool program accepts the Commit only if its checks pass: the signature verifies under an admitted pk_C that opened the Crossing; `prev_root` equals the stored root and `crossing` equals the stored index plus one; the data availability digest, mids and fills match what the chain recorded; and for every asset, once the Commit has paid the Crossing's fees to the fee recipient, the vault holds exactly the attested Ledger total plus the committed Exits not yet released and the deposits not yet credited, with the Pedersen commitments of the leaves summing to that total. These checks are the public face of the conservation theorem that Chapter 4 proves. The program never sees a balance, and it does not need to.

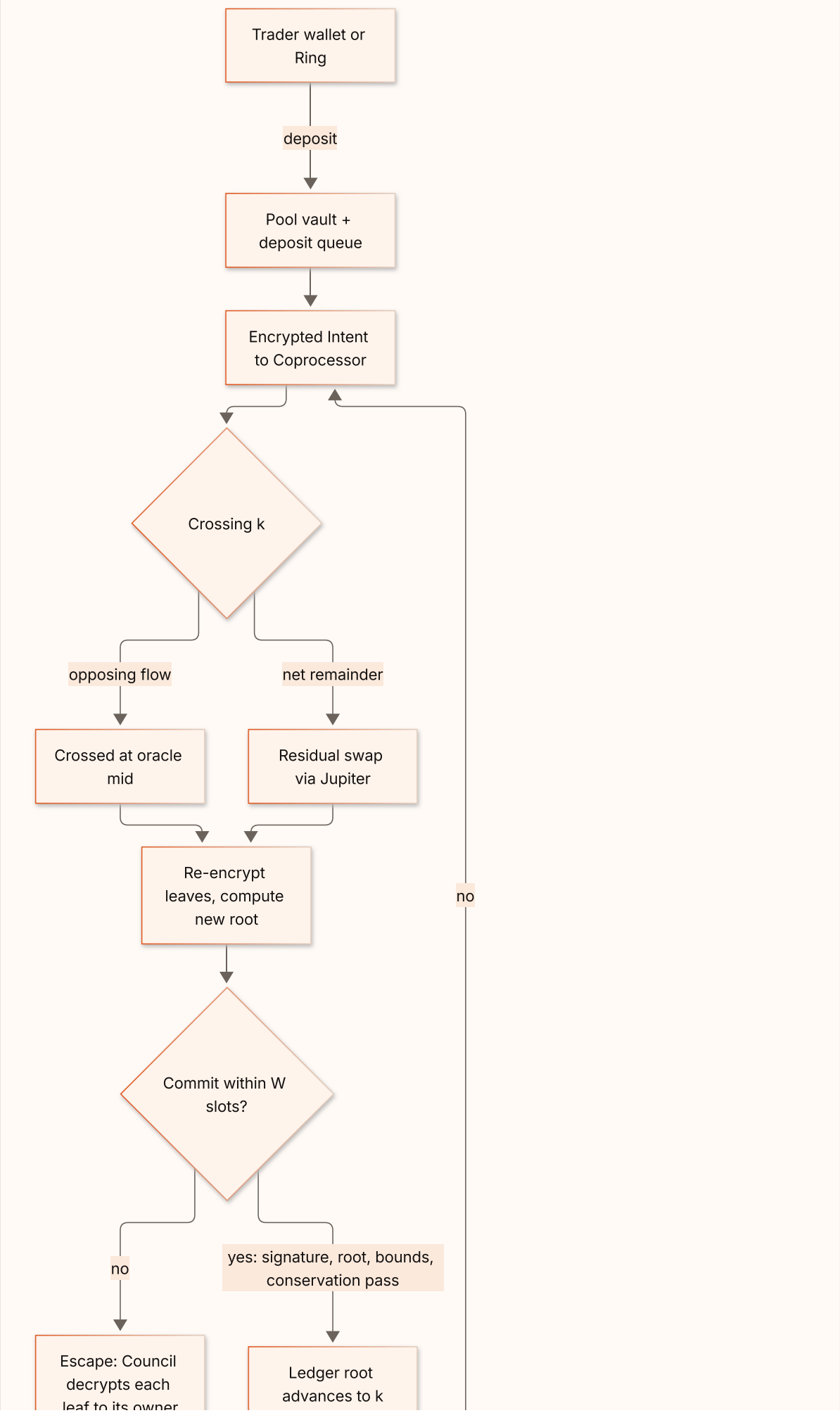
Each stage is atomic. A Residual leg outside its bound reverts and records nothing; a Commit that fails any check reverts, the root stays at $\rho^{(k-1)}$, and the Coprocessor resubmits it with the fills already on chain (Chapter 10). The Ledger credits exactly the fills the program recorded, so no trader's entry ever reflects a swap that did not execute.

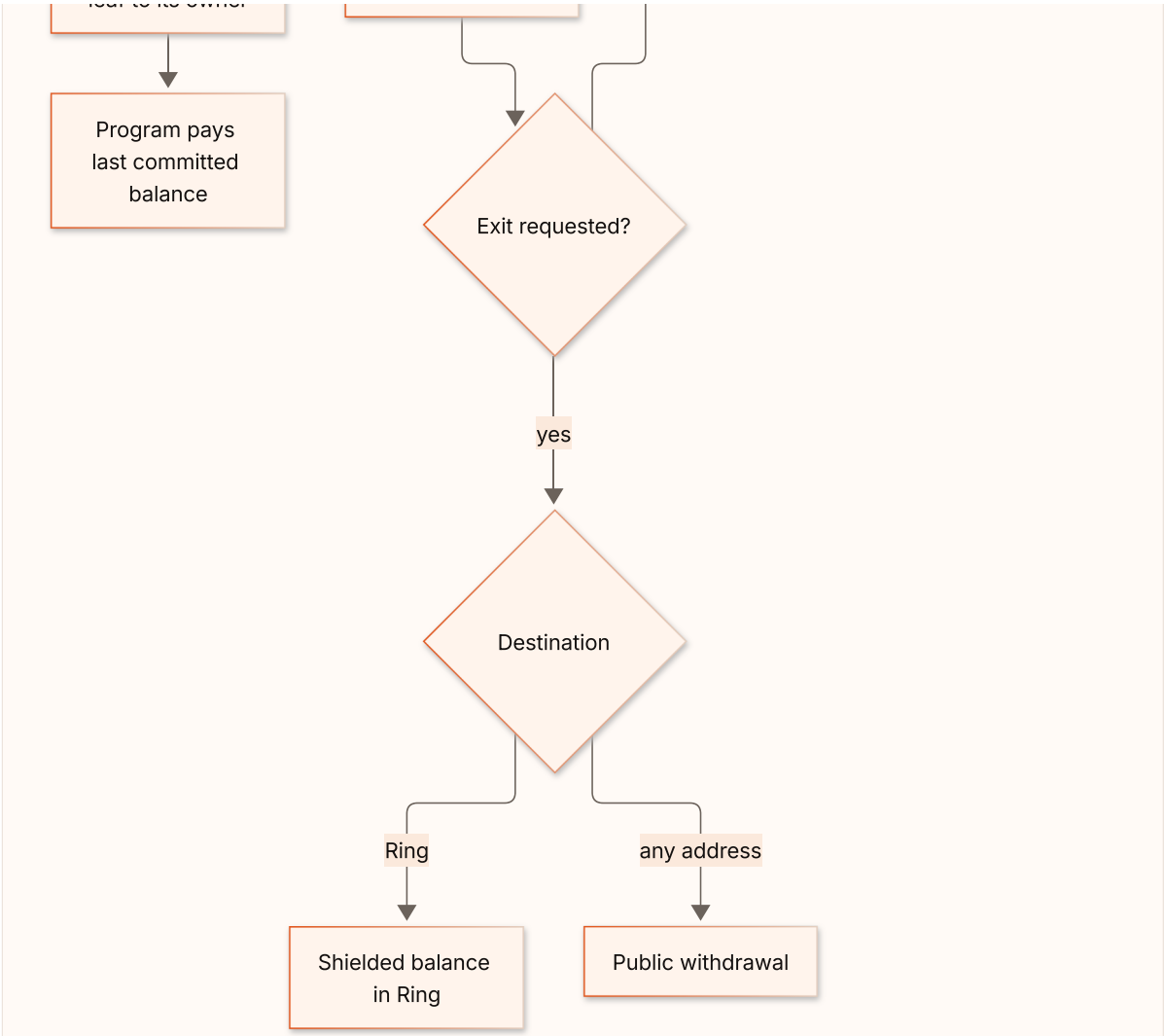
Exit

To withdraw, the trader signs an Exit request for an amount w of asset a to a destination address of their choice. The request travels sealed to the Coprocessor like an Intent; once its delay has passed (Chapter 8) the Coprocessor checks $b_{u,a}^{(k-1)} \geq w$ inside the enclave, debits the leaf, and lists the Exit in the Commit's exits root; a permissionless `release_exit` then pays w from the vault to the destination against that root (Chapter 9). The destination can be any address, including a fresh one or a Ring. An Exit into a Ring lands in a shielded balance whose assets and amounts are encrypted from then on [2]. Chapter 8 analyses Exit timing, denominations and delays as countermeasures against linkage.

Escape

If no Commit lands for W slots, any trader can call `trigger_escape`. The program freezes the Ledger at the last committed root $\rho^{(k^\star)}$ and opens the Escape. Each owner submits a claim with their leaf's inclusion proof under $\rho^{(k^\star)}$; Council members publish decryption shares for that leaf, re-encrypted to the owner's key, with proofs of correct decryption; once t shares are verified, the owner learns the balance and the program pays $b_{u,a}^{(k^\star)}$ to the owner's chosen address after checking a proof that the claimed amount matches the ciphertext, settling the fills of a Crossing left open at the halt across the holders of the assets it spent. Chapter 6 gives the protocol step by step and proves that the Escape pays every owner exactly their last committed balance and pays nobody else.


Trader wallet or Ring
deposit
Pool vault + deposit queue
Encrypted Intent to Coprocessor
Crossing k
opposing flow
net remainder
Crossed at oracle mid
Residual swap via Jupiter
Re-encrypt leaves, compute new root
Commit within W slots?
no
yes: signature, root, bounds, conservation pass
Escape: Council decrypts each
Ledger root advances to k
no



Trust domains

Privium's guarantees rest on five domains, each trusted for a specific statement and bounded by checks from the others. The table lists them; the paragraphs below describe the bound on each.

Domain	Trusted for	Bounded by
Solana consensus and runtime	executing the Pool program as written; ordering transactions	the chain's own security
Coprocessor enclave	correct decryption, crossing arithmetic and re-encryption; secrecy of plaintext in use	remote attestation of the measurement; on-chain signature and conservation checks; the Escape
Council	secrecy of the Ledger key below threshold; release of shares only to the attested enclave or the owner	threshold t; proofs of correct decryption; proactive resharing
Oracle	a mid price within its published confidence	the limit price ℓ of each Intent; the confidence filter
Jupiter and venues	executing the Residual at a market price	the enclave-signed `min_out` checked by the program

The **enclave** is trusted for computation and for confidentiality of plaintext while it is in use. Its operator is not trusted for anything. Encifher states the same trust base for its coprocessor: TEE integrity, honest threshold key management and the cloud attestation root [1]. What the enclave can and cannot do is bounded from both sides. It cannot move a token the public checks do not account for, because the program verifies vault conservation against public flows on every Commit. It cannot

sign under any key other than the one bound to its measurement, because the Registry accepts only attested keys (Chapter 5). And it cannot hold traders' funds hostage, because its silence for W slots opens the Escape.

The **Council** is trusted below threshold: fewer than t members together learn nothing about the Council secret or the Ledger data key, and t honest members suffice to complete an Escape. Every share a member publishes is accompanied by a proof that it is the correct partial decryption under that member's committed key share, so a dishonest member can withhold but cannot corrupt (Chapter 6).

The **oracle** is trusted only to the extent that a trader's own limit lets it be. A crossed fill happens at $m_\pi^{(k)}$ only if that price satisfies the limit of every Intent it fills, so a wrong mid can cause a fill the trader would have accepted anyway at that price, or no fill, but never a fill outside the trader's limit. The confidence filter keeps the Coprocessor from crossing at all when the oracle's own uncertainty is wide.

The keys are separated along the same lines. The trader's wallet key signs deposits and one message from which the SDK derives the rest (Chapter 10): the authority key A_u that signs Intents and Exit requests, the viewing root vk_u that decrypts the trader's own leaves (Chapter 11), and the escape key y_u that claims an Escape payout. The Ledger data key k_L exists only inside admitted enclaves, and the Council secret behind $P_{\mathcal{C}}$ exists only as t-of-n shares. The Coprocessor's epoch keys exist only inside the enclave that generated them, bound to its measurement and registered with a validity interval (Chapter 5). A Viewing key, when a trader issues one, decrypts that trader's leaves and nothing else (Chapter 11).

Design goals as properties

We state Privium's design goals as properties of the system. Each statement names the assumption it rests on and the chapter that proves it. The precise games and definitions are given in the owning chapters; here we fix what each property says.

Property 2.1 (Custody). For every asset a and every committed Crossing k,

$$\sum_{u \in \mathcal{U}} b_{u,a}^{(k)} \;=\; V_a^{(k)} \;-\; \Pi_a^{(k)}, \tag{2.5}$$

where $\Pi_a^{(k)}$ is the part of the vault the Ledger does not owe traders at root $\rho^{(k)}$: committed Exits not yet released and deposits not yet credited (Chapter 4). Every Residual fill of Crossing k is absorbed by Commit k, so none is outstanding at a committed root. Moreover, tokens leave a vault only through an Exit authorized by the balance's owner, a Residual swap that satisfies its signed bound, or an Escape payment to the balance's owner. *Rests on:* Solana's execution of the Pool program and the integrity of the enclave's arithmetic; the second sentence rests on the program alone. *Proved in:* Chapter 4 (conservation theorem) and Chapter 9 (the program's checks).

The release rule is enforced by code anyone can read and holds whatever the enclave does; the equality (2.5) concerns the partition of each vault among traders, which only the enclave sees, while the public check on every Commit keeps the vault itself equal to the public flows.

Property 2.2 (Confidentiality). For every trader u, asset a and Crossing k, an adversary that sees all on-chain state, all ciphertexts, all encrypted Intents, and the shares of fewer than t Council members distinguishes the Ledger and Intent contents of any two executions with the same public trace (1.2) of Chapter 1 with advantage at most $\mathsf{negl}(\lambda)$. *Rests on:* the integrity and confidentiality of the enclave, the IND-CCA2 security of the Intent encryption and of AE, the semantic security of the Ledger ciphertexts, and an honest threshold of the Council. *Proved in:* Chapter 3 (reductions) and Chapter 4 (the Ledger).

Property 2.3 (Unlinkability). Within a Crossing, an observer cannot attribute any Intent, fill or balance change to a trader beyond what the boundary events (deposits and Exits) reveal. Across the boundary, the probability that an observer links a given Exit to its originating deposit is bounded by a function of the anonymity set of the Pool over the interval between them, measured by its min-entropy H_∞:

$$\Pr[\mathrm{link}] \;\le\; 2^{-\mathsf{H}_\infty(D \mid \mathrm{view})} + \mathsf{negl}(\lambda), \tag{2.6}$$

where D is the deposit that funded the Exit and the view is everything the observer sees. *Rests on:* Property 2.2, and the Pool's activity over the interval. *Proved in:* Chapter 8, which defines the unlinkability game, computes H_∞ as a function of Pool activity, and shows how amount denominations, exit delays and Ring Exits raise it.

Property 2.4 (Liveness). If the Coprocessor submits a valid Commit at least once every W slots, a Crossing begins as soon as there is something to settle and Δ slots have passed since the last Commit, every Intent that is eligible at a Crossing's cutoff is processed in that Crossing, and every Exit request enters an exits root at the first Commit after its delay (Chapter 8). With nothing to settle, a live Coprocessor sends a heartbeat Commit 750 slots after the last (Chapter 10), so the gap between Commits stays far inside W whatever the trading activity. If no Commit lands within W slots, every owner of a nonzero leaf under the last committed root can obtain that balance at an address of their choice, provided t Council members respond. *Rests on:* Solana's liveness for the Commit and Escape transactions, and t honest and available Council members for the Escape. *Proved in:* Chapter 6 (the Escape safety theorem) and Chapter 10 (Commit submission and retries).

Property 2.5 (Price fairness). Every crossed fill on pair π in Crossing k executes at $p_\pi^{(k)} = m_\pi^{(k)}$, and every fill respects its Intent's limit:

$$s_i\big(\ell_i - \bar{p}_i\big) \;\geq\; 0 \qquad \text{for every Intent } i \text{ with } f_i > 0. \tag{2.7}$$

The routed portion executes at a price no worse than the bound the enclave signed, and the allocation of crossed and routed quantity among Intents follows a published rule whose fairness properties are proved. *Rests on:* the integrity of the enclave, the oracle within its confidence filter, and the program's check of each `min_out`. *Proved in:* Chapter 7, which also gives the expected-cost inequality that makes a crossed fill cheaper than the same trade routed, and the ordering analysis of a batched Residual against individual swaps.

Price fairness here has the same content as the best-execution obligation of a broker that internalizes: a client crossed internally receives the midpoint, and a client routed receives a market price within a bound, and neither can be filled outside their limit. The difference is that in Privium the rule is executed by attested code and its public consequences are checked by a program.

Property 2.6 (Auditability by the owner). A trader holding their viewing root vk_u can, using only public data, decrypt every one of their leaves, verify each leaf's inclusion under the committed root $\rho^{(k)}$ for every k, and reconstruct their balance history. The trader can delegate exactly this capability to a holder of a Viewing key, disclose a single entry, or issue a Receipt, an enclave-attested statement about their realized results that anyone can verify against the Registry without learning the trader's wallet. *Rests on:* the ciphertext diffs that every Commit publishes on Solana, and the security of the enclave's signing key. *Proved in:* Chapter 4 (inclusion), Chapter 11 (Viewing keys and Receipts).

Auditability belongs to the owner and to whomever the owner chooses. The design follows the disclosure model that Rings established on Solana, where an owner reads their own balance with a viewing key and can disclose a single transaction with a transaction viewing key [2], and extends it to trading results through the Receipt.

Summary

Property	Statement in one line	Rests on	Chapter
2.1 Custody	vaults equal the Ledger plus unreleased Exits, fees and uncredited deposits; tokens leave only by owner, bound or Escape	Solana; enclave arithmetic	4, 9
2.2 Confidentiality	Ledger and Intents indistinguishable given the public trace	enclave; IND-CCA2; Council threshold	3, 4
2.3 Unlinkability	linkage bounded by the anonymity set's min-entropy	2.2; Pool activity	8
2.4 Liveness	a Crossing once work waits and Δ has passed since the last Commit; a heartbeat Commit otherwise; Escape after W	Solana liveness; t honest Council members	6, 10
2.5 Price fairness	crosses at mid, limits respected, routed within signed bound	enclave; oracle filter; program check	7
2.6 Auditability by the owner	owner decrypts and verifies their history; delegates by Viewing key or Receipt	on-chain ciphertext diffs; sk_C	4, 11

Parameters introduced

The overview has introduced the protocol's parameters by symbol. Each is fixed, with a recommended value and its reason, in the chapter that owns its mechanism: the batch interval Δ, the oracle confidence filter and the Residual slippage bound in Chapter 7; the liveness window W and the Council size n and threshold t in Chapter 6; the heartbeat interval and the resting-order scan in Chapter 10; the key epoch length in Chapter 5; denominations, exit delays and the decoy floor in Chapter 8; and the fee rate ϕ, 0.10%, in Chapter 12. Two of them interact at the level of the whole system. The batch interval sets how much opposing flow each Crossing collects, and so how much of each order is crossed at the mid rather than routed; the liveness window must span many batch intervals, so that one delayed Commit never opens the Escape. Chapter 12 derives the crossing rate as a function of Δ and order flow, and Chapter 6 sets W more than two thousand batch intervals long.

References

1. N. Lokhande, R. Gupta, R. Chahar, A. Jangra, A. Prajapati, M. Kumari, "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
2. "Solana Privacy Rings", Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings

3. “Best Practices”, Pyth Network price feeds documentation, accessed 16 November 2026. https://docs.pyth.network/price-feeds/best-practices
4. “Swap API”, Jupiter Developer Docs, accessed 16 November 2026. https://developers.jup.ag/docs/swap-api

Chapter 3

Cryptographic foundations

Privium is built from a small set of standard primitives: a prime-order group, Pedersen commitments, twisted ElGamal encryption, Fiat-Shamir sigma protocols and Bulletproofs range proofs, AES-256-GCM, HPKE, binary Merkle trees, and two signature schemes. Each one is already deployed at scale, and most of them are the same objects Solana's ZK ElGamal proof stack verifies on chain today. This chapter fixes their exact form, states the security notions Privium uses, and proves the reductions on which the later chapters build: the confidential Ledger (Chapter 4), attestation (Chapter 5), threshold decryption and the Escape (Chapter 6), and Receipts (Chapter 11).

3.1 Conventions and hardness assumptions

Throughout, $\lambda = 128$ is the security parameter, an algorithm is efficient when it runs in probabilistic polynomial time (PPT) in λ, and a function f is negligible, $f = \mathsf{negl}(\lambda)$, when $f(\lambda) < \lambda^{-c}$ for every constant c and all large enough λ. We write $x \xleftarrow{R} S$ for uniform sampling and $\|$ for unambiguous concatenation of fixed-length or length-prefixed encodings. Every hash call carries a distinct ASCII domain tag of the form `privium/<purpose>/v1`, so that no two uses of H in this paper can be confused with one another.

Let $\mathbb{G}$ be a cyclic group of prime order $\ell_{\mathbb{G}}$ written additively, with scalars in $\mathbb{Z}_\ell$ where in this chapter $\ell = \ell_{\mathbb{G}}$. Privium relies on three standard problems in $\mathbb{G}$.

Definition 3.1 (DL, CDH, DDH). For a generator B of $\mathbb{G}$ and $\alpha, \beta, \gamma \xleftarrow{R} \mathbb{Z}_\ell$:

- the *discrete logarithm* problem is to compute α from $(B, \alpha B)$;
- the *computational Diffie-Hellman* problem is to compute $\alpha\beta B$ from $(B, \alpha B, \beta B)$;
- the *decisional Diffie-Hellman* problem is to distinguish $(B, \alpha B, \beta B, \alpha\beta B)$ from $(B, \alpha B, \beta B, \gamma B)$.

The advantage of an adversary $\mathcal{B}$ against DDH is $\mathsf{Adv}^{\mathrm{ddh}}_{\mathbb{G}}(\mathcal{B}) = |\Pr[\mathcal{B}(B, \alpha B, \beta B, \alpha\beta B) = 1] - \Pr[\mathcal{B}(B, \alpha B, \beta B, \gamma B) = 1]|$. Each assumption states that every efficient adversary has negligible advantage. DDH implies CDH implies DL. Because $\mathbb{G}$ has prime order, all three problems are random self-reducible: one instance can be turned into any polynomial number of independent, uniformly distributed instances at the cost of a few scalar multiplications, which we use in Chapter 4.

Where a proof uses a hash function as a random oracle, we say so. Elsewhere H is only assumed collision resistant: no efficient adversary outputs $x \neq x'$ with $\mathsf{H}(x) = \mathsf{H}(x')$ except with probability $\mathsf{negl}(\lambda)$.

3.2 The group: ristretto255

Why a prime-order group

Curve25519 in its twisted Edwards form (edwards25519) has a group of order $8 \cdot \ell$, where the cofactor 8 introduces small-order points. Protocols written for a prime-order group can fail on such a curve: two encodings can name the same logical point, a point can be multiplied by a torsion element without changing any check that compares only the prime-order part, and a zero-knowledge proof can verify for a statement that is false modulo the torsion. Ristretto255 removes this class of problems by construction. It defines $\mathbb{G}$ as a quotient of the edwards25519 group by its small torsion, together with a canonical 32-byte encoding that is injective on the quotient and a decoding that rejects every non-canonical string [3]. The resulting group has prime order

$$\ell_{\mathbb{G}} = 2^{252} + 27742317777372353535851937790883648493, \tag{3.1}$$

so every non-identity element generates $\mathbb{G}$ and every 32-byte string either decodes to exactly one element or is rejected [3]. Equality of encodings is equality of group elements, which is the property every check in Chapters 4 and 6 needs.

Generators

Privium uses the same two generators as Solana's ZK ElGamal SDK. G is the ristretto255 base point. H is obtained by hashing the compressed encoding of G with SHA3-512 and mapping the 64-byte digest into $\mathbb{G}$ with the ristretto255 one-way map [4]:

$$G = B_{\mathrm{ristretto255}}, \qquad H = \mathsf{MapToGroup}\big(\mathrm{SHA3\text{-}512}(\mathsf{enc}(G))\big). \tag{3.2}$$

Using Solana's (G, H) unchanged means Privium commitments and ciphertexts are byte-compatible with the ZK ElGamal proof program, so any of its verifiers can check statements about Ledger values (Section 3.5).

Lemma 3.1 (Independent generators). Model $\mathsf{MapToGroup} \circ \mathrm{SHA3\text{-}512}$ as a random oracle into $\mathbb{G}$. Then an efficient adversary that outputs η with $H = \eta G$ yields an efficient algorithm for DL in $\mathbb{G}$ with the same success probability, up to a polynomial loss in the number of oracle queries.

Proof. Given a DL instance (G, Y), program the oracle so that the query on $\mathsf{enc}(G)$ returns Y; every other query returns a fresh uniform element. The adversary's view is distributed exactly as in the real experiment, because Y is uniform in $\mathbb{G}$. If it outputs η with $H = \eta G$, then η is the discrete logarithm of Y. When the adversary may also query the oracle on other points that it later treats as generators, guess the relevant query index, which loses a factor equal to the number of queries. $\square$

No party, including Solana, the Privium developers or the Coprocessor, knows $\log_G H$. Every binding argument below rests on this lemma.

3.3 Security definitions

This section collects the notions referred to throughout the paper. The games are standard; we fix them so that the reductions in Chapters 4 to 6 have a single reference point.

Definition 3.2 (IND-CPA and IND-CCA2). Let $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme. In the IND-CPA game the challenger samples $(sk, pk) \leftarrow \mathsf{KeyGen}$ and gives pk to $\mathcal{A}$; $\mathcal{A}$ outputs equal-length messages m_0, m_1; the challenger samples $b \xleftarrow{R} \{0,1\}$ and returns $c^* = \mathsf{Enc}(pk, m_b)$; $\mathcal{A}$ outputs b'. The advantage is $\mathsf{Adv}^{\text{ind-cpa}}_{\Pi}(\mathcal{A}) = |2\Pr[b' = b] - 1|$. In the IND-CCA2 game, $\mathcal{A}$ additionally has a decryption oracle before and after the challenge, which it may query on any ciphertext other than c^*. Π is IND-CPA (respectively IND-CCA2) secure if every efficient $\mathcal{A}$ has negligible advantage.

Definition 3.3 (EUF-CMA and SUF-CMA). For a signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, the adversary receives pk and a signing oracle. It wins the EUF-CMA game if it outputs (m, σ) with $\mathsf{Verify}(pk, m, \sigma) = 1$ for a message m it never queried. It wins the SUF-CMA game if the pair (m, σ) was never returned by the oracle, so that even a fresh signature on a queried message counts as a forgery.

Definition 3.4 (Commitment hiding and binding). A commitment scheme $\mathsf{Com}(m; r)$ is *perfectly hiding* if for all m_0, m_1 the distributions of $\mathsf{Com}(m_0; r)$ and $\mathsf{Com}(m_1; r)$ over uniform r are identical. It is *computationally binding* if no efficient adversary outputs $(m, r) \neq (m', r')$ with $\mathsf{Com}(m; r) = \mathsf{Com}(m'; r')$ except with negligible probability.

Definition 3.5 (AEAD security). A nonce-based AEAD scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ with key k, nonce n, associated data ad and message m provides *IND-CPA* security against nonce-respecting adversaries (no nonce is used twice under one key) and *INT-CTXT*: no efficient adversary with an encryption oracle produces a fresh (n, ad, c) that decrypts to a value other than $\perp$. By the Bellare-Namprempre composition theorem, IND-CPA together with INT-CTXT implies IND-CCA2 for the symmetric setting [5].

Definition 3.6 (Sigma protocol). A three-move public-coin protocol (Y, c, z) for a relation $\mathcal{R}$ is a sigma protocol if it is complete; has *special soundness*, meaning that from two accepting transcripts (Y, c, z) and (Y, c', z') with $c \neq c'$ an efficient extractor computes a witness; and is *special honest-verifier zero knowledge* (SHVZK), meaning that for any given challenge c a simulator outputs transcripts distributed identically to honest ones.

3.4 Pedersen commitments

Definition 3.7 (Pedersen commitment). For a value $x \in \mathbb{Z}_\ell$ and opening $r \xleftarrow{R} \mathbb{Z}_\ell$,

$$\mathsf{Com}(x; r) = xG + rH. \tag{3.3}$$

This is the commitment of Pedersen [6] and the one Solana's SDK computes as a two-term multiscalar multiplication over (G, H) [4].

Theorem 3.1. Pedersen commitments are perfectly hiding, and computationally binding under the DL assumption in $\mathbb{G}$ and Lemma 3.1.

Proof. Hiding: write $H = \eta G$ with $\eta \neq 0$ (unknown to everyone, but well defined). For fixed x, the map $r \mapsto xG + rH = (x + \eta r)G$ is a bijection from $\mathbb{Z}_\ell$ to $\mathbb{G}$, so a uniform r yields a uniform commitment whatever x is. Binding: suppose an adversary outputs $(x, r) \neq (x', r')$ with $xG + rH = x'G + r'H$. Then $r \neq r'$ (otherwise $x = x'$), and $\eta = (x - x')(r' - r)^{-1}$ is the discrete logarithm of H to base G, contradicting Lemma 3.1. $\square$

Pedersen commitments are additively homomorphic: $\mathsf{Com}(x_1; r_1) + \mathsf{Com}(x_2; r_2) = \mathsf{Com}(x_1 + x_2; r_1 + r_2)$. Chapter 4 uses this to check asset conservation over the whole Ledger with a single group equation per asset. Two facts about integer values make that check meaningful. First, Ledger balances are integers in $[0, 2^{64})$, and any sum of at most 2^{32} of them is below $2^{96} < \ell$, so equality in $\mathbb{Z}_\ell$ implies equality over the integers. Second, a range proof (Section 3.6) is what ties a commitment to an integer in $[0, 2^{64})$ rather than to an arbitrary element of $\mathbb{Z}_\ell$.

3.5 ElGamal and twisted ElGamal

Exponential ElGamal

ElGamal encryption [7] with key pair (s, sB) encrypts a group element M as $(rB, M + r \cdot sB)$. Encrypting a scalar x "in the exponent" as $M = xG$ makes the scheme additively homomorphic, and decryption returns xG, from which x is recovered by a discrete logarithm search. That search is efficient only for small x, which is why every confidential balance system, Solana's included, bounds the plaintexts it must decrypt (below).

Twisted ElGamal

Twisted ElGamal, introduced for the PGC confidential payment system [8] and adopted by Solana's confidential transfer extension [9], rearranges the ciphertext so that one component is exactly a Pedersen commitment.

Definition 3.8 (Twisted ElGamal). A secret key is $s \xleftarrow{R} \mathbb{Z}_\ell^*$ and the public key is $P = s^{-1}H$. To encrypt x under P, sample $r \xleftarrow{R} \mathbb{Z}_\ell$ and output

$$\mathsf{ct} = (C, D) = (xG + rH,\ rP). \tag{3.4}$$

Decryption computes $C - sD = xG + rH - srs^{-1}H = xG$ and then x by bounded discrete logarithm. Solana's SDK implements exactly these formulas: the public key is the inverse of the secret times H, the ciphertext is a Pedersen commitment plus a *decryption handle* $D = rP$, and decryption subtracts $s \cdot D$ from the commitment [4]. The commitment C is independent of the key and the handle D is independent of the message [9], which is what allows proofs about Pedersen commitments to apply to ciphertexts directly.

Definition 3.9 (Grouped ciphertext). To encrypt one value to m keys $P_1, \dots, P_m$ with a single commitment, reuse r across handles: $(C, D_1, \dots, D_m) = (xG + rH, rP_1, \dots, rP_m)$. The holder of s_j decrypts with $C - s_j D_j$. Solana's confidential transfers encrypt each transfer amount this way to the source, the destination and the optional auditor, and its proof program verifies validity of grouped ciphertexts with two and three handles [10][11].

Theorem 3.2 (IND-CPA of grouped twisted ElGamal). For every adversary $\mathcal{A}$ against the IND-CPA security of grouped twisted ElGamal with m handles there is an adversary $\mathcal{B}$ against DDH in $\mathbb{G}$ with essentially the same running time such that

$$\mathsf{Adv}^{\text{ind-cpa}}(\mathcal{A}) \le 2 \cdot \mathsf{Adv}^{\text{ddh}}_{\mathbb{G}}(\mathcal{B}). \tag{3.5}$$

Proof. $\mathcal{B}$ receives (H, X, Y, Z) with $X = \alpha H$, $Y = \beta H$, and Z either $\alpha\beta H$ or uniform; DDH relative to the fixed generator H is equivalent to DDH relative to any generator, since $\mathbb{G}$ has prime order. $\mathcal{B}$ samples $\rho_1, \dots, \rho_m \xleftarrow{R} \mathbb{Z}_\ell^*$ and publishes the keys $P_j = \rho_j X$. These are distributed as honest keys: $P_j = (\rho_j \alpha)H$, so $s_j = (\rho_j \alpha)^{-1}$ is uniform in $\mathbb{Z}_\ell^*$ and independent across j. On challenge messages x_0, x_1, $\mathcal{B}$ picks $b \xleftarrow{R} \{0,1\}$ and returns

$$(C, D_1, \dots, D_m) = (x_b G + Y,\ \rho_1 Z, \dots, \rho_m Z),$$

then outputs 1 if $\mathcal{A}$ guesses b. If $Z = \alpha\beta H$, set $r = \beta$: then $C = x_b G + rH$ and $D_j = \rho_j \alpha\beta H = rP_j$, a correctly distributed ciphertext, so $\mathcal{B}$ outputs 1 with exactly $\mathcal{A}$'s success probability. If $Z = \gamma H$ for uniform γ, the handles depend only on γ, and $Y = \beta H$ is uniform and independent of them, so $C = x_b G + Y$ is uniform and independent of b, and $\mathcal{B}$ outputs 1 with probability exactly $1/2$. The difference of the two probabilities is $\frac{1}{2}\mathsf{Adv}^{\text{ind-cpa}}(\mathcal{A})$. $\square$

The proof covers randomness reuse across handles, the setting analysed in general by Bellare, Boldyreva and Staddon [12]. Theorem 3.2 is what Chapter 4 invokes for the Council handle carried by each Ledger leaf, and what Chapter 6 invokes for the Council's joint key.

How Solana deploys it

The confidential transfer extension of Token-2022 stores an account's available balance twice: as a twisted ElGamal ciphertext, used to generate proofs, and as an AES-GCM-SIV ciphertext (`decryptable_available_balance`) that the owner decrypts directly in routine use [9]. Incoming credits accumulate in a separate pending balance, split into a low and a high ciphertext, until the owner applies them. Transfer amounts are restricted to 48 bits, encrypted as a 16-bit low part and a 32-bit high part, because discrete logarithm decryption cost grows exponentially with the plaintext and already takes seconds for 32-bit values [9]. Privium takes the same two-layer pattern (a homomorphic layer that proofs and the Council act on, and a symmetric layer that the owner and the Coprocessor read) and drops the need for discrete logarithm decryption entirely: in the Ledger, the plaintext always travels alongside in an AEAD ciphertext, so the ElGamal layer only ever has to confirm a claimed value x by checking $C - sD = xG$, never to find it (Chapter 4, Chapter 6).

The ZK ElGamal proof program, which verifies these proofs on chain, was disabled at the start of mainnet epoch 805 on 19 June 2025 after a researcher reported a soundness flaw in its Fiat-Shamir transcript [13][14]. Confidential transfers were restored on mainnet in June 2026 after a competitive audit and a patched validator release [14], and the proof program is live alongside the BN254 and BLS12-381 syscalls delivered with Agave 4.0 [15].

3.6 Sigma protocols, Fiat-Shamir and range proofs

The template

Every proof in the confidential transfer stack apart from range proofs is a sigma protocol over $\mathbb{G}$ made non-interactive with Fiat-Shamir. Solana's documentation lists the ones its proof program verifies: public key validity, ciphertext validity for two and three handles, zero ciphertext, ciphertext-commitment equality, ciphertext-ciphertext equality and percentage-with-cap, plus batched range proofs over 64, 128 and 256 bits [10][11]. None of them needs a trusted setup [10]. The simplest is Schnorr's proof of knowledge of a discrete logarithm [16]; we give the ciphertext-commitment equality proof in full because the Ledger's batched validity proof follows its template (Chapter 4, Definition 4.4).

Protocol 3.1 (Ciphertext-commitment equality). Public input: a key P, a ciphertext (C_E, D) under P, and a Pedersen commitment C_P. Witness: (s, x, r') with $sP = H$, $C_E - sD = xG$ and $C_P = xG + r'H$.

1. The prover samples $y_s, y_x, y_r \xleftarrow{R} \mathbb{Z}_\ell$ and sends $Y_0 = y_s P$, $Y_1 = y_x G + y_s D$, $Y_2 = y_x G + y_r H$.
2. The verifier sends $c \xleftarrow{R} \mathbb{Z}_\ell$.
3. The prover sends $z_s = cs + y_s$, $z_x = cx + y_x$, $z_r = cr' + y_r$.

The verifier accepts if

$$z_s P = cH + Y_0, \qquad z_x G + z_s D = c\,C_E + Y_1, \qquad z_x G + z_r H = c\,C_P + Y_2. \tag{3.6}$$

Lemma 3.2. Protocol 3.1 is a sigma protocol for its relation.

Proof. Completeness: the first and third equations are immediate. For the second, $sD = srP = rH$, so $xG + sD = xG + rH = C_E$, and $z_x G + z_s D = c(xG + sD) + y_x G + y_s D = c\,C_E + Y_1$. Special soundness: from accepting transcripts with challenges $c \neq c'$ and the same (Y_0, Y_1, Y_2), subtracting the equations gives $(z_s - z_s')P = (c - c')H$, $(z_x - z_x')G + (z_s - z_s')D = (c - c')C_E$ and $(z_x - z_x')G + (z_r - z_r')H = (c - c')C_P$. Setting $\hat{s} = (z_s - z_s')/(c - c')$, $\hat{x} = (z_x - z_x')/(c - c')$ and $\hat{r} = (z_r - z_r')/(c - c')$ yields $\hat{s}P = H$, $C_E - \hat{s}D = \hat{x}G$ and $C_P = \hat{x}G + \hat{r}H$, a valid witness. SHVZK: given c, sample z_s, z_x, z_r uniformly and solve (3.6) for Y_0, Y_1, Y_2; in an honest run the responses are uniform and the commitments are determined by them, so the distributions are identical. □

The Chaum-Pedersen proof of discrete logarithm equality, which shows $\log_{B_1} Q_1 = \log_{B_2} Q_2$, is the one-equation special case of the same template and is what each Council member attaches to a partial decryption (Chapter 6).

Fiat-Shamir done strongly

The non-interactive form replaces c by $\mathsf{H}(\mathsf{tag} \,\|\, \mathrm{statement} \,\|\, Y_0 \,\|\, Y_1 \,\|\, Y_2)$. In the random oracle model, the forking lemma of Pointcheval and Stern turns any prover that convinces the verifier with non-negligible probability into two accepting transcripts with the same first message and distinct challenges, hence into a witness by special soundness [17]. That argument needs the hash input to contain the entire statement. When it omits part of the statement (the *weak* Fiat-Shamir transform), a prover can choose the unhashed part after seeing the challenge and prove false statements, a failure mode documented by Bernhard, Pereira and Warinschi [18] and the same class of defect that took Solana's proof program offline in 2025 [14]. Privium's rule is therefore mechanical: every transcript absorbs the domain tag, the program id, every public key, every ciphertext and commitment of the statement, and every first-round message, in a fixed order, before the challenge is squeezed.

Range proofs

A range proof shows that a commitment $V = vG + \gamma H$ opens to $v \in [0, 2^n)$. Solana's stack uses Bulletproofs [11], which prove the bit decomposition of v with an inner-product argument and need no trusted setup [19]. An aggregated proof for m values of n bits each consists of $2\lceil \log_2(nm) \rceil + 4$ group elements and 5 scalars [19]; with 32-byte encodings its size is

$$|\pi_{\mathrm{range}}(n, m)| = 32 \cdot \big(2\lceil \log_2(nm) \rceil + 9\big) \text{ bytes}. \tag{3.7}$$

For $n = 64$ this gives 672 bytes for one value and 1,312 bytes for 1,024 values aggregated. Logarithmic growth is what lets the Ledger publish a range proof for every balance it rewrites in a Crossing at a cost of about one kilobyte per asset (Chapter 4, Section 4.9).

3.7 Authenticated encryption and HPKE

AES-256-GCM

AE is AES-256-GCM as specified in NIST SP 800-38D: a 256-bit key, a 96-bit nonce and a 128-bit authentication tag [20]. GCM is IND-CPA and INT-CTXT secure, hence IND-CCA2 by Definition 3.5, for every adversary that never repeats a nonce under one key [20]. The Ledger therefore never draws nonces at random. It derives each nonce from the leaf position and the

leaf's monotone version counter (Chapter 4, Section 4.8), which makes nonce uniqueness a structural invariant rather than a probabilistic one. AES-256-GCM is also the symmetric layer the Encifher coprocessor reports at more than one million operations per second [2].

HPKE for Intents

Intents are encrypted to the Coprocessor with Hybrid Public Key Encryption, RFC 9180 [21], in base mode with the suite

$$\mathsf{KEM} = \mathrm{DHKEM}(\mathrm{X25519}, \mathrm{HKDF\text{-}SHA256}), \quad \mathsf{KDF} = \mathrm{HKDF\text{-}SHA256}, \quad \mathsf{AEAD} = \mathrm{AES\text{-}256\text{-}GCM},$$

whose RFC identifiers are `0x0020`, `0x0001` and `0x0002` [21]. The Coprocessor holds an X25519 key pair (dk_C, ek_C) generated inside the enclave for each key epoch and bound to its measurement through the attestation report (Chapter 5); ek_C is distinct from the Ed25519 signing key pk_C. An HPKE ciphertext is the 32-byte encapsulated key followed by the AEAD output, so it is 48 bytes longer than its plaintext [21].

Theorem 3.3 (Intent confidentiality). Under the Gap Diffie-Hellman assumption on X25519, with HKDF modeled as a random oracle and AES-256-GCM IND-CPA and INT-CTXT secure, HPKE base mode with the suite above is IND-CCA2 secure [22].

This is the result of Alwen et al.'s analysis of the HPKE standard [22]; we use it as stated. Two Privium-specific choices sit on top of it. The HPKE `info` string binds every ciphertext to the deployment and to the attested key epoch, so that an Intent sealed for one enclave key is rejected by any other:

$$\mathsf{info} = \texttt{privium/intent/v1} \,\|\, \mathsf{id}_{\mathrm{prog}} \,\|\, e \,\|\, \mathsf{H}(\mathsf{quote}_e), \tag{3.8}$$

where e is the key epoch and quote_e the attestation quote registered for it. And the plaintext has a fixed length: the Intent $i = (u, \pi, s, x, \ell, \tau, \nu)$ is serialized into fixed-width fields together with the trader's Ed25519 signature over those fields and zero padding to 128 bytes, so every Intent ciphertext is 176 bytes and its length says nothing about the order. The signature inside the ciphertext authenticates the trader to the enclave without appearing on chain.

```ts
import { CipherSuite, DhkemX25519HkdfSha256, HkdfSha256, Aes256Gcm } from "@hpke/core";
import { ed25519 } from "@noble/curves/ed25519";

const suite = new CipherSuite({
  kem: new DhkemX25519HkdfSha256(),
  kdf: new HkdfSha256(),
  aead: new Aes256Gcm(),
});

export const INTENT_PT_LEN = 128;

export async function sealIntent(
  intent: IntentFields,          // u, pair, side, size, limit, expiry, nonce
  authority: Uint8Array,         // trader's Ed25519 secret key
  enclave: AttestedKey,          // ek_C, epoch e, quote hash, from the Registry
): Promise<Uint8Array> {
  const body = encodeIntent(intent);                 // 45 bytes, fixed width
  const sig = ed25519.sign(body, authority);         // 64 bytes
  const pt = new Uint8Array(INTENT_PT_LEN);          // zero padded
  pt.set(body, 0);
  pt.set(sig, body.length);

  const sender = await suite.createSenderContext({
    recipientPublicKey: await suite.kem.importKey("raw", enclave.ekC),
    info: intentInfo(enclave.programId, enclave.epoch, enclave.quoteHash),
  });
  const ct = await sender.seal(pt);                  // 128 + 16 bytes
  return concat(new Uint8Array(sender.enc), new Uint8Array(ct)); // 176 bytes
}
```

The listing is the SDK's sealing routine. `encodeIntent` writes the trader's record position, the two asset indices of the pair, the side, the size, the limit price as a fixed-point integer, the expiry slot and the nonce in a fixed order; Chapter 10 shows where the SDK obtains the attested key and how `privium.trade(...)` calls this function.

3.8 Merkle trees and inclusion proofs

The Ledger is committed on chain as the root of a binary Merkle tree [23] of fixed depth d over ciphertext leaves (Chapter 4). Privium uses the domain separation of RFC 6962 [24], which prefixes leaf hashes and interior hashes with different bytes, and SHA-256 for H, which Solana programs compute through a runtime syscall [25].

Definition 3.10 (Ledger tree). For a leaf encoding L and children h_0, h_1,

$$\mathsf{H}_{\mathrm{leaf}}(L) = \mathsf{H}(\texttt{0x00} \,\|\, L), \qquad \mathsf{H}_{\mathrm{node}}(h_0, h_1) = \mathsf{H}(\texttt{0x01} \,\|\, h_0 \,\|\, h_1). \tag{3.9}$$

An empty position holds $z_0 = \mathsf{H}_{\mathrm{leaf}}(\varepsilon)$, and an empty subtree of height $j+1$ hashes to $z_{j+1} = \mathsf{H}_{\mathrm{node}}(z_j, z_j)$. An inclusion proof for position $\iota \in [0, 2^d)$ is the list of d sibling hashes $(\sigma_0, \ldots, \sigma_{d-1})$; the verifier recomputes $h_0 = \mathsf{H}_{\mathrm{leaf}}(L)$ and $h_{j+1} = \mathsf{H}_{\mathrm{node}}(h_j, \sigma_j)$ or $\mathsf{H}_{\mathrm{node}}(\sigma_j, h_j)$ according to bit j of ι, and accepts if $h_d = \rho$.

Theorem 3.4 (Position binding). If an efficient adversary outputs a root ρ, a position ι, and two accepting inclusion proofs for distinct leaf encodings $L \neq L'$ at ι, then a collision for H can be computed from its output in time $O(d)$.

Proof. Let $(h_j)_{j=0}^{d}$ and $(h'_j)_{j=0}^{d}$ be the two recomputed paths. Both follow the bits of the same ι, so at every level the recomputed value sits on the same side, and both end at $h_d = h'_d = \rho$. If $h_0 = h'_0$, the distinct strings $\texttt{0x00}\|L$ and $\texttt{0x00}\|L'$ collide. Otherwise let $j^* < d$ be the largest index with $h_{j^*} \neq h'_{j^*}$. Then $h_{j^*+1} = h'_{j^*+1}$, while the interior inputs $\texttt{0x01}\|h_{j^*}\|\sigma_{j^*}$ and $\texttt{0x01}\|h'_{j^*}\|\sigma'_{j^*}$ (in the order fixed by bit j^*) differ in the block holding h_{j^*}: a collision. Both cases take one pass over the path. The prefixes make every leaf input differ from every interior input as strings, so a leaf can never be passed off as an interior node or the reverse. □

With $d = 32$ (Chapter 4 sets it), a proof is $32 \times 32 = 1{,}024$ bytes and verification costs 33 hash evaluations.

```rust
pub const DEPTH: usize = 32;

pub fn leaf_hash(leaf: &[u8]) -> [u8; 32] {
    hashv(&[&[0x00], leaf]).to_bytes()
}

pub fn node_hash(l: &[u8; 32], r: &[u8; 32]) -> [u8; 32] {
    hashv(&[&[0x01], l, r]).to_bytes()
}

/// Recomputes the root for `leaf` at `index` from its sibling path.
pub fn root_from_path(leaf: &[u8], index: u32, path: &[[u8; 32]; DEPTH]) -> [u8; 32] {
    let mut h = leaf_hash(leaf);
    for (j, sib) in path.iter().enumerate() {
        h = if (index >> j) & 1 == 0 { node_hash(&h, sib) } else { node_hash(sib, &h) };
    }
    h
}
```

`hashv` is the Solana SHA-256 helper over a list of slices, which lowers to the `sol_sha256` syscall on chain [25]; the same function runs in the Coprocessor and in the SDK, so the root that the enclave signs, the root that a trader checks and the root that the Escape verifies are computed by one piece of code.

3.9 Signatures: Ed25519 and ECDSA-P256

Two signature schemes appear in Privium, each where its ecosystem already uses it.

Ed25519 (RFC 8032 [26]) signs every Commit with sk_C and every Intent with the trader's authority key. Brendel et al. prove that Ed25519 with the verification rules of RFC 8032 is SUF-CMA in the random oracle model under the DL assumption on edwards25519, provided the verifier rejects non-canonical encodings of the signature scalar, which is the setting that rules out signature malleability [27]. Privium needs strong unforgeability because the Commit signature is the only thing that authorizes a new Ledger root: a second valid signature on an old message must not be usable as a fresh authorization, and in Privium it cannot be, because the signed message includes the Crossing index (Chapter 4, Section 4.8). Solana verifies Ed25519 signatures through its native precompile; Chapter 5 gives the verification pattern and Chapter 9 the instruction layout.

ECDSA over P-256 (FIPS 186-5 [28]) signs the attestation material produced by trusted execution hardware. Privium does not choose it; it inherits it from the hardware vendor's chain of trust, and verifies it once per enclave key registration, not per Commit (Chapter 5). ECDSA is EUF-CMA in the generic group model, and it is malleable: (r, s) and $(r, \ell_{P256} - s)$ verify together. Registration is keyed on the signed content, never on the signature bytes, so malleability changes nothing.

Every signed message in the protocol starts with its own domain tag and the program id, so a signature produced for one purpose cannot be replayed for another, and a signature produced for one Privium deployment cannot be replayed on another. Formally, the tags make the message spaces of distinct purposes disjoint, so an adversary that wins one game against a tagged message family wins EUF-CMA (or SUF-CMA) against the underlying scheme with the same advantage.

3.10 What each guarantee rests on

The table states, for every cryptographic guarantee the later chapters use, the primitive that provides it and the assumption under which it holds. Trust in the enclave and in the Council is treated in Chapters 5 and 6; here we list only what the

mathematics guarantees.

Guarantee	Primitive	Rests on	Used in
Ledger amounts hidden from observers	twisted ElGamal, AES-256-GCM	DDH in ristretto255 (Thm 3.2); GCM IND-CPA with unique nonces	Ch. 4
Committed balances cannot be reopened to other values	Pedersen	DL in ristretto255 (Thm 3.1, Lemma 3.1)	Ch. 4, 6
Per-asset conservation check is sound	Pedersen homomorphism, range proofs	DL; Bulletproofs soundness	Ch. 4
Intents readable only by the attested enclave	HPKE base mode	Gap-DH on X25519, ROM (Thm 3.3)	Ch. 7, 10
A root names exactly one leaf per position	Merkle tree	collision resistance of SHA-256 (Thm 3.4)	Ch. 4, 6
Only the attested enclave advances the Ledger	Ed25519	DL on edwards25519, ROM, SUF-CMA [27]	Ch. 4, 5, 9
Enclave key bound to its measurement	ECDSA-P256	EUF-CMA of ECDSA; vendor root of trust	Ch. 5
Council decrypts only correct shares	Chaum-Pedersen	special soundness, ROM (Lemma 3.2)	Ch. 6
Rewritten leaves are well-formed ciphertexts of in-range balances	sigma protocols, range proofs	DL, ROM	Ch. 4, 6

Each row is a standard reduction; none relies on a new assumption. The composition that matters most, conservation of every asset at every committed root, is proved in Chapter 4 from Theorem 3.1, Theorem 3.4 and the range proofs of Section 3.6.

References

1. "Solana Privacy Rings", Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings
2. N. Lokhande, R. Gupta, R. Chahar, A. Jangra, A. Prajapati, M. Kumari, "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
3. H. de Valence, J. Grigg, M. Hamburg, I. Lovecruft, G. Tankersley, F. Valsorda, "The ristretto255 and decaf448 Groups", RFC 9496, IRTF CFRG, December 2023. https://www.rfc-editor.org/rfc/rfc9496
4. Solana Program, `zk-elgamal-proof` repository, `zk-sdk/src/encryption/pedersen.rs` and `elgamal.rs`, accessed 2026-11-16. https://github.com/solana-program/zk-elgamal-proof
5. M. Bellare, C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm", ASIACRYPT 2000. https://eprint.iacr.org/2000/025
6. T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing", CRYPTO 1991. https://doi.org/10.1007/3-540-46766-1_9
7. T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", IEEE Transactions on Information Theory, 1985. https://doi.org/10.1109/TIT.1985.1057074
8. Y. Chen, X. Ma, C. Tang, M. H. Au, "PGC: Decentralized Confidential Payment System with Auditability", IACR ePrint 2019/319, 2019. https://eprint.iacr.org/2019/319
9. Solana Program, "Confidential Balances: Encryption", accessed 2026-11-16. https://www.solana-program.com/docs/confidential-balances/encryption
10. Solana Program, "Confidential Balances: Zero-Knowledge Proofs", accessed 2026-11-16. https://www.solana-program.com/docs/confidential-balances/zkps
11. Anza, "ZK ElGamal Proof Program", Agave documentation, accessed 2026-11-16. https://docs.anza.xyz/runtime/zk-elgamal-proof
12. M. Bellare, A. Boldyreva, J. Staddon, "Randomness Re-use in Multi-recipient Encryption Schemes", PKC 2003. https://doi.org/10.1007/3-540-36288-6_7
13. Solana Foundation, "Post-mortem: June 25, 2025", June 2025. https://solana.com/news/post-mortem-june-25-2025

14. Solana Compass, “Confidential Transfers Return to Solana Mainnet After Year-Long Security Pause”, 18 June 2026. https://solanacompass.com/news/confidential-transfers-return-to-solana-mainnet-after-year-long-security-pause
15. Solana Compass, “Solana Activates BN254, BLS12-381 and ZK ElGamal: Three Cryptographic Primitives Now Live”, 7 August 2026. https://solanacompass.com/news/solana-activates-bn254-bls12-381-and-zk-elgamal-three-cryptographic-primitives-now-live
16. C. P. Schnorr, “Efficient Signature Generation by Smart Cards”, Journal of Cryptology, 1991. https://doi.org/10.1007/BF00196725
17. D. Pointcheval, J. Stern, “Security Arguments for Digital Signatures and Blind Signatures”, Journal of Cryptology, 2000. https://doi.org/10.1007/s001450010003
18. D. Bernhard, O. Pereira, B. Warinschi, “How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios”, ASIACRYPT 2012. https://eprint.iacr.org/2016/771
19. 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
20. M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC”, NIST SP 800-38D, November 2007. https://csrc.nist.gov/pubs/sp/800/38/d/final
21. R. Barnes, K. Bhargavan, B. Lipp, C. Wood, “Hybrid Public Key Encryption”, RFC 9180, February 2022. https://www.rfc-editor.org/rfc/rfc9180
22. J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, D. Riepel, “Analysing the HPKE Standard”, EUROCRYPT 2021. https://eprint.iacr.org/2020/1499
23. R. C. Merkle, “A Digital Signature Based on a Conventional Encryption Function”, CRYPTO 1987. https://doi.org/10.1007/3-540-48184-2_32
24. B. Laurie, A. Langley, E. Kasper, “Certificate Transparency”, RFC 6962, June 2013. https://www.rfc-editor.org/rfc/rfc6962
25. Anza, `solana-sha256-hasher` crate documentation, accessed 2026-11-16. https://docs.rs/solana-sha256-hasher
26. S. Josefsson, I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA)”, RFC 8032, January 2017. https://www.rfc-editor.org/rfc/rfc8032
27. J. Brendel, C. Cremers, D. Jackson, M. Zhao, “The Provable Security of Ed25519: Theory and Practice”, IEEE S&P 2021. https://eprint.iacr.org/2020/823
28. NIST, “Digital Signature Standard (DSS)”, FIPS 186-5, February 2023. https://csrc.nist.gov/pubs/fips/186-5/final

Chapter 4

The confidential Ledger

The Ledger is Privium's book of accounts: one balance per trader per asset, held as ciphertext, committed on chain as a single Merkle root, and advanced once per Crossing by a Commit. This chapter defines its leaves, its state transition function and its Commit, and proves the property everything else depends on: at every committed root, for every asset, the balances in the Ledger add up exactly to what the Pool's vaults hold for traders. The proof uses the Pedersen layer of each leaf and the integer bounds of Chapter 3, so conservation is checked on chain by the program, not taken on trust from the Coprocessor. The chapter closes with data availability, replay and nonce discipline, and the growth of the Ledger in bytes.

4.1 From ciphertext handles to a Ledger

Encifher introduced the model the Ledger follows [2]. On-chain programs hold only 128-bit ciphertext handles; ciphertexts live in a public data availability layer; an enclave fetches them, decrypts, computes on plaintext, re-encrypts and commits attested, Merkle-anchored results; and Solana's declared read and write sets serve as the dependency graph, so confidential computation is ordered correctly without a scheduler of its own [2]. In the Encifher implementation each fresh encrypted value receives a 16-byte handle from a cryptographically secure generator, result handles are derived deterministically from the operation and its inputs, and each batch's outputs are packaged into a Merkle tree whose root the enclave signs and submits on chain with a reference to the data availability layer [3].

Privium runs one program over one state shape, so it specializes the model rather than generalizing it.

	Encifher, general coprocessor [2][3]	**Privium Ledger**
Unit of state	any encrypted value, named by a handle	a balance leaf or a trader record at a fixed tree position
Handle	16 random bytes, or derived from an operation	$\chi_{\iota,v}$: 128 bits derived from position ι and version v
On-chain state	handles in each program's accounts	one root $\rho^{(k)}$, a Crossing index k, per-asset accumulators
Ciphertext location	external data availability layer	Solana transactions, hash-chained by the program (Section 4.7)
What the program checks	enclave signature over the batch root	enclave signature, root chaining, data availability digest, asset conservation

The handle of the leaf at position ι in version v is

$$\chi_{\iota,v} = \mathsf{H}\big(\texttt{privium/handle/v1} \,\|\, \mathsf{id}_{\mathrm{prog}} \,\|\, \iota \,\|\, v\big)\,[0..16], \tag{4.1}$$

which keeps Encifher's 128-bit width and its deterministic derivation for results, and makes every handle unique across the Ledger's history. Indexers and the Coprocessor's ciphertext store key their records by χ (Chapter 10).

4.2 Leaves

Each leaf is encrypted for three readers at once, each through the layer suited to it: the Coprocessor reads an AEAD ciphertext under the Ledger data key, the owner reads an AEAD ciphertext under a key only the owner and the enclave hold, and the Council decrypts through a twisted ElGamal handle under its joint key, which is the path the Escape takes (Chapter 6). A Pedersen commitment shared by all three is what the program and every watcher compute with.

The keys are: the Ledger data key k_L, the AES-256-GCM key that enclaves hold and that the Council releases to each admitted enclave (Chapter 5); the owner leaf key $k_{u,a} = \mathsf{F}(vk_{u,a}, \texttt{"balance"})$, derived from the trader's asset key $vk_{u,a}$ in the Viewing key hierarchy of Chapter 11 (equation 11.2), so that the trader and anyone they hand $vk_{u,a}$ or vk_u can read the leaf; and the Council's joint twisted ElGamal key $P_{\mathcal{C}} = s^{-1}H$, whose secret s exists only as shares (Chapter 6). Each trader also holds an authority key A_u (Ed25519), which signs Intents, an escape key $Y_u = y_u G$, which claims an Escape payout, and a 32-byte salt ς_u; the SDK derives all of them from one wallet signature (Chapter 10).

Definition 4.1 (Trader record). The record of trader u is the leaf

$$\mathsf{rec}_u = \big(\mathtt{kind} = 1,\ v,\ \mathsf{AE.Enc}(k_L, n_{\iota,v}, \omega_u \,\|\, A_u \,\|\, Y_u \,\|\, vk_u \,\|\, \varsigma_u, ad)\big), \tag{4.2}$$

where v is the leaf's version, ω_u is the trader's nonce watermark (Section 4.8), vk_u is the trader's viewing root (Chapter 11), $n_{\iota,v}$ is the AEAD nonce of Section 4.8 and ad is the leaf's public part together with $\mathsf{id}_{\mathrm{prog}}$ and ι. Institutional accounts add a policy record of kind 3 in the same form, carrying the policy of Chapter 11.

Definition 4.2 (Ledger leaf). The balance of trader u in asset a is the leaf

$$L_{u,a} = \big(\mathtt{kind} = 2,\ \alpha(a),\ v,\ \psi_{u,a},\ C,\ D,\ \mathsf{ct}_L,\ \mathsf{ct}_u\big) \tag{4.3}$$

with

$$\begin{aligned}
\psi_{u,a} &= \mathsf{H}(\mathtt{privium/owner/v1} \,\|\, Y_u \,\|\, \varsigma_u \,\|\, a \,\|\, v),\\
C &= b_{u,a}G + rH, \qquad D = rP_{\mathcal{C}},\\
\mathsf{ct}_L &= \mathsf{AE.Enc}\big(k_L, n_{\iota,v}, \iota_u \,\|\, b_{u,a} \,\|\, r, ad\big),\\
\mathsf{ct}_u &= \mathsf{AE.Enc}\big(k_{u,a}, n_{\iota,v}, b_{u,a} \,\|\, r, ad\big),
\end{aligned}$$

where $\alpha(a)$ is the asset's 4-byte index in the Registry, ι_u is the position of u's record, $r \xleftarrow{R} \mathbb{Z}_\ell$ is fresh at every write, and $ad = (\mathsf{id}_{\mathrm{prog}}, \iota, \mathtt{kind}, \alpha(a), v, \psi_{u,a}, C, D)$.

(C, D) is a twisted ElGamal ciphertext of $b_{u,a}$ under $P_{\mathcal{C}}$ (Definition 3.8), and C alone is a Pedersen commitment to it. The Council decrypts it with $C - sD = b_{u,a}G$, which is the Escape's only use of the Council (Chapter 6); there is no separate escape ciphertext. Both AEAD layers carry the opening r along with the balance, which gives the owner something the Token-2022 decryptable balance does not need to give: the owner can check $C = bG + rH$ locally and so verify that the public commitment matches the balance they read. The owner tag $\psi_{u,a}$ is a hiding commitment to the owner's escape key; because it includes the version, it changes at every write, so two versions of one leaf, and the leaves of one trader in different assets, carry unrelated tags. It is opened only in an Escape request, where the claimant reveals (Y_u, ς_u) and later proves knowledge of y_u (Chapter 6). The asset index is public, since each asset's leaves are summed against that asset's vault; owner, amount and history are not. Chapter 8 measures what an observer learns from the remaining public fields.

```rust
pub const LEAF_LEN: usize = 225;
pub const RECORD_LEN: usize = 161;

#[repr(C)]
pub struct LedgerLeaf {
    pub kind: u8,              // 2
    pub asset: u32,            // Registry index alpha(a)
    pub version: u64,          // index k of the Crossing that wrote it
    pub owner_tag: [u8; 32],   // psi_{u,a}, fresh at every write
    pub commitment: [u8; 32],  // C = b G + r H   (ristretto255)
    pub council_handle: [u8; 32], // D = r P_C
    pub ct_ledger: [u8; 60],   // AES-256-GCM(k_L): record pos 4 | b 8 | r 32 | tag 16
    pub ct_owner: [u8; 56],    // AES-256-GCM(k_{u,a}): b 8 | r 32 | tag 16
}

impl LedgerLeaf {
    /// Public part bound into both AEAD ciphertexts as associated data.
    pub fn ad(&self, program: &Pubkey, index: u32) -> [u8; 145] {
        let mut ad = [0u8; 145];
        ad[0..32].copy_from_slice(program.as_ref());
        ad[32..36].copy_from_slice(&index.to_le_bytes());
        ad[36] = self.kind;
        ad[37..41].copy_from_slice(&self.asset.to_le_bytes());
        ad[41..49].copy_from_slice(&self.version.to_le_bytes());
        ad[49..81].copy_from_slice(&self.owner_tag);
        ad[81..113].copy_from_slice(&self.commitment);
        ad[113..145].copy_from_slice(&self.council_handle);
        ad
    }

    /// AEAD nonce: position and version, 96 bits, never repeated under one key.
    pub fn nonce(index: u32, version: u64) -> [u8; 12] {
        let mut n = [0u8; 12];
        n[0..4].copy_from_slice(&index.to_be_bytes());
        n[4..12].copy_from_slice(&version.to_be_bytes());
        n
    }
}
```

The listing is the leaf as the Coprocessor, the SDK and the indexer share it. Field widths add up to 225 bytes for a balance leaf; a trader record is 1 + 8 + 152 = 161 bytes (a 136-byte plaintext plus the GCM tag). Because C and D sit in the associated data, neither AEAD ciphertext can be moved to another leaf, another position, another version or another deployment without failing authentication.

4.3 The tree and its root

The Ledger is a sparse binary Merkle tree of depth d in the form of Definition 3.10, with

$$d = 32 \ (\text{parameter}),$$

because $2^{32} = 4{,}294{,}967{,}296$ positions hold every record and balance leaf the product can reach, and a 1,024-byte inclusion proof fits in one v1 Solana transaction, whose limit is 4,096 bytes [4], which is the size the Escape claim needs (Chapter 6). Every write places the leaf at a position drawn by the Coprocessor uniformly at random among free positions and empties the position it held before, so a position carries no information about arrival order or about the leaf's earlier versions (Chapter 8 builds on this relocation). The Ledger state after Crossing k is the map $S^{(k)} : [0, 2^d) \to \{\text{leaves}\} \cup \{\varepsilon\}$, and its root $\rho^{(k)}$ is the tree hash of $S^{(k)}$. The plaintext view of $S^{(k)}$ is the balance vector $\big(b_{u,a}^{(k)}\big)_{u \in \mathcal{U}, a \in \mathcal{A}}$.

4.4 The state transition function

A Crossing consumes four kinds of input. Deposits are public Pool instructions that move tokens into a vault and append an entry to one of the asset's deposit queue shards (Chapter 9), carrying a note sealed to the enclave's HPKE key ekC that names the trader's record; the Commit states the vector σ_k of per-shard cuts it absorbs. Intents are HPKE ciphertexts (Section 3.7) collected by the Coprocessor. Exit requests are Intents of a dedicated kind that name an asset, an amount and a destination address. Route fills are the vault flows of the Crossing's Residual swaps, which execute from the Pool in `route_residual` transactions after `open_crossing` and before the Commit, and which the program records in the Crossing's state account (Chapters 7 and 9). Only one Crossing is open at a time, since `open_crossing` for k requires the stored index $k - 1$, so every recorded fill belongs to the Crossing that the next Commit closes.

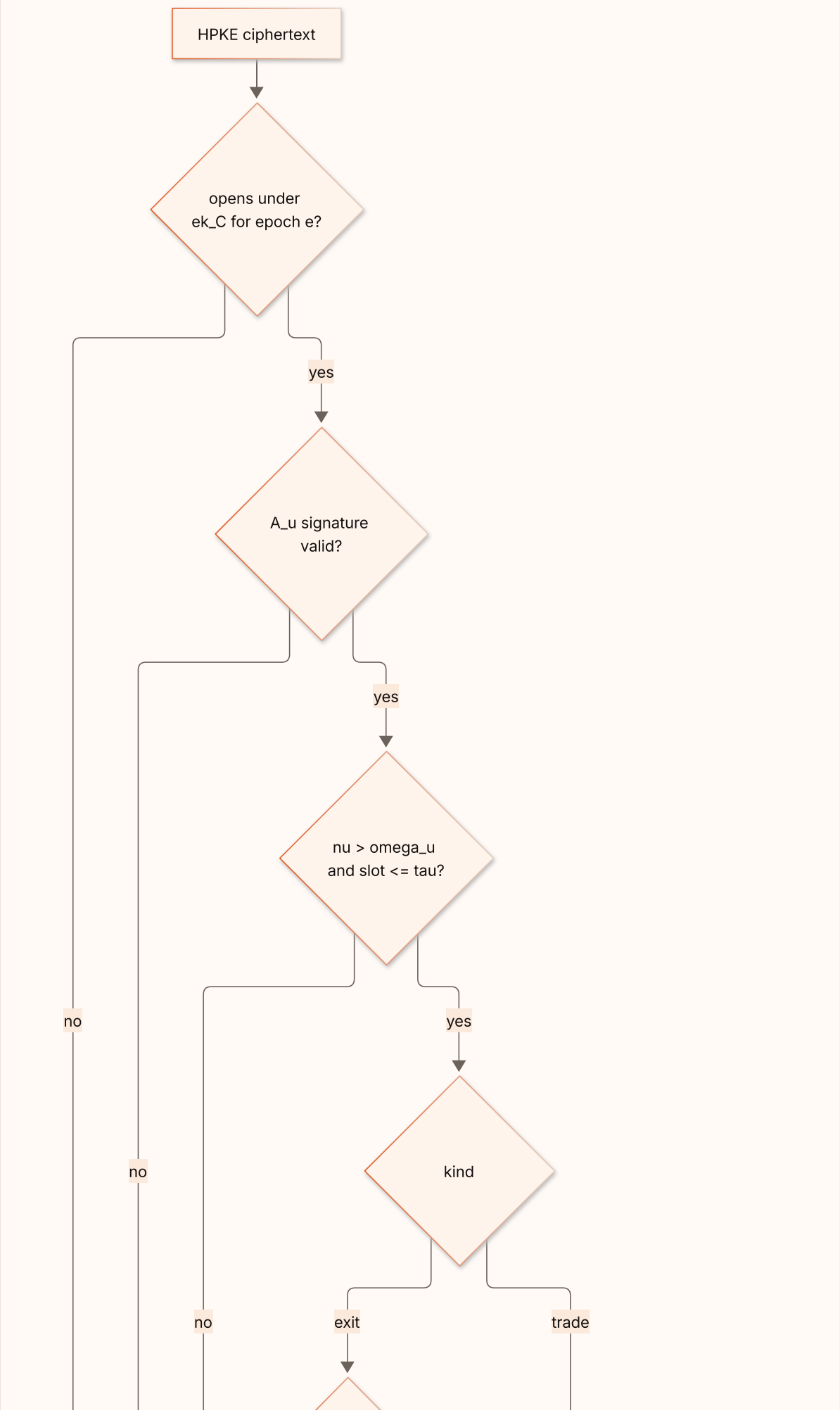
For asset a and Crossing k, let $D_a^{(k)}$ be the deposits absorbed (queue entries between the cuts σ_{k-1} and σ_k), $\Phi_a^{(k)}$ the signed net flow of asset a into the vaults from the Crossing's recorded fills, $X_a^{(k)}$ the total of the Exits the Commit lists, and $F_a^{(k)}$ the protocol fees the Crossing charges in a. All four are public on chain and recorded by the program.

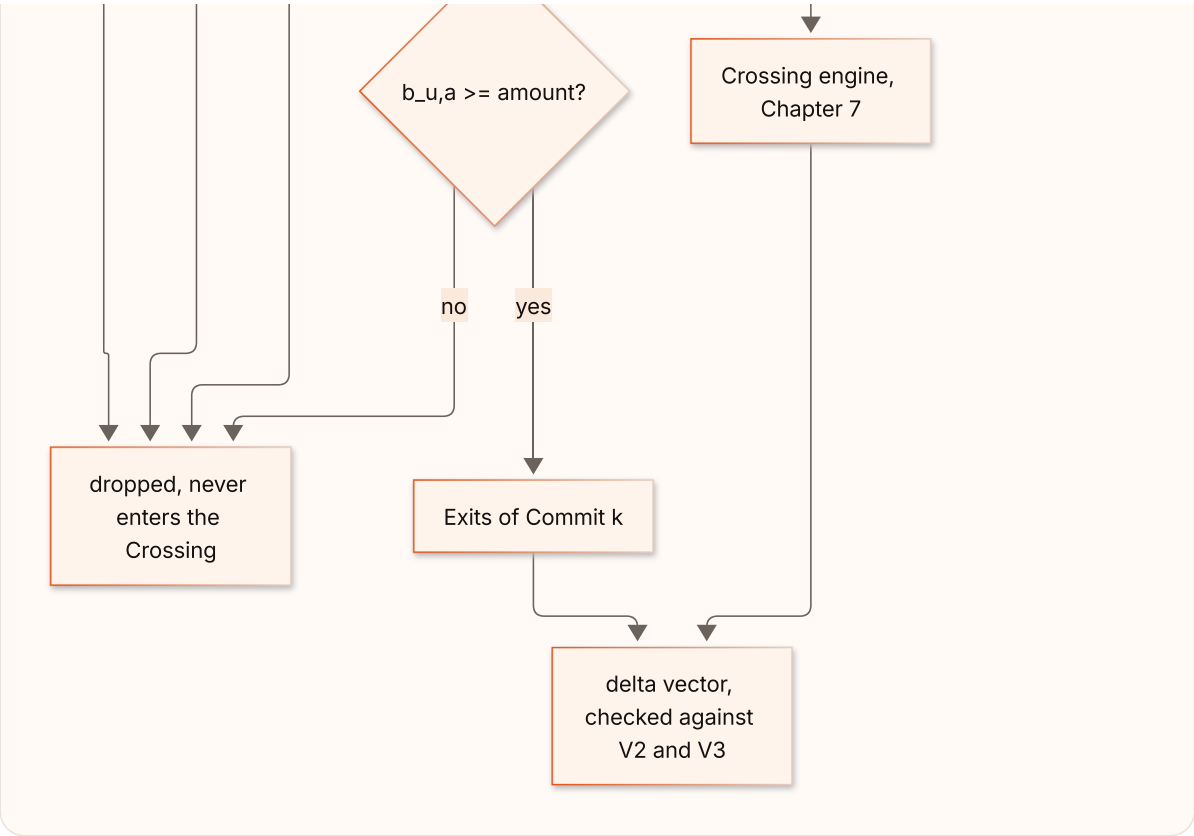
Definition 4.3 (Ledger transition). Given $S^{(k-1)}$ and the inputs of Crossing k, the Coprocessor decrypts every Intent, runs the Crossing engine of Chapter 7 on the valid ones, and obtains a delta vector $\big(\Delta_{u,a}^{(k)}\big)$. The transition is *valid* when:

- **(V1) Authorization.** Every negative delta of u is covered by an Intent or exit request of u that carries a valid A_u signature, an unexpired τ and a fresh nonce $\nu > \omega_u$.
- **(V2) Solvency.** $b_{u,a}^{(k)} = b_{u,a}^{(k-1)} + \Delta_{u,a}^{(k)} \in [0, 2^{64})$ for every u, a.
- **(V3) Balance.** For every asset a,

$$\sum_{u \in \mathcal{U}} \Delta_{u,a}^{(k)} = D_a^{(k)} + \Phi_a^{(k)} - X_a^{(k)} - F_a^{(k)}. \tag{4.4}$$

Fees leave the Ledger and the Pool together: the Commit attests $F_a^{(k)}$ and pays it in the same instruction from vault shard 0 to the fee recipient named in the Pool configuration, the token's launch wallet (Chapters 9 and 12). Each trader's fee is rounded down to whole base units and $F_a^{(k)}$ is the exact sum of those fees, and every proportional split of the Crossing uses the largest-remainder rule, so (4.4) holds exactly in integers with no remainder left over. The new state $S^{(k)}$ rewrites the leaf of every (u, a) with a nonzero delta, the record of every trader whose watermark moved, and a decoy set of further leaves that are re-randomized without a value change; Chapter 8 sets the decoy count. Re-randomizing a leaf replaces (C, D) by $(C + \zeta H, D + \zeta P_{\mathcal{C}})$ for fresh ζ, recomputes the owner tag for the new version, re-encrypts both AEAD layers, and moves the leaf to a fresh position (Section 4.3), so a rewritten leaf reveals nothing about whether its value changed (Theorem 4.3) or where it was before.


HPKE ciphertext
opens under ek_C for epoch e?
yes
A_u signature valid?
yes
nu > omega_u and slot <= tau?
yes
kind
exit
trade
no
no
no



An Intent that fails any check is dropped inside the enclave; nothing about it reaches the chain.

4.5 The Commit

The Commit for Crossing k carries the new root, the per-shard deposit cuts σ_k, the root $E^{(k)}$ of the Exits it lists, the data availability digest δ_k (Section 4.7), the hashes of the mids and fills recorded on chain for Crossing k, and for each asset a in the set J_k of assets it carries, the attested total $T_a^{(k)} = \sum_u b_{u,a}^{(k)}$, the Crossing's Exits $X_a^{(k)}$ and fees $F_a^{(k)}$, the commitment delta and the aggregate opening

$$\Delta\Gamma_a^{(k)} = \sum_{\iota \in \mathrm{rw}_k(a)} (C_\iota^{(k)} - C_\iota^{(k-1)}), \qquad R_a^{(k)} = \sum_{\iota\,:\,\alpha(\iota)=a} r_\iota^{(k)} \bmod \ell, \tag{4.5}$$

where $\mathrm{rw}_k(a)$ is the set of asset-a leaves rewritten in Crossing k (a new leaf counts with $C_\iota^{(k-1)} = O$, a vacated position contributes $-C_\iota^{(k-1)}$). The Coprocessor signs with sk_C the Commit message

$$M_k = \texttt{privium/commit/v1} \,\|\, \mathrm{id}_{\mathrm{prog}} \,\|\, e \,\|\, k \,\|\, \rho^{(k-1)} \,\|\, \rho^{(k)} \,\|\, \mathrm{body}_k, \tag{4.6}$$

where e is the signing key's epoch (Chapter 5) and body_k holds every other field above, in the byte layout of Equation 9.1. The program accepts the Commit only if all of the following hold.

- **(C1) Order.** No Escape is active, k equals the stored index plus one, and $\rho^{(k-1)}$ equals the stored root.
- **(C2) Authority.** An Ed25519 signature on M_k is present in the transaction, under a pk_C that the Registry admits for epoch e with a validity interval containing the current slot, and that opened Crossing k (Chapter 5 for the key, Chapter 9 for the precompile pattern).
- **(C3) Availability and binding.** δ_k equals the data availability accumulator in the Crossing's state account, which is then sealed; the mids and fills hashes equal what `open_crossing` and `route_residual` recorded; and the deposit digest matches the queue entries at the cuts.
- **(C4) Conservation.** For every $a \in J_k$, the attested total satisfies the vault equation (9.2) of Chapter 9, which states that, once the Commit has paid $F_a^{(k)}$ out, the vault holds exactly $T_a^{(k)}$, the committed Exits not yet released and the deposits not yet credited. With stored accumulators $\Gamma_a^{(k-1)}$, the weights $z_a = \mathsf{H}(\texttt{privium/conserve/v1} \,\|\, \mathsf{H}(M_k) \,\|\, \alpha(a) \,\|\, T_a^{(k)})$ satisfy

$$\sum_{a \in J_k} z_a (\Gamma_a^{(k-1)} + \Delta\Gamma_a^{(k)}) - \Big(\sum_{a \in J_k} z_a T_a^{(k)}\Big) G - \Big(\sum_{a \in J_k} z_a R_a^{(k)}\Big) H = O. \tag{4.7}$$

- **(C5) Exits.** The program adds each $X_a^{(k)}$ to the asset's committed, unreleased Exits and stores $E^{(k)}$; each Exit is then paid by a permissionless `release_exit` that proves its inclusion under $E^{(k)}$ (Chapter 9).

On success the program stores k, $\rho^{(k)}$, $\Gamma_a^{(k)} = \Gamma_a^{(k-1)} + \Delta\Gamma_a^{(k)}$, $T_a^{(k)}$ and the slot of the Commit, from which the Escape measures the liveness window W (Chapter 6). Check (4.7) is a single multiscalar multiplication of $|J_k| + 2$ ristretto255 points, which programs compute with Solana's curve25519 syscalls [5]. The listing is the per-asset body of check (C4) that the Commit handler of Chapter 9 calls for each carried asset.

```
/// Check C4 for one asset: pay the fee, Equation 9.2 in base units, then the terms of (4.7).
pub fn conserve(
    a: &AssetEntry, group: &[AccountInfo], pool: &PoolConfig, authority: &AccountInfo,
    chain: &mut Vec<[u8; 32]>, acc: &mut ConserveAcc,
) → Result<()> {
    let mut st = load_asset_state(&group[0], &a.mint)?;
    let (mint, vaults, queues) = (&group[1], &group[2..4], &group[4..6]);
    let (recipient, token_program) = (&group[6], &group[7]);
    require_keys_eq!(*recipient.key, fee_recipient_ata(pool, &st), PriviumError::Accounts);
    if a.fees > 0 {                                        // the Crossing's fees leave shard 0
        pay_from_vault(token_program, &vaults[0], mint, recipient, authority, a.fees, &st)?;
    }
    let (mut held, mut uncredited) = (0u128, 0u128);
    for j in 0..SHARDS {
        let v = vault_account(&vaults[j], &a.mint, &pool.authority)?;
        let q = queue_account(&queues[j], &a.mint, j)?;
        require!(a.cut[j] ≥ st.consumed[j] && a.cut[j] ⩽ q.head, PriviumError::Cut);
        let last = q.entry(a.cut[j] - 1)?;                 // last credited entry
        chain.push(last.chain_hash);
        held += v.amount as u128;
        uncredited += (q.cum_amount - last.cum_amount) as u128;
        st.consumed[j] = a.cut[j];
    }
    st.pending_exits = st.pending_exits.checked_add(a.exits).ok_or(PriviumError::Math)?;
    let owed = a.total as u128 + st.pending_exits as u128;
    require!(owed + uncredited == held, PriviumError::Conservation); // Equation 9.2

    let gamma = st.gamma.add(&a.delta_gamma)?;             // canonical ristretto decode
    acc.push(&a.mint, a.total, &gamma, &a.r_agg);          // z_a and the terms of (4.7)
    st.ledger_total = a.total;
    st.gamma = gamma;
    Ok(())
}
```

`ConserveAcc` derives each weight z_a from $\mathsf{H}(M_k)$ and evaluates (4.7) as one multiscalar multiplication after the last asset. The state writes take effect only if every check of the transaction passes, since any failure aborts it. Chapter 9 gives the account layouts, the deposit queue shards, the fill register in the Crossing's state account and the compute budget.

4.6 Conservation

The program measures each vault $V_a^{(k)}$ by its own accounting: every token movement into or out of a vault passes through a Pool instruction that records it. The pending amount $\Pi_a^{(k)}$ is the part of that balance the Ledger does not owe traders at Commit k: committed Exits not yet released and deposits queued after the cuts σ_k. The Crossing's fees are not part of it: the Commit pays them out of the vault before the check. No route flow is pending at a committed root, because routes run only for the open Crossing and its Commit absorbs every recorded fill. Check (C4) through Equation 9.2 gives $T_a^{(k)} = V_a^{(k)} - \Pi_a^{(k)}$ at every accepted Commit.

Theorem 4.1 (Conservation of the Ledger). If every transition is valid in the sense of Definition 4.3, then after every Commit k and for every asset a,

$$\sum_{u \in \mathcal{U}} b_{u,a}^{(k)} = V_a^{(k)} - \Pi_a^{(k)}. \tag{4.8}$$

Proof. By induction on k. At genesis all balances, vaults and pending amounts are zero. Assume (4.8) for $k-1$, so $\sum_u b_{u,a}^{(k-1)} = T_a^{(k-1)}$. By (V2) and (V3),

$$\sum_u b_{u,a}^{(k)} = \sum_u b_{u,a}^{(k-1)} + \sum_u \Delta_{u,a}^{(k)} = T_a^{(k-1)} + D_a^{(k)} + \Phi_a^{(k)} - X_a^{(k)} - F_a^{(k)} = T_a^{(k)},$$

and $T_a^{(k)} = V_a^{(k)} - \Pi_a^{(k)}$ by check (C4) and Equation 9.2, since the vault gains $D_a^{(k)} + \Phi_a^{(k)}$ and the pending amount gains $X_a^{(k)} + F_a^{(k)}$ at Commit k. □

Theorem 4.1 is a statement about the plaintext balances and holds whenever the enclave computes the transition it is attested to compute. The stronger statement is that the program's check (C4), together with data any watcher holds, enforces the same

equality on the committed values, so that what an Escape can pay out is bounded by the vault whatever the plaintext layers say.

Lemma 4.1 (Batched check). Model H as a random oracle and let $E_a = \Gamma_a^{(k-1)} + \Delta\Gamma_a^{(k)} - T_a^{(k)}G - R_a^{(k)}H$. If $E_{a^*} \neq O$ for some $a^* \in J_k$, an adversary making q oracle queries gets a Commit accepted with probability at most q/ℓ.

Proof. Every E_a is fixed by M_k and the program's state before the weights are computed, and each z_a is an oracle output on an input containing $\mathsf{H}(M_k)$, $\alpha(a)$ and $T_a^{(k)}$. Fix any candidate M_k and the weights for $a \neq a^*$. Because $\mathbb{G}$ has prime order and $E_{a^*} \neq O$, the equation $z_{a^*}E_{a^*} = -\sum_{a \neq a^*} z_a E_a$ has at most one solution $z_{a^*} \in \mathbb{Z}_\ell$, which a fresh oracle output hits with probability $1/\ell$. A union bound over the q queries the adversary can try gives q/ℓ. □

With $q \le 2^{80}$ and $\ell > 2^{252}$ the bound is below 2^{-172}.

Theorem 4.2 (Committed conservation). Suppose (i) every rewritten leaf carries a valid range proof for $[0, 2^{64})$ and a valid batched ciphertext validity proof in the diff of its Crossing (Section 4.7), and (ii) each published $\Delta\Gamma_a^{(k)}$ equals the sum (4.5) over the leaves in the diff, a check any watcher runs from data the program certified as published. Then, under the DL assumption in ristretto255 and the soundness of Bulletproofs, for every accepted Commit k and asset a, every opening (b_ι, r_ι) of the asset-a commitments in $S^{(k)}$ with $b_\iota \in [0, 2^{64})$ satisfies

$$\sum_{\iota\,:\,\alpha(\iota)=a} b_\iota = T_a^{(k)}, \tag{4.9}$$

except with negligible probability. Moreover, the Council decryption of every such leaf, the integer $\beta_\iota \in \mathbb{Z}_\ell$ with $\beta_\iota G = C_\iota - sD_\iota$, equals b_ι.

Proof. By (ii) and induction from $\Gamma_a^{(0)} = O$, the stored accumulator is the sum of the current commitments: $\Gamma_a^{(k)} = \sum_{\alpha(\iota)=a} C_\iota^{(k)}$. By Lemma 4.1, acceptance implies $E_a = O$, so $\Gamma_a^{(k)} = T_a^{(k)}G + R_a^{(k)}H$. By (i) and the knowledge soundness of Bulletproofs in the random oracle model [6], each C_ι has an extractable opening (b_ι, r_ι) with $b_\iota \in [0, 2^{64})$. Then $\big(\sum b_\iota\big)G + \big(\sum r_\iota\big)H = T_a^{(k)}G + R_a^{(k)}H$ gives two openings of one point. By Theorem 3.1 they agree, so $\sum b_\iota \equiv T_a^{(k)} \pmod{\ell}$, or else the adversary yields $\log_G H$. Both sides are non-negative integers below ℓ: the left is a sum of at most 2^{32} terms below 2^{64}, hence below 2^{96}, and $T_a^{(k)} < 2^{64}$. Congruence modulo $\ell > 2^{252}$ is therefore equality. Any other in-range opening of the same commitments that an Escape claimant presents equals the extracted one by binding, so (4.9) holds for every opening, not only the extracted one. For the decryptions, the validity proof of Definition 4.4 is a sigma proof of knowledge of (b^*, r^*) with $C^* = b^*G + r^*H$ and $D^* = r^*P_\mathcal{C}$ for $(C^*, D^*) = \sum_\iota \varrho^\iota (C_\iota, D_\iota)$, where ϱ is a random-oracle output on the whole diff. Binding gives $b^* = \sum \varrho^\iota b_\iota$ and $r^* = \sum \varrho^\iota r_\iota$, so $\sum_\iota \varrho^\iota(\beta_\iota - b_\iota)G = C^* - sD^* - b^*G = O$. If some $\beta_\iota \neq b_\iota$, the polynomial $\sum_\iota (\beta_\iota - b_\iota) z^\iota$ is nonzero of degree below 2^{32} and vanishes at ϱ with probability at most $2^{32}/\ell$ per oracle query. □

Corollary 4.1 (Escape solvency). Under the hypotheses of Theorem 4.2, the Council decryptions of the asset-a leaves under a committed root $\rho^{(k)}$ sum to exactly $T_a^{(k)} = V_a^{(k)} - \Pi_a^{(k)}$. An Escape pays each leaf its Council decryption (Chapter 6), so the claims it can establish against $\rho^{(k)}$ total $T_a^{(k)}$ and no more, whatever the AEAD layers say.

The on-chain check publishes $R_a^{(k)}$, the sum of all openings of an asset. That sum reveals the asset's total, which the vault already shows, and nothing more.

Theorem 4.3 (Aggregate openings reveal only totals). Let $\mathcal{A}$ choose two balance vectors $(b_1, \dots, b_N)$ and $(b'_1, \dots, b'_N)$ for the leaves of one asset with $\sum b_i = \sum b'_i$, and receive the public parts $(C_i, D_i)_{i \le N}$ of the leaves for one of them, chosen at random, together with $R = \sum r_i$. Then $\mathcal{A}$'s advantage in telling which vector was used is at most $2\,\mathsf{Adv}^{\mathrm{ddh}}_{\mathbb{G}}(\mathcal{B})$ for an adversary $\mathcal{B}$ of comparable running time.

Proof. The reduction samples $r_1, \dots, r_{N-1}$ uniformly and sets $r_N = R - \sum_{i<N} r_i$ for a uniform R; this is the honest joint distribution of $(r_1, \dots, r_N, R)$. From one DDH instance (H, X, Y, Z), random self-reducibility gives $N-1$ instances (H, X, Y_i, Z_i) with independent Y_i, all real or all random together. As in Theorem 3.2, set $P_\mathcal{C} = X$ and, for $i < N$, $D_i = Z_i$ and $C_i = b_i G + Y_i$, which is the honest form with $r_i = \log_H Y_i$ when the instances are real. Set $D_N = R\,X - \sum_{i<N} D_i$ and $C_N = \big(\sum_j b_j\big)G + RH - \sum_{i<N} C_i$, which are the honest values of the last leaf in the real case, since then $\sum_{i \le N} C_i = \big(\sum b_j\big)G + RH$ and $\sum_{i \le N} D_i = RP_\mathcal{C}$. In the random case each C_i for $i < N$ is uniform and independent of b_i, and (C_N, D_N) is a function of $\sum_j b_j$, R and the first $N-1$ leaves. The view then depends on the balances only through their sum, which is equal for both vectors. The real and random views differ by at most $\mathsf{Adv}^{\mathrm{ddh}}$, which with the factor of Definition 3.2 gives the bound. The AEAD layers are replaced by encryptions of zeros beforehand, at the cost of the IND-CPA advantage of AES-256-GCM under unique nonces. □

4.7 Data availability of ciphertexts

A root is useful only if the leaves under it can be retrieved: owners need their leaf to read their balance, watchers need every rewritten leaf to run check (ii), and an Escape needs the leaf and its inclusion proof. Encifher publishes each batch tree to an

external data availability service and references the submission on chain [3]. Privium keeps the ciphertexts on Solana itself.

Definition 4.4 (Crossing diff). The diff Diff_k is the byte string consisting of a header $(k, \rho^{(k-1)}, \rho^{(k)}, \sigma_k)$; the list of rewritten positions with their new leaves, sorted by position; and for each $a \in J_k$ the tuple $\big(\alpha(a), \Delta\Gamma_a^{(k)}, R_a^{(k)}, \pi_a^{\mathrm{val}}, \pi_a\big)$, where π_a is one aggregated Bulletproof (Section 3.6) over the commitments of all asset-a leaves rewritten in Crossing k and π_a^{val} is a 128-byte sigma proof, in the template of Protocol 3.1, that the random combination $\sum_\iota \varrho^\iota (C_\iota, D_\iota)$ of those leaves is a well-formed twisted ElGamal ciphertext; and the owner and auditor entry records of the Crossing (Chapter 11).

Before signing the Commit, the Coprocessor splits Diff_k into chunks and submits each as a `log_diff(k, j, chunk)` instruction. The program keeps the accumulator in the Crossing's state account and updates it in chunk order,

$$\delta_{k,0} = \mathsf{H}(\texttt{privium/da/v1} \,\|\, \mathrm{id}_{\mathrm{prog}} \,\|\, k), \qquad \delta_{k,j+1} = \mathsf{H}(\delta_{k,j} \,\|\, j \,\|\, \mathsf{chunk}_j), \tag{4.10}$$

rejecting any chunk whose index is not the next one. Check (C3) compares the final value with the δ_k inside the signed message and seals the accumulator.

Theorem 4.4 (Availability). If Commit k is accepted, then except with probability negligible under the collision resistance of H, the diff the Coprocessor signed is the in-order concatenation of the instruction data of successful Solana transactions invoking `log_diff` for Crossing k.

Proof. The accumulator changes only through `log_diff`, whose instruction data is recorded in the transaction that carried it, and the chunk index rule forces the order. If the signed δ_k equals the accumulator but the signed diff differs from the logged chunks, walk the chain (4.10) back from the final value: the first index at which the inputs differ while the outputs agree is a collision of H, by the same argument as Theorem 3.4. □

Corollary 4.2 (Reconstruction). $S^{(k)}$ is obtained from $S^{(0)}$ by applying $\mathsf{Diff}_1, \ldots, \mathsf{Diff}_k$ in order, so every committed Ledger state, every root and every inclusion proof can be recomputed by anyone from Solana's transaction history, without the Coprocessor and without its ciphertext store.

Recomputing $\rho^{(k)}$ from $S^{(k-1)}$ and Diff_k is how a watcher confirms root integrity, and summing commitment deltas from the same diff is check (ii) of Theorem 4.2. Root integrity thus holds under the integrity of the enclave and is auditable by anyone after every Commit. Chapter 10 describes the store that serves leaves and paths quickly; Corollary 4.2 is what makes that store a cache rather than a source of truth.

4.8 Replay, ordering and nonce discipline

Commits

Check (C1) makes the accepted Commits a single chain: each names its predecessor's root and the next index, the program stores exactly one root at a time, and a transaction carrying an old M_k fails (C1) because the stored index has moved past k. A second signature on the same M_k, which SUF-CMA forbids anyway, would also fail (C1). Every Commit writes the Ledger root account, so Sealevel's write lock executes Commits one at a time, while deposits, which write the deposit queue and the vaults but not the root account, proceed in parallel with each other. This is the read and write set scheduling that Encifher identifies as Solana's native dependency graph [2], and the Ledger needs nothing beyond it.

Deposits

Each deposit receives a sequence number in its queue shard from the program. A Commit absorbs exactly the entries of each shard between the stored cut and its new cut, `conserve` rejects a cut below the stored one, and (C1) orders the Commits, so every deposit is credited in exactly one Crossing.

Intents

Each trader's record holds a watermark ω_u, the largest Intent nonce applied so far. The enclave processes a trader's valid Intents in a Crossing in increasing order of ν, applies an Intent only if $\nu > \omega_u$ and the Crossing's slot does not exceed τ, and sets $\omega_u \leftarrow \nu$ after each applied Intent. The SDK draws ν from a per-trader counter, so concurrent orders from one trader simply carry consecutive nonces.

Lemma 4.2 (No replay). Under the integrity of the enclave and the SUF-CMA security of Ed25519, every Intent is applied at most once, and no Intent is applied that its trader did not sign.

Proof. Authenticity: an applied Intent carries a valid A_u signature over its fields, so an Intent the trader never signed is an Ed25519 forgery. Uniqueness: when an Intent with nonce ν is applied, ω_u becomes at least ν and never decreases, since the record is rewritten only by the enclave with the maximum. Any later presentation of the same Intent, in the same Crossing after sorting or in any later Crossing, has $\nu \le \omega_u$ and is rejected. An Intent sealed for an earlier key epoch fails to open under the current HPKE `info` (3.8), so epoch changes do not reopen old ciphertexts. □

AEAD nonces

Lemma 4.3 (Nonce uniqueness). No AES-256-GCM key in the Ledger is ever used twice with the same nonce.

Proof. Every leaf ciphertext at position ι in version v uses $n_{\iota,v} = \iota \,\|\, v$, a 96-bit string. The version of a leaf is the index k of the Crossing that wrote it, public and part of the authenticated data; a Crossing writes each position at most once, and Crossing indices strictly increase, so no (position, version) pair recurs, even when a relocation empties a position and a later Crossing fills it again. Distinct pairs give distinct nonces. Within one leaf the two ciphertexts use the same nonce under two different keys, k_L and $k_{u,a}$, and no owner leaf key is shared between traders, between assets or with the Ledger data key. □

Deterministic nonces remove the birthday bound that random 96-bit GCM nonces carry, and make Lemma 4.3 a property of the data structure that any watcher can check from the diffs.

4.9 Ledger growth

Let N_b be the number of balance leaves and N_r the number of trader records. The Ledger's on-chain footprint does not depend on either: it is the root account plus a fixed-size state per asset holding T_a and Γ_a, and the state account of the open Crossing, which carries the data availability accumulator and is closed after its Commit (Chapter 9). Everything that grows lives in Solana's transaction history and in caches built from it.

Full state. The leaves occupy $225N_b + 161N_r$ bytes. A sparse tree with $N = N_b + N_r$ randomly placed leaves in depth d shares its top $\log_2 N$ levels and has paths of about $d - \log_2 N$ unshared nodes per leaf below that, so its interior nodes take about $32N(d - \log_2 N + 2)$ bytes.

Per Crossing. With m_b balance leaves and m_r records rewritten, decoys included, and m_a of the balance leaves in asset a, the diff has

$$|\mathrm{Diff}_k| = 80 + 229\,m_b + 165\,m_r + \sum_{a \in J_k} \Big(196 + 32\big(2\lceil \log_2(64\,m_a)\rceil + 9\big)\Big) \text{ bytes}, \tag{4.11}$$

from an 80-byte header, a 4-byte position per leaf, 196 bytes per asset for $(\alpha(a), \Delta\Gamma_a, R_a, \pi_a^{\mathrm{val}})$ and the range proof size (3.7), with the entry records of Chapter 11 adding their own bytes. Each `log_diff` transaction carries about 3,800 bytes of chunk data (est.: the 4,096-byte v1 limit [4] less one signature, three account keys, the blockhash and the instruction header). The Ledger takes one diff per Crossing. A Crossing runs only when it settles something, at most once per batch interval Δ of Chapter 7, and an idle Ledger takes one heartbeat diff every 750 slots, with no asset (Chapter 10); its daily count lies between those two rates at the slot time Chapter 9 cites.

The table evaluates these formulas for an illustrative load; the counts are inputs to the formulas, not measurements.

Quantity	Example input	Result (est.)
Diff per Crossing	$m_b = 704$ over 11 assets of 64 each, $m_r = 256$	217,308 bytes
`log_diff` transactions per Crossing	same	58
Range proof per asset	$m_a = 64$	1,056 bytes
Escape inclusion proof	$d = 32$	1,024 bytes
Leaves, full state	$N_r = 10^6$ traders, $N_b = 4 \times 10^6$	1.06 GB
Interior nodes, full state	$N = 5 \times 10^6$, $d = 32$	about 1.9 GB
Positions available	$d = 32$	4,294,967,296

The per-Crossing figure follows from (4.11): $80 + 229 \cdot 704 + 165 \cdot 256 + 11 \cdot (196 + 1{,}056) = 217{,}308$ bytes, and $217{,}308/3{,}800$ rounds up to 58 transactions. Eleven assets of 64 leaves each is the largest Commit of Chapter 9 at the decoy floor of Chapter 8. The range proof term grows with the logarithm of the number of rewritten leaves, so batching more traders into a Crossing lowers the proof bytes per leaf: one 64-bit proof costs 672 bytes on its own and 16.5 bytes per leaf inside a batch of 64. The state figures show the Ledger fitting in the memory of a single enclave host at a million traders, with Corollary 4.2 guaranteeing that any party can rebuild it from the chain.

References

1. "Solana Privacy Rings", Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings

2. N. Lokhande, R. Gupta, R. Chahar, A. Jangra, A. Prajapati, M. Kumari, “Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana”, IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
3. Encifher, “Encifher core documentation: crates and coprocessor components”, accessed 2026-11-16. https://encifher-core-docs.vercel.app/docs/crates
4. Solana Foundation, “Larger Transaction Sizes” (SIMD-0296, v1 transactions of up to 4,096 bytes, live on mainnet from epoch 1035), 2026. https://solana.com/upgrades/larger-transaction-sizes
5. Anza, `solana-curve25519` crate documentation (syscall operations for curve25519 and ristretto255), accessed 2026-11-16. https://docs.rs/solana-curve25519
6. 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

Chapter 5

Trusted execution and attestation

The Coprocessor is the one component of Privium that sees plaintext: it decrypts Intents and Ledger leaves, crosses orders, and re-encrypts the result. This chapter makes that power checkable. It follows an Intel TDX quote from the CPU's measurement registers to the Intel root, binds the Coprocessor's keys to its measured code through REPORTDATA, registers those keys on Solana by verifying the quote's ECDSA-P256 signatures with the secp256r1 precompile, and checks an Ed25519 signature on every Commit against them. It closes with AWS Nitro Enclaves, reproducible builds, key epochs, sealing through the Council, and upgrade by re-attestation. Threshold mechanics are in Chapter 6; the Ledger state the enclave signs is in Chapter 4.

5.1 The Coprocessor's trust model

Encifher computes over encrypted Solana state inside a hardware Trusted Execution Environment, posts attested Merkle-anchored outputs, verifies enclave attestation on chain, and spreads key custody across a threshold committee so that no single enclave holds a master key [1]. Privium adopts that division of trust and makes each part explicit.

A Trusted Execution Environment here is a hardware-isolated execution context with three properties. **Isolation:** code and memory inside are inaccessible to the host, hypervisor and operator; Intel TDX runs each Trust Domain (TD) as a hardware-isolated virtual machine whose CPU state a TDX module in a dedicated CPU mode (SEAM) keeps from other software, with memory encrypted by the processor [2]. **Measurement:** the hardware records digests of what was loaded, which software can extend but never rewrite. **Attestation:** the hardware signs a statement binding the measurements to 64 bytes chosen by the code inside, verifiable by anyone who trusts the vendor's root key. The Coprocessor's guarantees reduce to these properties plus standard cryptographic assumptions:

Guarantee	What it rests on
Every Commit the Pool accepts was produced by code whose measurement is in the Registry	TDX isolation and measurement, the Intel attestation root, EUF-CMA of ECDSA-P256 and Ed25519, collision resistance of SHA-256 (Theorems 5.1 and 5.2)
The Coprocessor's signing and decryption keys exist only inside admitted enclaves	TDX isolation, the audited reproducible build (Section 5.7), and an honest threshold of the Council for keys released across machines (Section 5.8, Chapter 6)
Intents and Ledger plaintext are confidential to admitted enclaves	the two rows above, and IND-CCA2 security of HPKE and AES-256-GCM (Chapter 3)
Funds remain claimable if every enclave stops	the Escape, which depends on the Council and the last committed root, not on the enclave (Chapter 6)

Attestation protects confidentiality and the authenticity of state transitions; custody never depends on the enclave staying alive.

5.2 Measurement and the chain of trust

Measurement registers

A TD Quote reports the measurement registers of the TD and of the TDX module itself. The fields relevant to Privium are taken from the version 4 TD Quote Body [3]:

- **MRTD** (48 bytes, SHA-384): the initial contents of the TD, here the virtual firmware.
- **RTMR0 to RTMR3** (48 bytes each, SHA-384): runtime-extendable registers; by convention RTMR0 and RTMR1 are extended by the firmware, RTMR2 by the operating system and RTMR3 by the workload owner [3].
- **TDATTRIBUTES** (8 bytes): bit 0 is DEBUG; when set, the host can read the TD's CPU state and private memory and the TD is untrusted [3].
- **XFAM** (8 bytes): the extended CPU features available to the TD.
- **REPORTDATA** (64 bytes): data chosen by the TD, covered by the quote signature [3].

An RTMR starts at zero and is changed only by an extend operation, which replaces it with the SHA-384 digest of its old value concatenated with a 48-byte input [5]:

$$\mathrm{RTMR}_j \leftarrow \mathrm{SHA384}\big(\mathrm{RTMR}_j \,\|\, d\big). \tag{5.1}$$

Writing $h_i = \mathrm{SHA384}(h_{i-1} \| d_i)$ from $h_0 = 0^{384}$, if two different extend sequences reach the same value, then walking both chains back from the end, the first step whose inputs (h_{i-1}, d_i) differ while outputs agree is a SHA-384 collision; the register commits to the whole ordered boot log.

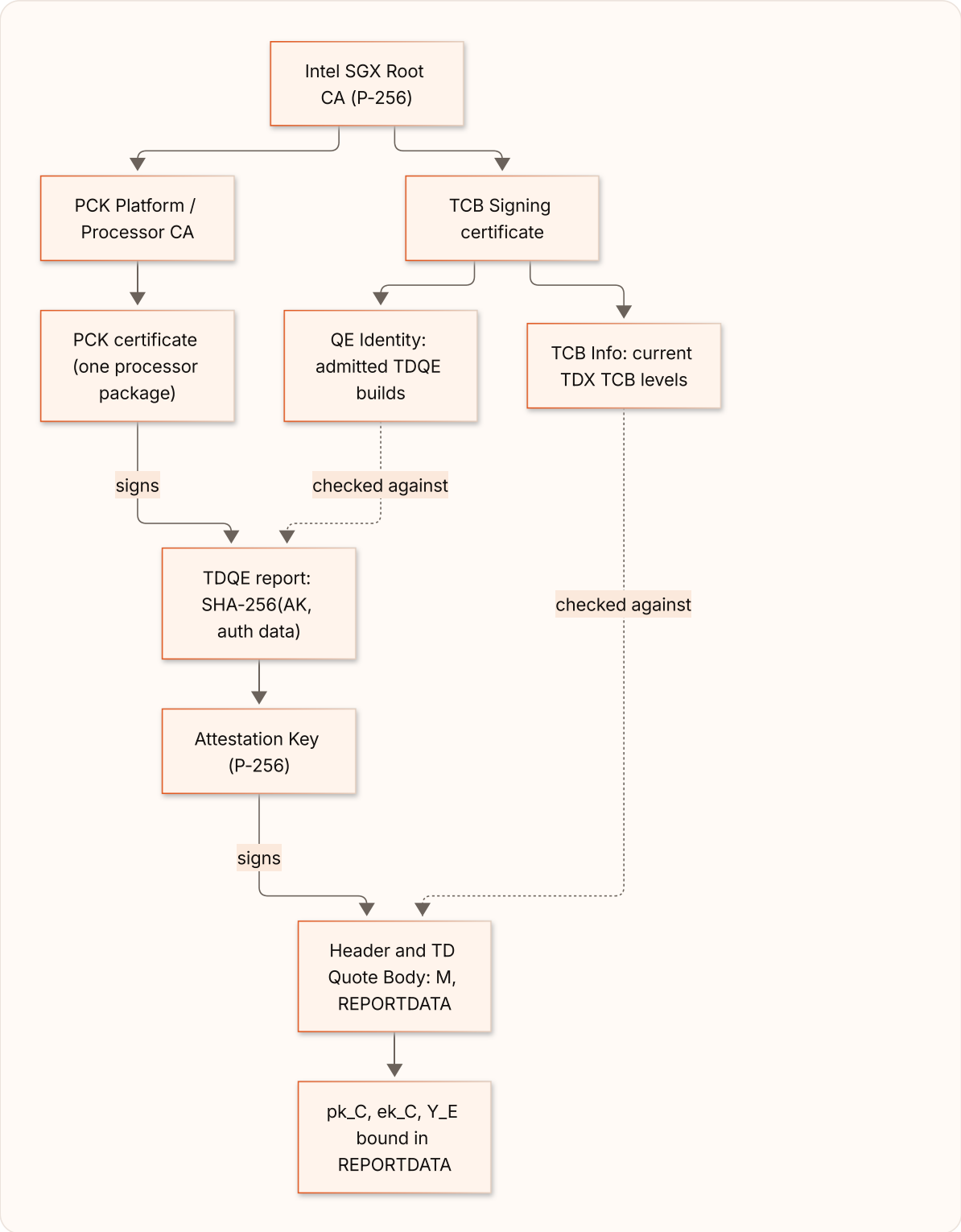
Privium's measurement vector

Privium boots the Coprocessor as a minimal TD: virtual firmware, a kernel and an initramfs that contains the statically linked Coprocessor binary and nothing that reads further code from disk. We define the measurement vector

$$M = \big(\mathrm{MRTD},\ \mathrm{RTMR}_0,\ \mathrm{RTMR}_1,\ \mathrm{RTMR}_2,\ \mathrm{XFAM}\big), \tag{5.2}$$

and the Registry stores a set $\mathcal{M}$ of admitted vectors. The Coprocessor never extends RTMR3 and reads no owner configuration, so RTMR3, MRCONFIGID, MROWNER and MROWNERCONFIG must be zero: any runtime extension or operator-supplied configuration changes the quote and fails admission.

The chain of trust runs as follows. The Intel SGX Root CA certifies a PCK Platform or Processor CA, which certifies the Provisioning Certification Key (PCK) of one processor package; every certificate is X.509 with `ecdsa-with-SHA256` over P-256 (`prime256v1`) keys [4]. The PCK, held only by the Intel-signed Provisioning Certification Enclave (PCE), signs the report of the TD Quoting Enclave (TDQE), whose data field holds the SHA-256 digest of the Attestation Key (AK); the AK, a P-256 key generated inside the TDQE, signs the quote header and body, which carry M and REPORTDATA, which in turn carries the digest of the Coprocessor's keys (Section 5.4) [3]. A second branch, the TCB Signing certificate under the same root, signs the TCB Info and QE Identity collateral that state which TCB levels and TDQE builds are current [3][4].



5.3 The TDX quote and its signing chain

A version 4 TD Quote has a fixed-length prefix and a variable-length signature section [3]. All integers are little-endian.

Region	Offset	Size (bytes)	Content
Header	0	48	version = 4, attestation key type = 2 (ECDSA-256 with P-256), TEE type = `0x81` (TDX), QE vendor ID, user data
TD Quote Body	48	584	TEE_TCB_SVN, MRSEAM, MRSIGNERSEAM, SEAMATTRIBUTES, TDATTRIBUTES (at 168), XFAM, MRTD (at 184), MRCONFIGID, MROWNER, MROWNERCONFIG, RTMR0 to RTMR3 (from 376), REPORTDATA (at 568)
Signature data length	632	4	length L of what follows
Quote signature	636	64	ECDSA P-256 by the AK over bytes $[0, 632)$, $r \| s$
AK public key	700	64	$x \| y$
QE certification data	764	variable	type 6: the 384-byte TDQE report, its 64-byte PCK signature, QE authentication data, then type 5: the PEM PCK certificate chain

Body offsets follow from the field sizes in [3]. The TDQE report is an SGX report body whose MRSIGNER (offset 128), ISVPRODID (256), ISVSVN (258) and REPORTDATA (320) identify the quoting enclave; its REPORTDATA is $\mathrm{SHA256}(\mathsf{AK} \| \mathsf{auth}) \| 0^{256}$ [3].

Definition 5.1 (Quote acceptance). Given quote Q, collateral coll (PCK chain, TCB Info, QE Identity, CRLs), admitted set $\mathcal{M}$ and expected value $rd^{\star}$, $\mathsf{Accept}(Q, \mathsf{coll}, \mathcal{M}, rd^{\star})$ holds if and only if:

1. the header has version 4, key type 2 and TEE type `0x81`;
2. the PCK certificate chains to the pinned Intel root key, no certificate in the chain is revoked, and the TCB level derived from TEE_TCB_SVN and the PCK certificate is current under TCB Info;
3. the TDQE report verifies under the PCK, and its MRSIGNER, ISVPRODID and ISVSVN satisfy QE Identity;
4. $\mathrm{SHA256}(\mathsf{AK} \| \mathsf{auth}) \| 0^{256}$ equals the TDQE report's REPORTDATA;
5. the quote signature verifies under AK over $Q[0, 632)$;
6. TDATTRIBUTES bit 0 (DEBUG) is zero, MRCONFIGID, MROWNER, MROWNERCONFIG and RTMR3 are zero, and $M(Q) \in \mathcal{M}$;
7. the REPORTDATA field of Q equals $rd^{\star}$.

Intel's own Quote Verification Library performs checks 1 to 5 and leaves the content checks in 6 and 7 to the verifier [3]. Privium performs all seven on chain, split across platform admission and quote admission (Section 5.5).

ECDSA signatures are malleable, and the secp256r1 precompile rejects signatures whose s exceeds half the group order [8] [10]. The quote format places no low-S requirement on its signers [3], so the relayer normalizes s before submission. The following lemma shows this leaves every signature in the chain valid.

Lemma 5.1 (Low-S normalization). Let n_{256} be the order of the P-256 group and (r, s) a valid ECDSA signature on digest z under public key P. Then $(r, n_{256} - s)$ is also valid, and exactly one of $s, n_{256} - s$ is at most $\lfloor n_{256}/2 \rfloor$.

Proof. Verification computes $R' = s^{-1} z \cdot G_{256} + s^{-1} r \cdot P$ and accepts if $x(R') \equiv r \pmod{n_{256}}$. Replacing s by $-s \bmod n_{256}$ replaces R' by $-R'$, which has the same x-coordinate. Since n_{256} is odd and $s \neq 0$, exactly one of s and $n_{256} - s$ lies in $[1, \lfloor n_{256}/2 \rfloor]$. □

The precompile also takes the public key in 33-byte SEC1 compressed form [8][10], while the quote carries the AK as 64 bytes $x \| y$. Compression is $\mathsf{cmp}(x, y) = (\mathtt{0x02} + (y \bmod 2)) \| x$; the program recomputes it from the AK it already trusts, so no new trust enters.

5.4 Binding keys to the measurement

At the start of key epoch e, a Coprocessor instance generates three key pairs inside the TD from the CPU's hardware random source:

- (sk_C, pk_C), Ed25519: signs Crossing opens, Residual orders and Commits (Chapter 9) and Receipts (Chapter 11), each message under its own domain tag;

- (dk_C, ek_C), X25519: the HPKE recipient key to which traders seal Intents, deposit notes and account registration (Chapter 3);
- $(y_E, Y_E = y_E G)$ in $\mathbb{G}$: the key to which the Council re-encrypts the Ledger data key on release (Section 5.8, Chapter 6).

It binds them to the quote by setting

$$rd = \mathsf{H}\big(\texttt{"privium/attest/v1"} \,\|\, \mathsf{id}_{mathrmprog} \,\|\, e \,\|\, \sigma \,\|\, \eta_\sigma \,\|\, pk_C \,\|\, ek_C \,\|\, Y_E\big) \,\|\, 0^{256}, \tag{5.3}$$

where H is SHA-256 (a Solana syscall, so the program recomputes (5.3) cheaply), $\mathsf{id}_{\mathrm{prog}}$ is the Pool program ID, and (σ, η_σ) is a recent slot and its slot hash. All fields have fixed length, so the encoding is injective; the zero half mirrors Intel's convention for the AK digest [3].

```rust
use curve25519_dalek::{constants::RISTRETTO_BASEPOINT_POINT as G, Scalar};
use sha2::{Digest, Sha256};

pub const ATTEST_DOMAIN: &[u8; 17] = b"privium/attest/v1";

pub struct EpochKeys {
    pub commit: ed25519_dalek::SigningKey,   // sk_C
    pub intent: x25519_dalek::StaticSecret,  // dk_C
    pub release: Scalar,                     // y_E
}

pub struct Anchor { pub program_id: [u8; 32], pub epoch: u64, pub slot: u64, pub slot_hash: [u8; 32] }

impl EpochKeys {
    pub fn public(&self) -> ([u8; 32], [u8; 32], [u8; 32]) {
        (
            self.commit.verifying_key().to_bytes(),
            x25519_dalek::PublicKey::from(&self.intent).to_bytes(),
            (self.release * G).compress().to_bytes(),
        )
    }

    /// Equation (5.3).
    pub fn report_data(&self, a: &Anchor) -> [u8; 64] {
        let (pk_c, ek_c, y_e) = self.public();
        let digest = Sha256::new()
            .chain_update(ATTEST_DOMAIN)
            .chain_update(a.program_id)
            .chain_update(a.epoch.to_le_bytes())
            .chain_update(a.slot.to_le_bytes())
            .chain_update(a.slot_hash)
            .chain_update(pk_c)
            .chain_update(ek_c)
            .chain_update(y_e)
            .finalize();
        let mut rd = [0u8; 64];
        rd[..32].copy_from_slice(&digest);
        rd
    }
}
```

The enclave passes `report_data` to the TDREPORT call, and the host's TDQE turns the report into a signed quote. The binding ties each public key to the measured code that generated it and ties the quote to a point in chain time.

Lemma 5.2 (Key binding). A tuple $(\mathsf{id}'_{mathrmprog}, e', \sigma', \eta', pk', ek', Y')$ that differs from a TD's inputs yet yields the same rd is a SHA-256 collision, since the fixed-width encoding is injective. □

Lemma 5.3 (Freshness). If the program accepts rd only when η_σ equals the SlotHashes entry for slot σ, the quote was generated after slot σ was produced.

Proof. Computing rd requires η_σ, a hash of bank state that no party can compute before the slot exists. SlotHashes keeps the latest 512 entries [12], so the program checks the anchor at staging, while it is recorded. □

5.5 On-chain registration

Registration turns a quote into an `EnclaveKey` record (e, pk_C, ek_C, Y_E, M) with the validity interval $[\,eE_{\mathrm{key}},\ (e+1)E_{\mathrm{key}})$ in slots of its epoch (Section 5.8); the record stays on chain after its epoch ends, so signatures made under it remain checkable. The PCK chain and collateral change only when Intel publishes new collateral or a machine joins, while quotes come every key

epoch, so Privium splits registration into **platform admission** (per processor package and collateral period) and **quote admission** (per enclave and epoch).

Platform admission

Platform admission covers check 2 and the QE Identity part of check 3: X.509 chain, CRLs, TCB Info. These are kilobytes of DER and JSON, while the precompile verifies only messages carried in the same transaction (size limits in Chapter 9). Privium runs the standard DCAP collateral checks in a zero-knowledge virtual machine and posts a Groth16 proof with public inputs

$$\big(\mathsf{H}(\mathsf{RootCA}),\ \mathsf{cmp}(\mathsf{PCK}),\ \mathsf{FMSPC},\ \mathsf{tcbStatus},\ \mathsf{qeIdentity},\ t_{\mathrm{exp}}\big),$$

where t_{exp} is the earliest `nextUpdate` across the collateral. The program verifies the proof with the alt_bn128 syscalls, which the `groth16-solana` verifier does in under 200,000 compute units [13], checks that $\mathsf{H}(\mathsf{RootCA})$ equals the constant compiled into the program, and writes a `Platform` account keyed by the PCK hash. The account is valid until t_{exp} and only while $\mathsf{tcbStatus}$ is `UpToDate`. Automata's DCAP verifier follows the same zkVM route on other chains [14].

Quote admission with the secp256r1 precompile

Solana's secp256r1 program (`Secp256r1SigVerify1111111111111111111111111`, SIMD-0075) verifies ECDSA over P-256 with SHA-256 [8]. Its feature gate `srremy31J5Y25FrAApwVb9kZcfXbusYMMsvTK9aWv5q` is active on mainnet-beta, with activation recorded at slot 345,600,000 in the feature account (read 2026-11-16) [9]. Its instruction data is a count byte, a padding byte, then per signature seven little-endian `u16` fields (signature, public key and message offsets with their instruction indices, and the message length), so data begins at byte 16; signatures are 64 bytes and keys 33-byte compressed points, and an index of `0xFFFF` points into the precompile instruction itself [8][10]. The Ed25519 program (`Ed25519SigVerify111111111111111111111111`, live on mainnet-beta [9]) uses the same layout with 32-byte keys [11].

Privium uses one pattern for both: exactly one signature, all three indices `0xFFFF`, and the program reads key and message back out of the precompile instruction through the Instructions sysvar. Since it never trusts offsets into its own data, no offset trick can make the precompile verify one message while the program parses another (general pattern and cost in Chapter 9).

Quote admission runs in three transactions:

1. **Stage.** The relayer submits $(e, \sigma, pk_C, ek_C, Y_E)$. The program checks η_σ against SlotHashes, computes $rd^\star$ by (5.3), and creates a pending `EnclaveKey` account holding it.
2. **QE.** A secp256r1 instruction verifies the PCK signature over the 384-byte TDQE report. The program checks that the key equals the `Platform` account's PCK, that MRSIGNER, ISVPRODID and ISVSVN satisfy its QE Identity, and that the report's last 32 REPORTDATA bytes are zero; it records h_{AK}, the first 32.
3. **Quote.** A secp256r1 instruction verifies the AK signature over the 632-byte prefix. The program checks the prefix against Definition 5.1, items 1, 6 and 7, and checks $\mathrm{SHA256}(\mathsf{AK} \| \mathsf{auth}) = h_{\mathsf{AK}}$.

The third transaction is the largest: 745 bytes of precompile data (16-byte header, signature, compressed key, 632-byte prefix) plus a program instruction carrying the AK's y-coordinate, a discriminator and the QE authentication data (est. 44 bytes plus that data), sent as a v1 transaction, whose 4,096-byte limit Chapter 9 cites (est.). The AK's x-coordinate is read from the compressed key, not resent.

```rust
// Accounts: registry, platform (current), staged EnclaveKey (Stage::QeVerified), Instructions sysvar.
pub fn admit_quote(ctx: Context<AdmitQuote>, ak_y: [u8; 32], qe_auth: Vec<u8>) -> Result<()> {
    let v = precompiled(&ctx.accounts.ixs, &SECP256R1_ID)?; // one sig, all indices 0xFFFF
    let q = v.message;
    require!(q.len() == 632, PrivErr::QuoteFormat);
    require!(u16le(q, 0) == 4 && u16le(q, 2) == 2 && u32le(q, 4) == 0x81, PrivErr::QuoteFormat);

    // The precompile's key must be the AK certified in the QE step.
    let ak_x: [u8; 32] = v.pubkey[1..33].try_into().unwrap();
    require!(v.pubkey[0] == 0x02 + (ak_y[31] & 1), PrivErr::AkEncoding);
    let h_ak = hashv(&[&ak_x, &ak_y, &qe_auth]).to_bytes();
    let k = &mut ctx.accounts.staged;
    require!(h_ak == k.ak_hash, PrivErr::AkBinding);

    // Content checks: Definition 5.1, items 6 and 7.
    require!(q[168] & 0x01 == 0, PrivErr::DebugTd);
    require!(is_zero(&q[232..376]) && is_zero(&q[520..568]), PrivErr::OwnerFields);
    let m = Measurement::from_quote(q); // MRTD, RTMR0..2, XFAM
    require!(ctx.accounts.registry.admits(&m), PrivErr::Measurement);
    require!(q[568..632] == k.report_data, PrivErr::ReportData);

    k.measurement = m.digest();
    k.stage = Stage::Admitted;
    emit!(EnclaveAdmitted { epoch: k.epoch, commit_key: k.commit_key, measurement: k.measurement });
    Ok(())
}
```

`precompiled` loads the instruction immediately before the current one, checks its program ID, requires a count of one and all three indices equal to `0xFFFF`, and returns the key and message slices at the stated offsets. Quote offsets 232 to 376 cover MRCONFIGID, MROWNER and MROWNERCONFIG; 520 to 568 is RTMR3. An admitted key becomes the active Commit key at the start of epoch e (Section 5.8).

Theorem 5.1 (Admission soundness). Assume (A1) the TDX module, TDQE and PCE behave as Intel specifies and PCK private keys never leave the PCE; (A2) ECDSA-P256 is EUF-CMA; (A3) SHA-256 is collision resistant; (A4) the platform-admission proof system is knowledge-sound. If the Registry marks (e, pk_C, ek_C, Y_E) as admitted, then, except with probability at most $q_s \cdot \mathbf{Adv}^{\mathrm{euf}}_{\mathrm{P256}} + \mathbf{Adv}^{\mathrm{cr}}_{\mathrm{SHA256}} + \mathbf{Adv}^{\mathrm{ks}}$, where q_s is the number of P-256 keys in the chain, a TD with DEBUG clear and measurement $M \in \mathcal{M}$ ran on a current, Intel-certified platform after slot σ and placed exactly (pk_C, ek_C, Y_E) in its REPORTDATA.

Proof sketch. Game hops from the real game G_0. G_1 aborts if the Platform PCK does not chain to the pinned root under current collateral: by (A4) a witness chain is extractable, and an accepting chain without genuine issuance contains a forgery under an Intel CA key. G_2 aborts if an accepted TDQE report was never signed by the PCE, which by (A1) is a PCK forgery. G_3 aborts if the accepted AK differs from the one the TDQE certified, a SHA-256 collision on h_{AK}. G_4 aborts if the accepted 632-byte prefix was never signed by that TDQE; by (A1) the TDQE signs only bodies the TDX module produced for a real TD, so this is an AK forgery. In G_4 the M, DEBUG bit and REPORTDATA are those of a real TD; Lemmas 5.2 and 5.3 finish. Summing the hops gives the bound. □

Per-Commit signatures with the Ed25519 program

Quote verification happens once per epoch. Each Commit needs only an Ed25519 check against the admitted pk_C. The Coprocessor signs the Commit digest

$$\mu^{(k)} = \texttt{privium/commit/v1} \,\|\, \mathsf{id}_{\mathrm{prog}} \,\|\, e \,\|\, k \,\|\, \rho^{(k-1)} \,\|\, \rho^{(k)} \,\|\, \mathsf{body}^{(k)}, \tag{5.4}$$

the Commit message M_k of Chapter 4 (Equation 4.6), where $\mathsf{body}^{(k)}$ is the public part of the Commit laid out in Chapter 9 (Equation 9.1). Ed25519 hashes the message internally, so the program reads the fields from the signed bytes themselves. The signature travels in an Ed25519 program instruction before the Commit instruction, not as a transaction signer, so any relayer can submit the Commit, and the root it certifies is what a Receipt later points to (Chapter 11).

```rust
pub fn check_commit_signature(
    ixs: &AccountInfo,
    active: &EnclaveKey,
    prev_root: &[u8; 32],
    new_root: &[u8; 32],
    k: u64,
    slot: u64,
) -> Result<Vec<u8>> {
    require!(active.stage == Stage::Admitted, PrivErr::Stage);
    require!(active.valid_from <= slot && slot < active.valid_until, PrivErr::KeyExpired);
    let v = precompiled(ixs, &ed25519_program::ID)?; // one sig, all indices 0xFFFF
    require!(v.pubkey == active.commit_key.as_ref(), PrivErr::CommitKey);

    let prefix = [
        b"privium/commit/v1".as_slice(),
        crate::ID.as_ref(),
        &active.epoch.to_le_bytes(),
        &k.to_le_bytes(),
        prev_root,
        new_root,
    ]
    .concat();
    require!(v.message.starts_with(&prefix), PrivErr::CommitDigest);
    Ok(v.message[prefix.len()..].to_vec())                // body^(k), parsed by the Commit
}
```

Theorem 5.2 (Commit authenticity). Assume Theorem 5.1, EUF-CMA of Ed25519, and TDX isolation of sk_C inside the admitted TD. If the Pool accepts a Commit with index k in epoch e, then an admitted enclave of epoch e signed $\mu^{(k)}$ for exactly this $(\rho^{(k-1)}, \rho^{(k)}, \mathsf{body}^{(k)})$, except with probability $\mathbf{Adv}^{\mathrm{euf}}_{\mathrm{Ed25519}}$ beyond the bound of Theorem 5.1.

Proof sketch. By Theorem 5.1 the key the program compares against was generated in an admitted TD, and by isolation sk_C never left it. A valid signature on a $\mu^{(k)}$ the enclave never signed is a forgery. The fields of $\mu^{(k)}$ have fixed widths and the body has a fixed layout, so one message names one tuple of arguments. Because $\mu^{(k)}$ contains $\rho^{(k-1)}$, k, e and $\mathsf{id}_{\mathrm{prog}}$, a signature cannot be replayed onto another root, index, epoch or deployment. □

Conservation is checked in the same transaction against vault balances and the commitment accumulators (Chapter 4), so an accepted Commit is both authentic and conserving.

5.6 AWS Nitro Enclaves as a second platform

AWS Nitro Enclaves satisfies the same model with a different chain. The Nitro Hypervisor issues a CBOR attestation document signed as COSE_Sign1 with ECDSA P-384 (COSE `-35`), containing a timestamp, SHA-384 PCRs, the signing certificate, a CA bundle, and optional `public_key`, `user_data` and `nonce` fields of up to 1,024 bytes each [6]. The root is the 30-year AWS Nitro Enclaves Root-G1 certificate, pinned by its published SHA-256 fingerprint [6]. PCR0, PCR1 and PCR2 measure the image, kernel and application and are printed by `nitro-cli build-enclave`; debug-mode enclaves report all-zero PCRs [7].

	Intel TDX	AWS Nitro Enclaves
Isolation unit	Trust Domain (VM)	enclave carved from an EC2 instance
Code measurement	MRTD, RTMR0 to RTMR2 (SHA-384)	PCR0 to PCR2 (SHA-384)
Debug signal	TDATTRIBUTES bit 0	all-zero PCRs
Key binding field	REPORTDATA, 64 bytes	`user_data`, up to 1,024 bytes
Signing chain	P-256 ECDSA, Intel SGX Root CA	P-384 ECDSA, AWS Nitro Root-G1
On-chain path	platform proof, then secp256r1 precompile per quote	Groth16 proof per document

On Nitro, $M = (\mathrm{PCR}_0, \mathrm{PCR}_1, \mathrm{PCR}_2)$ and `user_data` carries rd from (5.3). Signatures over P-384 take the zkVM route: a Groth16 proof attests that the COSE signature and chain verify to the pinned Root-G1 and outputs $(M, \mathsf{user_data}, \mathsf{timestamp})$. Everything after admission is identical. Running both platforms gives the Registry two independent hardware roots, and the Council can release keys only to code admitted on both.

5.7 Reproducible builds

A measurement in $\mathcal{M}$ is meaningful only if anyone can derive it from source. A build is reproducible when, given the same source, build environment and instructions, any party can recreate bit-for-bit identical artifacts [15]. Privium's Coprocessor image is built that way:

1. The toolchain is pinned by content hash: the Rust compiler, the C toolchain for the kernel, and every crate through `Cargo.lock` with vendored sources.
2. The binary is statically linked with fixed codegen units, `SOURCE_DATE_EPOCH`, path remapping and no build IDs, so the output is independent of the build machine.
3. The kernel, initramfs and firmware are assembled by a script that sorts entries and zeroes timestamps.
4. A measurement tool replays the firmware's and loader's extend sequence over the artifacts using (5.1) and prints M.

The Registry stores, next to each $M \in \mathcal{M}$, the source commit and the build manifest digest; a reviewer builds, measures and compares. Theorem 5.1 then speaks about source code rather than an opaque binary: the keys in the Registry were generated by the reviewed program. Chapter 10 describes the build pipeline.

5.8 Key epochs, rotation and sealing

Epochs

Keys live for one epoch of E_{key} slots, recommended $E_{\text{key}} = 216{,}000$ (parameter): re-attestation costs a handful of transactions, so a short epoch is cheap and picks up new TCB collateral promptly. For each Coprocessor instance the Registry lists the record of epoch e and at most one staged record for $e+1$, so the two instances of Chapter 10 keep at most four keys live; the Pool accepts a signature only under a key whose validity interval contains the current slot. SDKs read ek_C from the Registry; Intents to the previous key are accepted for one batch interval Δ past the boundary, after which the enclave erases dk_C.

Sealing through the Council

The Coprocessor holds one long-lived secret: the Ledger data key k_L, the AES-256-GCM key under which Ledger ciphertexts are encrypted (Chapter 4). Hardware sealing would bind k_L to one machine, which makes failover and upgrade depend on that machine. Privium seals k_L to the Council instead. At genesis the first admitted enclave samples k_L and publishes a wrapped form

$$\mathsf{wrap}(k_L) = \big(c_1,\ \mathsf{AE.Enc}(\mathsf{H}(c_1 \| Z),\ k_L)\big), \qquad c_1 = r\mathcal{P}_C, \quad Z = rH, \tag{5.5}$$

where $\mathcal{P}_C = s^{-1}H$ is the Council's public key (Chapter 6), so that $sc_1 = Z$. A later enclave obtains k_L by re-encryption: Council members check on chain that Y_E belongs to an admitted Registry record, each publishes a share re-encrypting the handle c_1 from $\mathcal{P}_C$ to Y_E with a proof, and the enclave combines t shares into $A = \kappa G$, $B = sc_1 + \kappa Y_E$ and recovers $Z = sc_1 = B - y_E A$ (protocol and proofs in Chapter 6). So k_L is available exactly to enclaves whose measurement is in $\mathcal{M}$, under enclave isolation and an honest threshold of the Council. AWS KMS offers a PCR-conditioned analog on Nitro [7]; Privium uses the Council path on both platforms so one mechanism governs release everywhere.

Rotation scopes every key to one epoch: an epoch's sk_C signs only inside that epoch's validity interval, each still subject to the conservation checks, and its dk_C opens only that epoch's Intents.

5.9 Upgrade by re-attestation

Code changes enter Privium only as new measurements. An upgrade is the following sequence.

1. **Propose.** The upgrade authority (Chapter 9) submits M' with its source commit and manifest digest, pending until $s_{\text{act}} = s_{\text{now}} + T_{\text{up}}$, recommended $T_{\text{up}} = 2W$ (parameter): every trader can reproduce M' and, if they choose, Exit under the current code before M' can receive any key.
2. **Activate.** After s_{act}, $M' \in \mathcal{M}$; new enclaves are admitted by Section 5.5 and receive k_L from the Council.
3. **Hand over.** The new key is staged for the next epoch, and its first Commit chains from the last root signed under the old key, since (5.4) includes $\rho^{(k-1)}$.
4. **Retire.** The authority removes M from $\mathcal{M}$, effective at the next epoch boundary.

Theorem 5.3 (Upgrade transparency). Under the assumptions of Theorem 5.1 and an honest threshold of the Council, every enclave that ever holds k_L or signs an accepted Commit runs code whose measurement was recorded on chain at least T_{up} slots before it received k_L.

Proof. An enclave holds k_L only by Council release or by having generated it at genesis. Release requires t shares, at least one from an honest member, and honest members release only to Y_E in an admitted Registry entry. Admission requires $M \in \mathcal{M}$, and M enters $\mathcal{M}$ only at $s_{\text{act}} = s_{\text{prop}} + T_{\text{up}}$. Signing an accepted Commit requires an admitted pk_C by Theorem 5.2, which again requires $M \in \mathcal{M}$. The genesis measurement is the first entry of $\mathcal{M}$ and is recorded before the Pool accepts deposits. □

When the enclave stops producing Commits, custody passes to the Escape, which uses the Council and the last committed root and no enclave at all (Chapter 6).

References

1. N. Lokhande, R. Gupta, R. Chahar, A. Jangra, A. Prajapati, M. Kumari, "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
2. "Intel Trust Domain Extensions (Intel TDX)", Intel Corporation, accessed 2026-11-16. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html
3. "Intel Trust Domain Extensions Data Center Attestation Primitives (Intel TDX DCAP): Quote Generation Library and Quote Verification Library", Rev 0.9, Intel Corporation, December 2023 (Appendix A.3, Version 4 Quote Format). https://download.01.org/intel-sgx/sgx-dcap/1.21/linux/docs/Intel_TDX_DCAP_Quoting_Library_API.pdf
4. "Intel SGX PCK Certificate and Certificate Revocation List Profile Specification", Rev 1.4, Intel Corporation, March 2020. https://download.01.org/intel-sgx/sgx-dcap/1.21/linux/docs/SGX_PCK_Certificate_CRL_Spec-1.4.pdf
5. TianoCore EDK II, `MdePkg/Library/TdxLib/Rtmr.c` (RTMR extend via TDG.MR.RTMR.EXTEND), 2025. https://sources.debian.org/src/edk2/2025.02-8/MdePkg/Library/TdxLib/Rtmr.c
6. "Verifying the root of trust", AWS Nitro Enclaves User Guide, Amazon Web Services, accessed 2026-11-16. https://docs.aws.amazon.com/enclaves/latest/user/verify-root.html
7. "Cryptographic attestation", AWS Nitro Enclaves User Guide, Amazon Web Services, accessed 2026-11-16. https://docs.aws.amazon.com/enclaves/latest/user/set-up-attestation.html
8. "SIMD-0075: Precompile for secp256r1 sigverify", Solana Foundation, Solana Improvement Documents. https://github.com/solana-foundation/solana-improvement-documents/blob/main/proposals/0075-precompile-for-secp256r1-sigverify.md
9. "Feature Gate: Enable secp256r1 precompile, SIMD #0075", Anza, agave issue #3656; feature account `srremy31J5Y25FrAApwVb9kZcfXbusYMMsvTK9aWv5q` and program accounts read from mainnet-beta RPC on 2026-11-16. https://github.com/anza-xyz/agave/issues/3656 and https://explorer.solana.com/address/srremy31J5Y25FrAApwVb9kZcfXbusYMMsvTK9aWv5q
10. `solana-secp256r1-program` crate documentation, Anza, 2025. https://docs.rs/solana-secp256r1-program/latest/solana_secp256r1_program/
11. `solana-ed25519-program` crate documentation, Anza, 2025. https://docs.rs/solana-ed25519-program/latest/solana_ed25519_program/
12. `solana-slot-hashes` crate, `MAX_ENTRIES`, Anza, 2025. https://docs.rs/solana-slot-hashes/latest/src/solana_slot_hashes/lib.rs.html
13. `groth16-solana`, Light Protocol, 2025. https://lib.rs/crates/groth16-solana
14. "Automata DCAP Attestation", Automata Network documentation, accessed 2026-11-16. https://docs.ata.network/tee-overview/tee-verifiers/intel-sgx-tdx-dcap/automata-dcap-attestation
15. "Definitions", Reproducible Builds project, accessed 2026-11-16. https://reproducible-builds.org/docs/definition/

Chapter 6

Threshold keys and the Escape

The Council is the second root of Privium's trust and the only one custody rests on. It holds a single secret $s \in \mathbb{Z}_\ell^*$, split t-of-n among its members so that no coalition below the threshold learns anything about it. The Council has two duties. It releases the Ledger data key to enclaves the Registry has admitted (Chapter 5), and it runs the Escape: if no Commit lands within the liveness window W, it re-encrypts each owner's last committed balance to that owner alone, and the Pool program pays the owner out on a proof that never touches an enclave. This chapter builds the Council's key from first principles: Shamir sharing, Feldman and Pedersen verifiable sharing, distributed key generation, threshold decryption with proofs of correct decryption, re-encryption to a verified key, and proactive resharing as membership changes. It then specifies the Escape step by step and proves its safety and liveness. Group and proof preliminaries are in Chapter 3; the Ledger leaf whose Council handle the Escape decrypts is in Chapter 4.

6.1 What the Council holds

Encifher places decryption keys with a threshold committee so that no single enclave holds a master key [11]; Privium's Council is that committee with a second duty, the Escape. The Council's public key is the twisted ElGamal key $P_{\mathcal{C}} = s^{-1}H$ of Definition 3.8, so that $sP_{\mathcal{C}} = H$. Member $j \in \{1, \dots, n\}$ holds a share s_j and has a public verification key $S_j = s_j P_{\mathcal{C}}$; both $P_{\mathcal{C}}$ and every S_j are stored in the Council account on chain. Two objects are encrypted under $P_{\mathcal{C}}$:

- the wrapped Ledger data key $\mathsf{wrap}(k_L)$ (Chapter 5, equation 5.5);
- the Council handle $D = rP_{\mathcal{C}}$ of every Ledger leaf (C, D) (Definition 4.2), which decrypts as

$$C - sD = b_{u,a}^{(k)} G. \tag{6.1}$$

The leaf's owner tag commits to the owner's escape key $Y_u = y_u G$ (Chapter 4). The SDK derives y_u deterministically from the trader's wallet (Chapter 10), so a trader who keeps their wallet keeps their escape key.

The Council never computes (6.1) in public. It only re-encrypts, either to an admitted enclave's key Y_E or to an owner's key Y_u. The guarantees are:

Property	Rests on
No coalition of fewer than t members learns s, k_L or any balance	perfect secrecy of Shamir sharing, hiding of Pedersen VSS, DDH in $\mathbb{G}$
No Escape payout goes to anyone but the owner, or exceeds the committed balance	Merkle binding, soundness of the share and claim proofs, Commit authenticity (Theorem 5.2), the proofs of Theorem 4.2; this holds against a fully corrupt Council
Every owner completes the Escape	t honest members live on chain

Parameters. $n = 9$, $t = 5$ (parameter): confidentiality tolerates four colluding members and the Escape completes with four offline; $n = 2t - 1$ is the honest-majority setting in which the key generation of Section 6.4 is proven secure [4], and $2t > n$ means any two quorums intersect.

6.2 Shamir sharing

To share $x \in \mathbb{Z}_\ell$ with threshold t, the dealer samples $a_1, \dots, a_{t-1} \leftarrow \mathbb{Z}_\ell$, sets $f(z) = x + \sum_{m=1}^{t-1} a_m z^m$, and gives member j the share $x_j = f(j)$ [1]. For any set Q of t members, the Lagrange coefficients

$$\gamma_j^Q = \prod_{m \in Q,\, m \neq j} \frac{m}{m - j} \pmod{\ell} \tag{6.2}$$

satisfy $x = f(0) = \sum_{j \in Q} \gamma_j^Q x_j$, and because the map is linear the same coefficients work in the exponent of any base B: $xB = \sum_{j \in Q} \gamma_j^Q x_j B$. The member indices are distinct and nonzero modulo the prime ℓ, so every denominator is invertible.

Lemma 6.1 (Perfect secrecy). For any set S of $t - 1$ members and any two secrets x, x', the distributions of $(x_j)_{j \in S}$ under x and under x' are identical.

Proof. Fix the $t-1$ share values $(v_j)_{j\in S}$. Polynomial interpolation through the t points $\{(0,x)\}\cup\{(j,v_j)\}_{j\in S}$ gives exactly one polynomial of degree at most $t-1$, hence exactly one coefficient vector $(a_1,\dots,a_{t-1})$. The map from coefficients to the shares in S is therefore a bijection for each fixed secret, and with uniform coefficients the shares are uniform on $\mathbb{Z}_\ell^{t-1}$ whatever the secret. □

6.3 Verifiable secret sharing

Plain Shamir trusts the dealer. Members must be able to check their shares against public data, and in a distributed key generation every member is a dealer.

Feldman VSS

The dealer publishes $C_m = a_m G$ for $m = 0,\dots,t-1$ with $a_0 = x$ [2]. Member j accepts x_j if and only if

$$x_j G = \sum_{m=0}^{t-1} j^m C_m. \tag{6.3}$$

Anyone can compute every verification key $x_j G = \sum_m j^m C_m$ from the commitments alone. The cost is that $C_0 = xG$ is public, which is exactly what a public key should be, but it also means the dealer's commitments reveal $a_m G$ for every coefficient.

Pedersen VSS

The dealer samples a second polynomial $f'(z) = \sum_m a'_m z^m$, publishes $E_m = a_m G + a'_m H$, and sends $(x_j, x'_j) = (f(j), f'(j))$ [3]. Member j accepts if and only if

$$x_j G + x'_j H = \sum_{m=0}^{t-1} j^m E_m. \tag{6.4}$$

The generator H is derived by hashing to ristretto255 [10], so nobody knows $\log_G H$.

Lemma 6.2. Pedersen VSS commitments are perfectly hiding: $(E_m)_m$ is independent of f. Two different accepted share vectors for the same commitments, consistent with different secrets, yield $\log_G H$.

Proof. Each $E_m = a_m G + a'_m H$ with uniform a'_m is a uniform group element independent of a_m. For binding, suppose two openings of (6.4) for the same (E_m) interpolate to different $f(0)$; interpolating both openings at $z=0$ in the exponent gives $f(0)G + f'(0)H = \hat{f}(0)G + \hat{f}'(0)H$ with $f(0) \neq \hat{f}(0)$, so $\log_G H = (f(0)-\hat{f}(0))/(\hat{f}'(0)-f'(0))$. The denominator is nonzero because otherwise $f(0) = \hat{f}(0)$. □

6.4 Distributed key generation

The Council's key is generated without any party ever holding x. Running n parallel Feldman sharings and summing them (Joint-Feldman) is not enough: a rushing adversary who sees honest commitments before choosing its own can bias the resulting public key, which breaks the uniformity that security reductions for threshold ElGamal assume [4]. Privium uses the protocol of Gennaro, Jarecki, Krawczyk and Rabin, which first fixes the qualified set with hiding commitments and only then reveals the public key [4].

1. **Deal.** Each member i runs Pedersen VSS with a random secret z_i: it posts $(E_{i,m})_m$ to the Council account and sends $(f_i(j), f'_i(j))$ to each member j, encrypted with HPKE to j's registered key (Chapter 3) and posted on chain so delivery is publicly verifiable.
2. **Complain.** Member j checks (6.4) for every dealer and posts a complaint against each failure. A dealer answers a complaint by revealing that share pair in the clear; anyone checks it against (6.4). Dealers with an unanswered or failed complaint are removed. The remaining dealers form QUAL.
3. **Reveal.** Each $i \in$ QUAL posts Feldman commitments $C_{i,m} = a_{i,m} G$. Member j checks $f_i(j) G = \sum_m j^m C_{i,m}$; on failure it posts the share pair it received, which is valid under (6.4) and so proves the dealer's misbehavior, and the other members reconstruct z_i from their Pedersen shares to recompute $C_{i,0}$ in public.
4. **Output.** $s = \sum_{i\in\text{QUAL}} z_i$, $s_j = \sum_{i\in\text{QUAL}} f_i(j)$, and the Feldman keys $sG = \sum_{i\in\text{QUAL}} C_{i,0}$ and $s_j G = \sum_{i\in\text{QUAL}} \sum_m j^m C_{i,m}$.
5. **Twist.** The members run steps 1 to 4 again for a secret $\varkappa$ with Feldman commitments on base H, giving shares $\varkappa_j$ and $\varkappa_j H$, and once more for a sharing $o(z)$ of zero of degree $2t-2$ with commitments $o_j H$. Each member posts $w_j = s_j \varkappa_j + o_j$ with a Chaum-Pedersen proof that $\log_G(s_j G) = \log_{\varkappa_j H}(w_j H - o_j H)$. The w_j lie on a polynomial of degree $2t-2 \le n-1$, so

the n posted values interpolate to $u = s\varkappa$, and the program sets $P\mathcal{C} = u^{-1}(\varkappa H) = s^{-1}H$. Each member then posts $S_j = s_j P\mathcal{C}$ with a proof that $\log_G(s_j G) = \log_{P\mathcal{C}} S_j$.

Theorem 6.1 (Secure DKG [4]). If at most $t-1$ members are corrupt and $n \geq 2t-1$, steps 1 to 4 output shares of a secret s such that every honest member's share satisfies the Feldman relation for sG, s can be reconstructed from any t shares, sG is uniformly distributed in $\mathbb{G}$, and the adversary learns nothing about s beyond sG.

The twist adds only uniform values to that view. The masked product polynomial $s(z)\varkappa(z) + o(z)$ is uniform among polynomials of degree $2t-2$ with constant term u, and $u = s\varkappa$ is uniform and independent of s because $\varkappa$ is; so the transcript is simulatable from sG and $P\mathcal{C}$, and $P\mathcal{C} = s^{-1}H$ is uniform in $\mathbb{G}$. The twist runs once, at genesis, with all n members posting; every later change of membership preserves s and therefore $P\mathcal{C}$ (Section 6.6).

The Council account acts as the broadcast channel: every commitment, encrypted share and complaint is a Solana transaction, so the transcript is public and ordered, and step boundaries are slot deadlines rather than synchronized clocks.

6.5 Threshold decryption with proofs

Chaum-Pedersen proofs

The basic tool is a proof that two group elements share a discrete logarithm. For bases (P_1, P_2) and targets (T_1, T_2), the relation is $\{w : T_1 = wP_1 \wedge T_2 = wP_2\}$. The prover samples $\omega \leftarrow \mathbb{Z}_\ell$, sends $(R_1, R_2) = (\omega P_1, \omega P_2)$, receives ε and answers $z = \omega + \varepsilon w$; the verifier checks $zP_i = R_i + \varepsilon T_i$ for $i = 1, 2$ [6]. The protocol is complete, special-sound (two accepting answers z, z' for one commitment and distinct challenges give $w = (z - z')/(\varepsilon - \varepsilon')$), and honest-verifier zero-knowledge. Privium makes it non-interactive with Fiat-Shamir [7], hashing the whole statement into the challenge,

$$\varepsilon = \mathsf{H}\big(\mathsf{tag} \,\|\, P_1 \,\|\, P_2 \,\|\, T_1 \,\|\, T_2 \,\|\, R_1 \,\|\, R_2 \,\|\, \mathsf{ctx}\big), \tag{6.5}$$

where tag separates proof types and ctx binds the proof to its use (program ID, Council epoch, request). Omitting statement elements from the hash lets an adversary choose them after the challenge and forge proofs; the strong form in (6.5) is the one that yields a sound non-interactive proof in the random-oracle model [8].

Threshold decryption. A member's decryption share for a handle D is $s_j D$ with a Chaum-Pedersen proof on bases $(P\mathcal{C}, D)$ and targets $(S_j, s_j D)$. Any t valid shares combine to $sD = \sum_{j \in Q} \gamma_j^Q \, s_j D$ and the plaintext point is $C - sD$, as in (6.1). This is the scheme analysed by Shoup and Gennaro, whose proofs of correct decryption are what make the combination robust: an invalid share is rejected individually, so t honest members always suffice regardless of what corrupt members post [9].

Re-encryption to a verified key

Privium never needs the plaintext point in public. It needs the handle D moved from $P\mathcal{C}$ to a recipient key Y (an enclave's Y_E or an owner's Y_u). Member j samples $\kappa_j \leftarrow \mathbb{Z}_\ell$ and posts

$$A_j = \kappa_j G, \qquad B_j = s_j D + \kappa_j Y, \tag{6.6}$$

with a proof Π_j of knowledge of (s_j, κ_j) such that $S_j = s_j P\mathcal{C}$, $A_j = \kappa_j G$ and $B_j = s_j D + \kappa_j Y$. Π_j is the two-witness sigma protocol with commitments $(R_1, R_2, R_3) = (\omega_1 P\mathcal{C}, \omega_2 G, \omega_1 D + \omega_2 Y)$, responses $z_1 = \omega_1 + \varepsilon s_j$, $z_2 = \omega_2 + \varepsilon \kappa_j$, challenge as in (6.5) over the full statement $(P\mathcal{C}, G, D, Y, S_j, A_j, B_j, R_1, R_2, R_3, \mathsf{ctx})$, and checks

$$z_1 P\mathcal{C} = R_1 + \varepsilon S_j, \qquad z_2 G = R_2 + \varepsilon A_j, \qquad z_1 D + z_2 Y = R_3 + \varepsilon B_j. \tag{6.7}$$

Combining any t valid shares,

$$A = \sum_{j \in Q} \gamma_j^Q A_j = \kappa G, \qquad B = \sum_{j \in Q} \gamma_j^Q B_j = sD + \kappa Y, \qquad \kappa = \sum_{j \in Q} \gamma_j^Q \kappa_j. \tag{6.8}$$

Lemma 6.3 (Re-encryption correctness). For a leaf $(C, D) = (bG + rH,\ rP\mathcal{C})$ and $Y = yG$, the pair $(A, C - B)$ is an ElGamal ciphertext of bG under Y: $(C - B) + yA = bG$.

Proof. $C - B = bG + rH - srP\mathcal{C} - \kappa Y = bG + rH - rH - \kappa y G = bG - \kappa y G$, and $yA = y\kappa G$. $\square$

Lemma 6.4 (Share soundness). If (6.7) holds for two challenges $\varepsilon \neq \varepsilon'$ on the same commitments, then $s_j = (z_1 - z_1')/(\varepsilon - \varepsilon')$ satisfies $S_j = s_j P\mathcal{C}$ and $\kappa_j = (z_2 - z_2')/(\varepsilon - \varepsilon')$ satisfies $A_j = \kappa_j G$ and $B_j = s_j D + \kappa_j Y$. Hence, in the random-oracle model, a member who posts an accepted B_j different from $s_j D + \kappa_j Y$ for its registered S_j succeeds with negligible probability.

Proof. Subtracting the two instances of each equation in (6.7) gives $(z_1 - z_1')P\mathcal{C} = (\varepsilon - \varepsilon')S_j$, $(z_2 - z_2')G = (\varepsilon - \varepsilon')A_j$ and $(z_1 - z_1')D + (z_2 - z_2')Y = (\varepsilon - \varepsilon')B_j$. Divide by $\varepsilon - \varepsilon'$. The forking lemma turns a prover that convinces with non-negligible probability into one that produces such a pair. □

Lemma 6.5 (Re-encryption privacy). Under DDH in $\mathbb{G}$, the shares $(A_j, B_j)_j$ of any set of members, together with any $t-1$ secret shares, reveal nothing about bG to a party that does not know y.

Proof sketch. Each $B_j - s_j D = \kappa_j Y$ with $A_j = \kappa_j G$; the pair $(A_j, \kappa_j Y)$ is a Diffie-Hellman tuple over (G, Y), indistinguishable from (A_j, U_j) with U_j uniform. Replacing every mask by an independent uniform element (one hybrid per share) makes each B_j uniform and independent of $s_j D$. What remains is (C, D) and $t-1$ shares of s, which by Lemma 6.1 and Theorem 3.2 reveal nothing about bG. □

The listing below is the check that the Pool program and every recipient run on each share, followed by the combination (6.8).

```rust
use curve25519_dalek::{constants::RISTRETTO_BASEPOINT_POINT as G, ristretto::RistrettoPoint as P, Scalar};

pub struct ReEncShare { pub member: u8, pub a: P, pub b: P, pub r: [P; 3], pub z1: Scalar, pub z2: Scalar }

/// Checks (6.7) for one share: `pc` is P_C, `sj` the member's S_j, `d` the handle, `y` the recipient key.
pub fn verify_share(s: &ReEncShare, pc: &P, sj: &P, d: &P, y: &P, ctx: &[u8]) -> bool {
    let eps = challenge(b"privium/reenc/v1", &[pc, &G, d, y, sj, &s.a, &s.b, &s.r[0], &s.r[1], &s.r[2]], ctx);
    s.z1 * pc == s.r[0] + eps * sj
        && s.z2 * G == s.r[1] + eps * s.a
        && s.z1 * d + s.z2 * y == s.r[2] + eps * s.b
}

/// Lagrange coefficient (6.2) of `j` in the quorum `q`.
fn gamma(j: u8, q: &[u8]) -> Scalar {
    let (mut num, mut den) = (Scalar::ONE, Scalar::ONE);
    for &m in q.iter().filter(|&&m| m != j) {
        num *= Scalar::from(m as u64);
        den *= Scalar::from(m as u64) - Scalar::from(j as u64);
    }
    num * den.invert()
}

/// Combines t verified shares into (A, B) as in (6.8).
pub fn combine(shares: &[ReEncShare]) -> (P, P) {
    let q: Vec<u8> = shares.iter().map(|s| s.member).collect();
    shares.iter().fold((P::identity(), P::identity()), |(a, b), s| {
        let g = gamma(s.member, &q);
        (a + g * s.a, b + g * s.b)
    })
}
```

On chain the same equations run through Solana's curve25519 syscalls for ristretto255 group operations and multiscalar multiplication [12]; each share verification is a fixed number of group operations, and Chapter 9 accounts for its compute budget.

6.6 Council membership and proactive resharing

Members change, and shares held for long periods accumulate exposure. Privium reshares the same secret s to a new committee (n', t') at every membership change and at least once every E_C slots, recommended $E_C = 10\,E_{\text{key}}$ (parameter): an adversary who corrupts members over time must collect t shares within one period, because shares from different periods do not combine [5].

Protocol. Let Q be a set of t current members. Each $i \in Q$ acts as a Feldman dealer of its own share on base $P\mathcal{C}$: it samples $g_i(z) = s_i + \sum_{m=1}^{t'-1} a_{i,m} z^m$, posts $C_{i,m} = a_{i,m} P\mathcal{C}$ with $C_{i,0} = s_i P\mathcal{C}$, and sends $g_i(j)$ to each new member j, encrypted as in Section 6.4. The program and every new member check $C_{i,0} = S_i$, the dealer's registered verification key, which forces each dealer to reshare its real share. New member j verifies its sub-shares by (6.3) with $P\mathcal{C}$ in place of G and sets

$$s'_j = \sum_{i \in Q} \gamma_i^Q \, g_i(j), \qquad S'_j = \sum_{i \in Q} \gamma_i^Q \sum_{m=0}^{t'-1} j^m C_{i,m}. \tag{6.9}$$

Old members erase their shares once the new committee's verification keys are recorded.

Lemma 6.6 (Resharing preserves the key). The new shares are a t'-of-n' Shamir sharing of the same s, so $P\mathcal{C}$ is unchanged and the new verification keys interpolate to $sP\mathcal{C} = H$.

Proof. Let $g(z) = \sum_{i \in Q} \gamma_i^{Q} g_i(z)$. It has degree at most $t' - 1$, $s'_j = g(j)$, and $g(0) = \sum_{i \in Q} \gamma_i^{Q} s_i = s$ by (6.2). In the exponent, $\sum_{i \in Q} \gamma_i^{Q} C_{i,0} = \sum_{i \in Q} \gamma_i^{Q} S_i = s P_{\mathcal{C}} = H$. □

Because $P_{\mathcal{C}}$ is unchanged, nothing encrypted under it needs to be touched: every leaf's Council handle and $\mathbf{wrap}(kL)$ remain valid across any number of membership changes. A dealer whose sub-shares fail (6.3) is excluded and the protocol reruns with a different Q; since t honest old members exist, a valid Q always exists. The resharing transcript lives in the Council account, and the program switches to the new verification keys only after t' new members have posted acknowledgements of valid sub-shares.

6.7 The Escape

The Escape is the path by which every owner leaves with their last committed balance when the Coprocessor stops committing, using only the chain, the Council and the owner's own key.

State at trigger

Let s_{last} be the slot of the last accepted Commit, with root $\rho^{(k^\star)}$. The Coprocessor commits each Crossing that settles something and, with nothing to settle, sends a heartbeat Commit 750 slots after the last (Chapter 10), so a live enclave keeps the gap to 750 slots plus the heartbeat's own landing, and the gap measures enclave liveness rather than trading activity. The Pool holds two public queues: deposits received but not yet credited by a Commit, and Exits listed in a committed exits root but not yet released. It may also hold the recorded fills of the open Crossing $k^\star + 1$: Residual legs execute in their own transactions before the Commit that absorbs them (Chapter 9), each leg ℓ with input asset $\text{in}(\ell)$, output asset $\text{out}(\ell)$ and measured amounts $(\text{spent}_\ell, \text{got}_\ell)$, and the program caps the legs of one Crossing so that $S_a = \sum_{\ell : \text{in}(\ell) = a} \text{spent}_\ell \le T_a$, the committed total of the input asset.

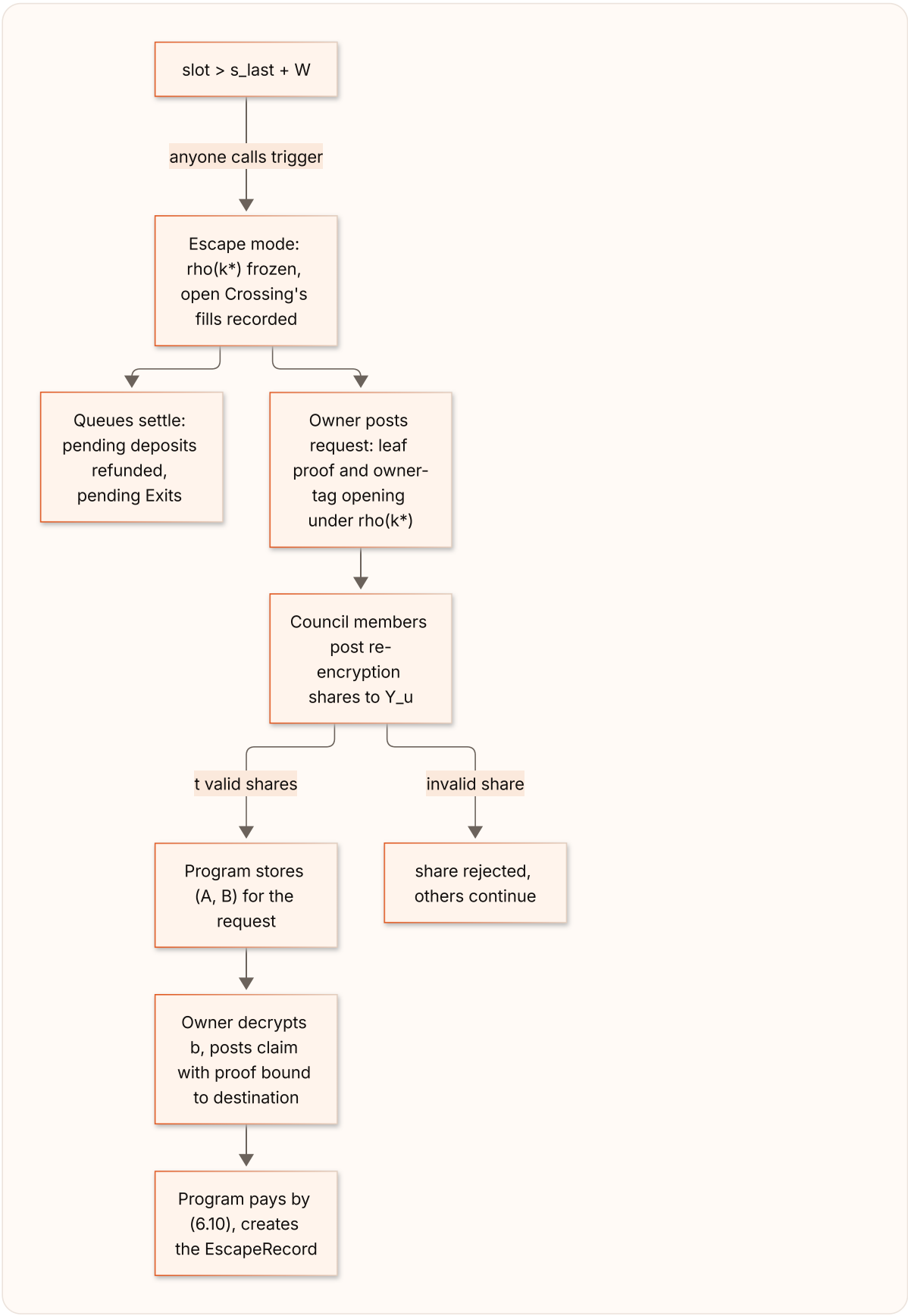
The Escape settles those legs over the committed holders of each input asset, in proportion to committed balance. With T_a the frozen total of asset a, a claim of b units of a pays

$$\left\lfloor b \, \frac{T_a - S_a}{T_a} \right\rfloor \text{ units of } a, \qquad \left\lfloor b \, \frac{\text{got}_\ell}{T_a} \right\rfloor \text{ units of } \text{out}(\ell) \ \text{ for each leg } \ell \text{ with } \text{in}(\ell) = a. \tag{6.10}$$

When no Crossing is open, or none of its legs spent a, (6.10) pays b units of a. Each leg's rate $\text{got}_\ell / \text{spent}_\ell$ is its executed price, which the enclave's signed bound keeps within $\beta_\pi^{(k)}$ of the oracle mid (Chapter 7), so every claimant receives their committed balance in value at that price.

Parameter. $W = 9{,}000$ slots (parameter): it spans twelve heartbeat intervals and exceeds by a wide margin the few transactions a Coprocessor failover needs (attestation in Section 5.5, key release in Section 6.5, first Commit) and the lease of Chapter 10, so neither an idle Pool nor a routine failover triggers it, while still bounding how long a halted Ledger holds funds. Chapter 9 gives the slot time that converts it to minutes.

Steps



1. **Trigger.** When the current slot exceeds $s_{\text{last}} + W$, any account may call `trigger_escape`. The program sets the Pool to Escape mode, freezes $\rho^{(k^\star)}$ and the totals T_a, copies the open Crossing's recorded fills into `EscapeState`, and rejects every later Commit, open, route and deposit. Intents not yet in a committed root carry no tokens and simply lapse.
2. **Queues.** Anyone may crank the queues: each pending deposit is refunded to the source token account its queue entry records, and each pending Exit is paid to its recorded destination.

3. **Request.** Owner u posts, for asset a, the leaf index, the leaf's pair (C, D), the opening (Y_u, ς_u) of its owner tag $\psi_{u,a}$ (Definition 4.2) and a Merkle path to $\rho^{(k^\star)}$; the AEAD layers are passed as their hash, so the request reveals nothing beyond what is already public except the escape key. The program recomputes $\psi_{u,a}$ from the opening, the leaf's asset and its version, verifies the path, and opens an `EscapeRequest` account.
4. **Respond.** Council members post shares (6.6) toward Y_u for the handle D, with proofs Π_j and $\mathsf{ctx} = (\mathrm{id}_{\mathrm{prog}}, \text{request address})$. The program verifies each with (6.7) against the registered S_j and, after t valid shares, stores (A, B) by (6.8). Members may batch shares for many requests in one transaction.
5. **Claim.** The owner computes $bG = (C - B) + y_u A$ by Lemma 6.3. The owner takes the candidate b from their own leaf, decrypted with $k_{u,a}$ (Chapter 4), or from the SDK's record, and checks it against bG. They post b, a destination d (a fresh address, or a Ring deposit as in Chapter 8), and a Chaum-Pedersen proof that $\log_G Y_u = \log_A Z$ with $Z = bG - (C - B)$, whose context includes d, a and the request address.
6. **Pay.** The program recomputes Z, verifies the proof, creates the `EscapeRecord` for the leaf and asset, and transfers the amounts of (6.10) from the vaults to d's token accounts.

```
pub fn escape_claim(ctx: Context<EscapeClaim>, amount: u64, proof: DleqProof) → Result<()> {
    require!(ctx.accounts.ledger.load()?.escape_from_slot ≠ 0, PrivErr::NotEscape);
    let req = &mut ctx.accounts.request;
    require!(req.combined && !req.claimed, PrivErr::RequestState);
    // `EscapeRecord` is created with `init` for (leaf, mint): a second claim fails there.

    // Z = bG - (C - B); the proof shows log_G(Y_u) = log_A(Z).
    let (c, a, big_b) = (req.commitment.decompress(), req.a.decompress(), req.b.decompress());
    let z = Scalar::from(amount) * G - (c - big_b);
    let dest = ctx.accounts.destination.key();
    let claim_ctx = claim_context(&crate::ID, &req.key(), &req.mint, &dest);
    require!(
        proof.verify(&G, &req.owner_key.decompress(), &a, &z, &claim_ctx),
        PrivErr::ClaimProof
    );
    req.claimed = true;

    // Settlement rule (6.10): the leaf's asset, plus output assets of open legs that spent it.
    let total = ctx.accounts.asset.load()?.ledger_total;
    for (mint, units) in ctx.accounts.escape.load()?.settle(&req.mint, amount, total)? {
        transfer_from_vault(&ctx, &mint, &dest, units)?;
    }
    emit!(Escaped { request: req.key(), mint: req.mint, amount });
    Ok(())
}
```

`G` is the ristretto255 base point; `decompress` rejects non-canonical encodings, so every point the program uses is a valid group element. The request account holds (C, D), Y_u and the leaf index exactly as verified against $\rho^{(k^\star)}$ in step 3, and the claim context binds the proof to this request, mint and destination, so the proof cannot be lifted onto a different payout or redirected by whoever submits the transaction.

6.8 Safety and liveness of the Escape

For each asset a, write $T_a = \sum_u b_{u,a}^{(k^\star)}$ for the frozen committed total, P_a^{dep} for the pending deposit queue, and P_a^{exit} for the pending Exit queue at trigger. Fees are not among them: each Commit paid its Crossing's fees out of the vaults to the fee recipient. By the conservation check at committed roots (Equation 9.2 and Theorem 4.1) and the recorded fills of the open Crossing, the vault at trigger holds

$$V_a = T_a + P_a^{\mathrm{exit}} + P_a^{\mathrm{dep}} - S_a + \sum_{\ell : \mathrm{out}(\ell) = a} \mathrm{got}_\ell. \tag{6.11}$$

Theorem 6.2 (Escape safety). Assume Commit authenticity (Theorem 5.2), collision resistance of H, the discrete-logarithm assumption in $\mathbb{G}$, and the random-oracle model for (6.5). Then, against any set of corrupt Council members (including all n) and any number of corrupt relayers, except with negligible probability in λ:

1. *(Exactness)* every Escape claim for entry (u, a) is for the leaf's Council decryption $\beta_{u,a}$, defined by $\beta_{u,a} G = C - sD$, which equals $b_{u,a}^{(k^\star)}$, and is paid by (6.10);
2. *(Exclusivity)* every Escape payout for (u, a) is to a destination chosen by the holder of y_u;
3. *(Uniqueness)* each entry (u, a) is paid at most once;
4. *(Solvency)* under the hypotheses of Theorem 4.2, the total paid in asset a during the Escape is at most V_a, so every claim, refund and queued Exit is payable.

Proof. (1) The request's (C, D) and owner-tag opening are bound to $\rho^{(k^\star)}$ by the Merkle path, so substituting other values requires a collision in H. Each share in (A, B) passed (6.7) against a registered S_j, so by Lemma 6.4 each satisfies $B_j = s_j D + \kappa_j Y_u$ with $A_j = \kappa_j G$ for the true share s_j; hence (6.8) gives $B = sD + \kappa Y_u$ and $A = \kappa G$ for some κ, even if every member is corrupt, since corrupt members can choose κ_j but not s_j. Then $Z = bG - (C - B) = (b - \beta_{u,a})G + \kappa Y_u$. An accepted claim proof yields, by special soundness, y with $Y_u = yG$ and $Z = yA = \kappa Y_u$, so $(b - \beta_{u,a})G = O$ and $b = \beta_{u,a}$ because G has prime order ℓ and balances are below ℓ. By Theorem 5.2, $\rho^{(k^\star)}$ was signed by an admitted enclave, which writes $D = rP_{\mathcal{C}}$ for the opening r of $C = b_{u,a}^{(k^\star)} G + rH$, so $\beta_{u,a} = b_{u,a}^{(k^\star)}$; Theorem 4.2 gives the same equality from the published proofs alone.

(2) The extractor from (1) outputs $y_u = \log_G Y_u$. A claimant who does not hold y_u and produces an accepted proof therefore solves a discrete logarithm. A relayer who copies an honest owner's proof cannot change d, because d is in the hashed context and a changed context gives an independent challenge.

(3) The `EscapeRecord` for (leaf, a) is created in the same transaction as the payout and its existence aborts any later claim, and a leaf index determines a unique position in $\rho^{(k^\star)}$.

(4) By (1), (3) and Corollary 4.1, claims in asset a are for decryptions that sum to at most T_a, so by (6.10) they pay at most $T_a - S_a$ units of a, and claims in the input assets of legs with $\mathrm{out}(\ell) = a$ pay at most $\sum_{\ell:\mathrm{out}(\ell)=a} \mathrm{got}_\ell$ units of a, since the floors only round down. Refunds pay exactly P_a^{dep} and queued Exits exactly P_a^{exit}, each entry once. The sum is at most V_a by (6.11), and in Escape mode no Commit, Residual or new Exit moves the vault. □

The theorem holds with every Council member corrupt: a corrupt Council can withhold shares, but it cannot direct a payout, inflate one, or pay twice. Custody therefore rests on the chain, the Merkle root, the published proofs and the owner's key; the Council contributes availability.

Theorem 6.3 (Escape liveness). If at least t Council members are honest and each lands its share transactions on chain, then every owner u who posts a valid request for (u, a) receives $b_{u,a}^{(k^\star)}$, settled by (6.10), at a destination of their choice.

Proof. Honest members respond to every valid request by (6.6), and by completeness of the sigma protocol their shares pass (6.7). Shares that fail verification are discarded individually, so t honest shares suffice regardless of other posts, and (6.8) produces (A, B) with $B = sD + \kappa Y_u$. By Lemma 6.3 the owner computes $b_{u,a}^{(k^\star)} G$, matches it against their leaf or balance record, and produces an honest claim proof that passes verification. The vaults hold enough by Theorem 6.2 (4), and no `EscapeRecord` exists for the entry, because only a claim by the holder of y_u can create it. □

Confidentiality during the Escape. By Lemma 6.5, the shares posted for owner u reveal nothing about $b_{u,a}^{(k^\star)}$ to anyone without y_u, and requests reveal only leaf fields that were already committed in ciphertext, together with the escape key. Owners who never claim keep their balances confidential at the level of the honest threshold of the Council. A claimed amount becomes visible on chain as a vault transfer, exactly as a normal Exit does, and the destination may be a Ring deposit to unlink it (Chapter 8).

The Escape gives the Council a narrow role: it holds a key no fewer than t members can use, it re-encrypts only toward keys the chain has verified, and every share it posts comes with a proof the program checks. The Coprocessor of Chapter 5 provides confidentiality and authentic Commits; the Council and the Escape guarantee that a trader's last committed balance remains theirs whether or not the Coprocessor ever commits again.

References

1. A. Shamir, "How to share a secret", Communications of the ACM 22(11), November 1979. https://doi.org/10.1145/359168.359176
2. P. Feldman, "A practical scheme for non-interactive verifiable secret sharing", 28th IEEE Symposium on Foundations of Computer Science (FOCS), 1987. https://doi.org/10.1109/SFCS.1987.4
3. T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing", CRYPTO 1991, LNCS 576. https://doi.org/10.1007/3-540-46766-1_9
4. R. Gennaro, S. Jarecki, H. Krawczyk, T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems", Journal of Cryptology 20(1), 2007. https://doi.org/10.1007/s00145-006-0347-3
5. A. Herzberg, S. Jarecki, H. Krawczyk, M. Yung, "Proactive secret sharing or: how to cope with perpetual leakage", CRYPTO 1995, LNCS 963. https://doi.org/10.1007/3-540-44750-4_27
6. D. Chaum, T. P. Pedersen, "Wallet databases with observers", CRYPTO 1992, LNCS 740. https://doi.org/10.1007/3-540-48071-4_7
7. A. Fiat, A. Shamir, "How to prove yourself: practical solutions to identification and signature problems", CRYPTO 1986, LNCS 263. https://doi.org/10.1007/3-540-47721-7_12
8. D. Bernhard, O. Pereira, B. Warinschi, "How not to prove yourself: pitfalls of the Fiat-Shamir heuristic and applications to Helios", ASIACRYPT 2012, LNCS 7658. https://doi.org/10.1007/978-3-642-34961-4_38

9. V. Shoup, R. Gennaro, “Securing threshold cryptosystems against chosen ciphertext attack”, Journal of Cryptology 15(2), 2002. https://doi.org/10.1007/s00145-001-0020-9
10. H. de Valence, J. Grigg, M. Hamburg, I. Lovecruft, G. Tankersley, F. Valsorda, “The ristretto255 and decaf448 Groups”, RFC 9496, IETF, December 2023. https://www.rfc-editor.org/rfc/rfc9496
11. N. Lokhande, R. Gupta, R. Chahar, A. Jangra, A. Prajapati, M. Kumari, “Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana”, IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
12. `solana-curve25519` crate documentation (ristretto255 and edwards25519 syscall wrappers), Anza, 2025. https://docs.rs/solana-curve25519/latest/solana_curve25519/

Chapter 7

The Crossing engine

The Crossing is the unit of trading in Privium. Whenever it holds something to settle, as often as once every Δ slots, the Coprocessor takes the sealed Intents it holds, nets them pair by pair at the oracle mid, fills opposing size inside the Pool, and sends only what is left, the Residual, to public liquidity as one swap per pair. This chapter defines that procedure precisely and proves what it guarantees: netting is maximal, allocation within a batch is independent of arrival time and immune to splitting, every fill respects its limit, pairs settle independently and in any order, and the value a searcher can extract from a Crossing is bounded by the Residual alone. It closes with an expected-cost inequality that tells a trader when a Crossing beats a direct swap, and a worked example. The cryptography that seals Intents is in Chapter 3, the Ledger the Crossing writes into is in Chapter 4, and the on-chain instructions that execute the Residual and the Commit are in Chapter 9.

The batch model

Time on Solana is measured in slots. Let $s^{(k-1)}$ be the slot in which Commit $k-1$ landed. Crossing k closes at its cutoff slot $t^{(k)}$, the slot at which the Coprocessor begins it, never sooner than one batch interval after that Commit:

$$t^{(k)} \geq s^{(k-1)} + \Delta.$$

The Coprocessor begins a Crossing only when it has something to settle; Chapter 10 lists what counts. While work waits, Crossing k begins as soon as the bound allows, $t^{(k)} = s^{(k-1)} + \Delta$. After an idle stretch, $t^{(k)}$ is the slot at which the Coprocessor finds work, or 750 slots after $s^{(k-1)}$ for a heartbeat, a Crossing that settles nothing and keeps the Ledger anchored (Chapter 10). Crossing k owns the half-open slot interval

$$\mathcal{T}^{(k)} = [\,t^{(k-1)},\ t^{(k)}\,).$$

An Intent that the Coprocessor has received and decrypted before $t^{(k)}$ is a candidate for Crossing k; one that arrives later waits for Crossing $k+1$, which begins as soon as Δ slots have passed since Commit k. The batch interval is a design parameter: $\Delta = 4$ slots (parameter), one leader span, since Solana assigns each leader four consecutive slots and keeps that span fixed across the reduction of slot duration [8]; Chapter 9 gives the slot time in force. Consecutive cutoffs therefore lie at least one leader span apart, and a Crossing produces at most one Residual per pair per leader.

The batch model is the frequent batch auction of Budish, Cramton and Shim [1], who show that in continuous limit order books a public signal creates a race in which the fastest participant picks off stale quotes, and that discretizing time into short uniform-price batches turns that race into competition on price. Privium adopts the same discretization and goes one step further on information: inside the interval every Intent is a ciphertext to everyone except the enclave (Chapter 5), so no participant, fast or slow, can condition on another's order.

Definition 7.1 (Eligible set). For pair $\pi = (a, q)$ and Crossing k, an Intent $i = (u, \pi, s, x, \ell, \tau, \nu)$ is *eligible* if (i) it was decrypted before $t^{(k)}$, (ii) $\tau \geq t^{(k)}$, (iii) its nonce ν has not been consumed by an earlier Crossing (Chapter 4 fixes the nonce discipline), and (iv) it is *funded*: the trader's committed balance covers the worst case of the order,

$$s = -1 \Rightarrow x \leq \hat{b}_{u,a}, \qquad s = +1 \Rightarrow x \cdot \ell \cdot (1+\phi) \leq \hat{b}_{u,q}, \tag{7.1}$$

where $\hat{b}$ is the trader's *unreserved* balance defined in the section on multi-pair consistency. The eligible set is $\mathcal{E}_\pi^{(k)}$. A market order is an Intent whose limit is the SDK's protective bound rather than an unbounded value, so (7.1) is always finite.

Clearing at the oracle mid

Each pair clears at a single price taken from the oracle, not discovered from the batch:

$$p_\pi^{(k)} = m_\pi^{(k)}, \tag{7.2}$$

where $m_\pi^{(k)}$ is the Pyth aggregate price of a in q as `open_crossing` reads it after the cutoff slot $t^{(k)}$, from the price accounts Pyth keeps current on Solana (Chapter 10). Pyth publishes a confidence interval $c_\pi^{(k)}$ with every price and recommends that integrators pause price-dependent activity when c/m exceeds a threshold and reject stale updates [7]. Privium applies both rules: a pair crosses in Crossing k only if its prices were published within the maximum price age that Chapter 10 sets for `open_crossing` and

$$c_\pi^{(k)} / m_\pi^{(k)} \leq \kappa, \tag{7.3}$$

with $\kappa = 25$ bps (parameter): wide enough to admit normal markets in liquid pairs, tight enough that a cross at the mid is never further from a fair price than the interval Pyth itself publishes. The enclave holds the mid to the same width against a live market price. With $P_\pi^{(k)}$ the ratio of Jupiter's current USD prices for a and q, read through the host after the cutoff (Chapter 10), the pair crosses only if

$$\left| m_\pi^{(k)} - P_\pi^{(k)} \right| \le \kappa \, P_\pi^{(k)}.$$

The reference decides only whether the pair trades; a pair that passes still clears at (7.2). A pair that fails the age bound, (7.3) or this band does not trade in Crossing k: none of it crosses or routes, and its eligible Intents rest until the next Crossing.

Why the mid and not a price discovered from the batch? A uniform-price auction over the batch's own orders would let the batch's composition set the price, and a thin batch would clear far from the market. The mid is the one price at which neither side of a cross is favored: each side saves exactly the half-spread it would have paid to a market maker. Brokers that internalize client flow fill it at or inside the prevailing quote, and Easley, Kiefer and O'Hara show that flow internalized this way carries different information from flow sent to the exchange [5]; fixing the price at the mid gives both sides of an internal cross the same improvement. It also leaves the enclave no discretion. The clearing price is a public input, read from a public feed at a public slot, so any observer can recompute it.

Definition 7.2 (Active sets). With $p = p_\pi^{(k)}$, the active buyers and sellers are

$$\mathcal{B} = \{ i \in \mathcal{E}_\pi^{(k)} : s_i = +1,\ \ell_i \ge p \}, \qquad \mathcal{S} = \{ i \in \mathcal{E}_\pi^{(k)} : s_i = -1,\ \ell_i \le p \}.$$

Write $Q^+ = \sum_{i \in \mathcal{B}} x_i$ and $Q^- = \sum_{i \in \mathcal{S}} x_i$ for active buy and sell size. The crossed volume, the Residual and the active gross volume are

$$X_\pi = \min(Q^+, Q^-), \qquad R_\pi^{(k)} = Q^+ - Q^-, \qquad G_\pi = Q^+ + Q^-. \tag{7.4}$$

The *heavy side* is the side with the larger total ($\mathcal{B}$ if $R > 0$, $\mathcal{S}$ if $R < 0$); the other is the *light side*. The *crossing rate* is $\gamma_\pi = 2X_\pi / G_\pi \in [0, 1]$, the fraction of active gross size that never reaches the market. The identity $G_\pi = 2X_\pi + |R_\pi^{(k)}|$ follows from (7.4) and is used repeatedly below.

Theorem 7.1 (Maximal netting). Let a feasible internal allocation be a vector (f_i) with $0 \le f_i \le x_i$ for active Intents, $f_i = 0$ for inactive ones, and base conservation $\sum_{i \in \mathcal{B}} f_i = \sum_{i \in \mathcal{S}} f_i$. Then the maximal crossed volume over all feasible allocations is X_π, and every maximal allocation fills the light side in full.

Proof. For any feasible allocation, $\sum_{\mathcal{B}} f_i \le Q^+$ and $\sum_{\mathcal{S}} f_i \le Q^-$, and the two sums are equal, so the crossed volume is at most $\min(Q^+, Q^-) = X_\pi$. The bound is attained: fill the light side in full, which uses exactly X_π, and distribute X_π over the heavy side, which is possible since the heavy side's total is at least X_π. If an allocation attains X_π and some light-side Intent j has $f_j < x_j$, then the light side's sum is strictly below its total, which is X_π, a contradiction. □

Allocation within a batch

Theorem 7.1 fixes the light side completely. The heavy side holds claims $(x_i)_{i \in \mathcal{H}}$ with total $Q_{\mathcal{H}} = \max(Q^+, Q^-)$, against a crossed volume $X_\pi \le Q_{\mathcal{H}}$ that must be divided among them. This is a claims problem in the sense of the axiomatic rationing literature [9], and two rules are natural: *time priority*, which fills claims in order of arrival until X_π is exhausted, and *pro rata*, which fills every claim in proportion

$$f_i = X_\pi \cdot \frac{x_i}{Q_{\mathcal{H}}}, \qquad i \in \mathcal{H}. \tag{7.5}$$

Privium allocates pro rata. Two properties decide it.

Theorem 7.2 (Arrival-time invariance). Under pro rata (7.5), the fill of an Intent in Crossing k is a function of the eligible set alone and does not depend on the order or slot at which Intents arrived within $\mathcal{T}^{(k)}$. Under time priority, whenever $X_\pi < Q_{\mathcal{H}}$ there exist two heavy-side Intents for which arriving earlier strictly increases fill.

Proof. The right side of (7.5) depends only on x_i, X_π and $Q_{\mathcal{H}}$, which are functions of the eligible set, so permuting arrivals within $\mathcal{T}^{(k)}$ leaves every f_i unchanged. For time priority, order heavy-side Intents by arrival and let j be the first whose cumulative size exceeds X_π; such a j exists since $X_\pi < Q_{\mathcal{H}}$. If j is the first Intent, it is partially filled while any later Intent receives nothing, and moving that later Intent ahead of j gives it a positive fill. Otherwise the Intent immediately before j is filled in full and j is not, and swapping the two strictly increases j's fill. □

Theorem 7.2 is the in-batch form of the argument in [1]: time priority inside a batch rebuilds the speed race the batch was introduced to remove. Under pro rata there is nothing to gain from latency inside $\mathcal{T}^{(k)}$, and the only timing that matters is whether an Intent makes the cutoff at all.

Theorem 7.3 (Split and merge neutrality). Under pro rata, if a trader replaces an Intent of size x by Intents of sizes $x^{(1)}, \dots, x^{(r)}$ with $\sum_j x^{(j)} = x$ and identical other fields, the total fill of the pieces equals the fill of the original. Conversely, merging Intents does not change their total fill.

Proof. The heavy-side total $Q_{\mathcal{H}}$ and the crossed volume X_π are unchanged by the replacement, since both are sums over sizes. The pieces receive $\sum_j X_\pi x^{(j)} / Q_{\mathcal{H}} = X_\pi x / Q_{\mathcal{H}}$, the fill of the original. Every other Intent's fill is unchanged for the same reason. Merging is the same identity read backwards. □

The converse also holds in the continuous model: the proportional rule is the only rule on claims problems that is immune to both merging and splitting of claims [9], so pro rata is not one fair choice among several but the unique rule with this property.

Integer fills

Sizes are integers in base units, and (7.5) is generally fractional. Privium rounds by the largest-remainder method. Let $f_i^\circ = \lfloor X_\pi x_i / Q_{\mathcal{H}} \rfloor$, let $r_i = X_\pi x_i \bmod Q_{\mathcal{H}}$, and let $L = X_\pi - \sum_{\mathcal{H}} f_i^\circ$. The L Intents with the largest r_i receive one more base unit. Ties are broken by $\mathsf{H}(\sigma^{(k)} \,\|\, \nu_i)$, where $\sigma^{(k)}$ is a 32-byte seed the enclave draws at the cutoff and includes in the Commit.

Lemma 7.4 (Rounding). The rounded allocation satisfies $\sum_{\mathcal{H}} f_i = X_\pi$ exactly, $|f_i - X_\pi x_i / Q_{\mathcal{H}}| < 1$ for every i, and no trader can bias the tie-break by choice of nonce.

Proof. Since $\sum_{\mathcal{H}} X_\pi x_i = X_\pi Q_{\mathcal{H}}$, we have $\sum_{\mathcal{H}} r_i = Q_{\mathcal{H}} \left(X_\pi - \sum f_i^\circ \right) = Q_{\mathcal{H}} L$. Each $r_i < Q_{\mathcal{H}}$, so L is at most the number of Intents with $r_i > 0$, and adding one unit to L of them yields the exact total. Each fill is either the floor or the floor plus one of a value whose fractional part is positive whenever the unit is added, which gives the bound. The seed $\sigma^{(k)}$ is drawn after every eligible nonce is fixed, so the tie-break order is independent of each nonce from the trader's view, and its effect is at most one base unit. □

The listing below is the Coprocessor's per-pair kernel. It takes no clock and no arrival order, which is Theorem 7.2 in code; route eligibility is applied afterwards.

```rust
pub struct OpenIntent { pub id: IntentId, pub side: Side, pub size: u64, pub limit: Price, pub nonce: [u8; 16] }
pub struct PairCross { pub fills: Vec<(IntentId, u64)>, pub crossed: u64, pub residual: i128 }

pub fn cross_pair(eligible: &[OpenIntent], mid: Price, seed: &[u8; 32]) -> PairCross {
    let active = |i: &&OpenIntent| match i.side {
        Side::Buy => i.limit >= mid,
        Side::Sell => i.limit <= mid,
    };
    let (buys, sells): (Vec<&OpenIntent>, Vec<&OpenIntent>) =
        eligible.iter().filter(active).partition(|i| i.side == Side::Buy);
    let q_buy: u128 = buys.iter().map(|i| i.size as u128).sum();
    let q_sell: u128 = sells.iter().map(|i| i.size as u128).sum();
    let crossed = q_buy.min(q_sell);
    let (light, heavy, q_heavy) =
        if q_buy >= q_sell { (sells, buys, q_buy) } else { (buys, sells, q_sell) };

    // Light side fills in full (Theorem 7.1).
    let mut fills: Vec<(IntentId, u64)> = light.iter().map(|i| (i.id, i.size)).collect();
    if q_heavy == 0 {
        return PairCross { fills, crossed: 0, residual: 0 };
    }

    // Heavy side: pro rata with largest remainder (Lemma 7.4).
    let mut rows: Vec<(IntentId, u128, u128, [u8; 32])> = heavy
        .iter()
        .map(|i| {
            let num = crossed * i.size as u128;
            (i.id, num / q_heavy, num % q_heavy, tie_key(seed, &i.nonce))
        })
        .collect();
    let floor_sum: u128 = rows.iter().map(|r| r.1).sum();
    let mut left = crossed - floor_sum;
    rows.sort_by(|a, b| b.2.cmp(&a.2).then(a.3.cmp(&b.3)));
    for row in rows.iter_mut() {
        if left == 0 { break; }
        row.1 += 1;
        left -= 1;
    }
    fills.extend(rows.into_iter().map(|r| (r.0, r.1 as u64)));
    PairCross { fills, crossed: crossed as u64, residual: q_buy as i128 - q_sell as i128 }
}

fn tie_key(seed: &[u8; 32], nonce: &[u8; 16]) -> [u8; 32] {
    let mut h = Sha256::new();
    h.update(seed);
    h.update(nonce);
    h.finalize().into()
}
```

Routing the Residual

After crossing, each heavy-side Intent i has an uncrossed remainder $x_i - f_i$. That remainder can reach public liquidity only at a price worse than the mid, so it is routed only if the trader's limit tolerates the worst price the route may produce.

Definition 7.3 (Route eligibility and the slippage bound). For each pair with $R \neq 0$, the Coprocessor obtains a quote for the Residual direction from Jupiter and sets a slippage bound $\beta_\pi^{(k)} \geq \beta_0$ from it, with floor $\beta_0 = 50$ bps (parameter), the default tolerance used in Jupiter's own quote examples [6]. The worst admissible prices are $\bar{p}^+ = m(1+\beta)$ for a buy Residual and $\bar{p}^- = m(1-\beta)$ for a sell Residual. A heavy-side remainder is *route-eligible* if $\ell_i \geq \bar{p}^+$ (buy) or $\ell_i \leq \bar{p}^-$ (sell). The routed size is

$$\tilde{R}_\pi = \sum_{i \in \mathcal{H},\ \text{route-eligible}} (x_i - f_i), \qquad |\tilde{R}_\pi| \leq |R_\pi^{(k)}|, \tag{7.6}$$

and a remainder that is not route-eligible rests, unfilled, for the next Crossing until its expiry τ.

Jupiter's Swap API V2 returns, with every route, the expected output and an `otherAmountThreshold`, the minimum output after applying the requested `slippageBps`; its Router path `/swap/v2/build` returns raw swap instructions routed by the Metis engine across Solana DEXs, for a program to compose into its own transaction [6]. Privium uses the exact-input form in both directions, so the Pool always knows what it spends:

- *Sell Residual.* Input $A_{\text{in}} = |\tilde{R}|$ base units of a; minimum output $A_{\text{out}} = \lceil |\tilde{R}| \cdot m(1-\beta) \rceil$ in q.
- *Buy Residual.* Input $A_{\text{in}} = \lceil \tilde{R} \cdot m \rceil$ in q; minimum output $A_{\text{out}} = \lceil \tilde{R}/(1+\beta) \rceil$ base units of a.

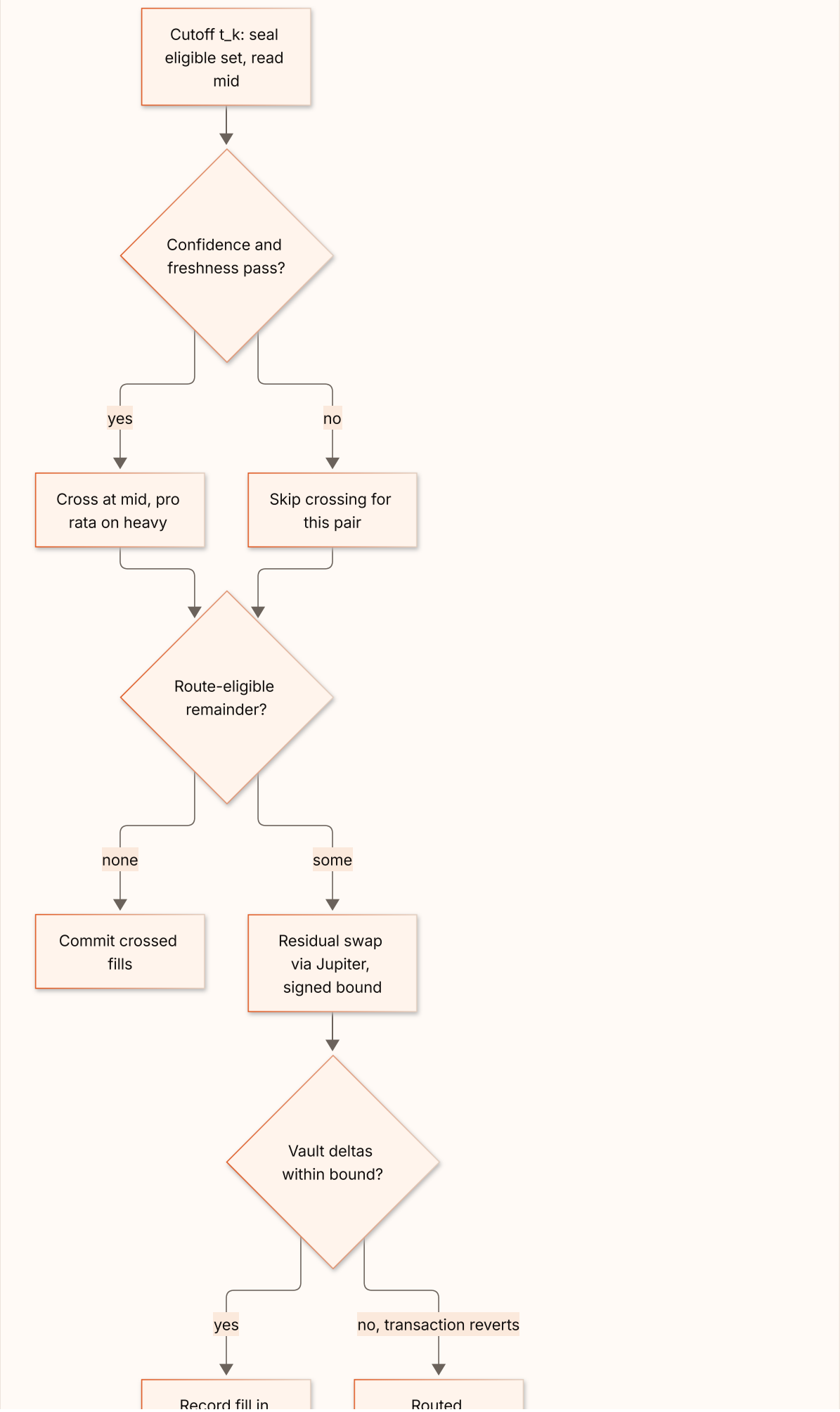
In the buy case the Pool spends the mid value of the Residual and receives slightly less base than $\tilde{R}$; routed buyers share the received base pro rata to their routed remainders and the shortfall rests or expires with them. Both cases realize an average price inside $[\bar{p}^{-}, \bar{p}^{+}]$.

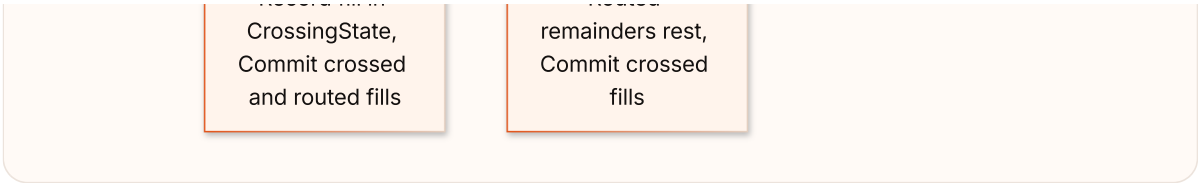
The enclave signs the bound with sk_C as a structured message under the domain tag `privium/route/v1`, and the Pool's Residual instruction executes the Jupiter route and then checks the vault deltas, not the route's own claims:

```rust
#[derive(AnchorSerialize, AnchorDeserialize)]
pub struct ResidualOrder {
    pub crossing: u64,          // k
    pub pair: [Pubkey; 2],      // (a, q)
    pub input_mint: Pubkey,
    pub amount_in: u64,         // A_in
    pub min_out: u64,           // lower bound on A_out
    pub deadline_slot: u64,     // t^(k) + Delta
    pub route_hash: [u8; 32],   // H(Jupiter id | data | accounts), Chapter 9
}

pub fn settle_residual_bounds(
    order: &ResidualOrder,
    in_before: u64, in_after: u64,
    out_before: u64, out_after: u64,
    slot: u64,
) -> Result<(u64, u64)> {
    require!(slot ≤ order.deadline_slot, PriviumError::ResidualExpired);
    let spent = in_before.checked_sub(in_after).ok_or(PriviumError::VaultUnderflow)?;
    let got = out_after.checked_sub(out_before).ok_or(PriviumError::VaultUnderflow)?;
    require!(spent ≤ order.amount_in, PriviumError::OverSpend);
    require!(got ≥ order.min_out, PriviumError::SlippageBound);
    Ok((spent, got)) // recorded in CrossingState as this Crossing's fill
}
```

The signature check on `ResidualOrder` uses the ed25519 instruction-introspection pattern of Chapter 9. Because the check reads balances of Pool-owned vaults before and after the CPI, its guarantee does not depend on which DEXs the route touches.


Cutoff t_k: seal eligible set, read mid
Confidence and freshness pass?
yes
no
Cross at mid, pro rata on heavy
Skip crossing for this pair
Route-eligible remainder?
none
some
Commit crossed fills
Residual swap via Jupiter, signed bound
Vault deltas within bound?
yes
no, transaction reverts
Record fill in
Routed



The Commit that follows posts the new Ledger root computed from the crossed fills and the recorded vault deltas. Chapter 4 proves the conservation invariant over that root; the Crossing's contribution is the following local identity.

Lemma 7.5 (Crossing balance). In Crossing k, for pair π with fee rate ϕ, the crossed fills move exactly X_π base units from sellers to buyers and $X_\pi m$ quote units (before fees) from buyers to sellers; fees move $\phi X_\pi m$ from each side out of the Ledger, and the Commit pays the Crossing's total F_q from the quote vault to the fee recipient (Chapter 9); the routed fills move exactly the recorded deltas $(\mathrm{spent}, \mathrm{got})$ between routed traders and the vaults. Hence the Crossing changes $\sum_u b_{u,a} - V_a$ by zero for the base asset and $\sum_u b_{u,q} - V_q$ by zero for the quote asset once the fee is paid, which is the balance condition (4.4) of Chapter 4.

Proof. Each crossed unit is debited from one Ledger entry and credited to another, so the vaults do not move and the sum of entries is unchanged except for the fee, which leaves the entries and the quote vault by the same amount when the Commit pays it out. Each routed unit moves between the vaults and the market, and the routed traders' entries change by exactly the recorded vault deltas, distributed pro rata, so entries and vaults change together. □

Theorem 7.6 (Limit safety). Every fill produced by a Crossing executes at a price no worse than its Intent's limit.

Proof. Crossed fills execute at m, and only active Intents cross, for which $\ell_i \geq m$ (buy) or $\ell_i \leq m$ (sell). A routed buyer pays A_{in} shared pro rata for at least A_{out} shared in the same proportion, so its average price is at most $\lceil \tilde{R} m \rceil / \lceil \tilde{R}/(1+\beta) \rceil$, which rounds to $\bar{p}^{+} \leq \ell_i$ up to one base unit by route eligibility; the vault check in `settle_residual_bounds` makes any execution outside the bound revert. The sell case is symmetric with $\bar{p}^{-} \geq \ell_i$. A trader whose fill is both crossed and routed pays a volume-weighted average of two prices each within its limit, which is within its limit. □

Price improvement versus routing

The question a trader asks is whether a Crossing is cheaper than the same trade sent straight to an aggregator. We answer it with a cost model that is standard for automated market makers and state the answer as an inequality.

Model. Measure every cost against the mid m, in quote units. A direct trade of signed base size y through public liquidity pays three components: a per-unit fee f (DEX fees along the route), a per-unit half-spread $h \geq 0$ (the distance between the mid and the best executable marginal price), and price impact. For impact we use the property that makes constant-function market makers tractable: their state after a sequence of swaps depends only on the net flow, so the total quote paid by a sequence of trades in one pool is a function of their sum [10]. Write $K(y) \geq 0$ for the total impact cost, relative to the marginal price, of net flow y, with $K(0) = 0$ and K convex. A batch of direct trades $y_1, \ldots, y_n$ executed in any order within one interval then costs, in aggregate,

$$C_{\text{direct}} = m\,(f+h) \sum_j |y_j| \;+\; K\Big(\sum_j y_j\Big). \tag{7.7}$$

The same Intents through a Crossing, where the Pool pays the protocol fee ϕ on crossed and routed notional (the fee itself belongs to Chapter 12), cost

$$C_{\text{Privium}} = m\,\phi\,(2X_\pi + |R|) \;+\; m\,(f+h)\,|R| \;+\; K(R), \tag{7.8}$$

taking every active remainder as route-eligible so that both sides execute the same quantities.

Theorem 7.7 (Expected-cost inequality). Under the model, $C_{\text{Privium}} \leq C_{\text{direct}}$ for a batch if and only if

$$\phi \;\leq\; \gamma_\pi\,(f+h), \tag{7.9}$$

where γ_π is the crossing rate. In expectation over batch composition, the Crossing is cheaper in aggregate if and only if $\phi \cdot \mathbb{E}[G_\pi] \leq (f+h)\,\mathbb{E}[2X_\pi]$.

Proof. In the direct case $\sum_j |y_j| = G_\pi$ and $\sum_j y_j = R$, since the direct trades are the active Intents themselves. Substituting into (7.7) and subtracting (7.8), the impact terms $K(R)$ cancel and

$$C_{\text{direct}} - C_{\text{Privium}} = m(f+h)(G_\pi - |R|) - m\phi\,G_\pi = m\,G_\pi\big((f+h)\gamma_\pi - \phi\big),$$

using $G_\pi - |R| = 2X_\pi$ and $2X_\pi + |R| = G_\pi$. The sign is that of $(f+h)\gamma_\pi - \phi$. The expectation statement follows by linearity of $\mathbb{E}$ applied to the same difference. □

Two features of (7.9) matter. First, impact cancels: aggregation neither creates nor removes impact for path-independent liquidity, so the entire advantage of crossing is the fee and spread that crossed units never pay, and it scales with the crossing rate. Second, (7.9) is a statement about aggregate cost, and it is distributed unevenly in a way that favors the light side.

Corollary 7.8 (Light side). A light-side trader of size x is filled in full at the mid and pays $m\phi x$, against $m(f+h)x + K(x)$ to trade alone. The Crossing is cheaper for that trader whenever $\phi \le f + h + K(x)/(mx)$, which (7.9) implies for every $\gamma_\pi \le 1$.

Proof. Theorem 7.1 fills the light side in full at m. The standalone cost is (7.7) with one trade. The implication holds since $\gamma_\pi(f+h) \le f + h \le f + h + K(x)/(mx)$. □

A heavy-side trader receives fraction $X_\pi / Q_{\mathcal{H}}$ at the mid and the rest inside the Residual, sharing the Residual's impact per unit. That trader's comparison depends on how its own size relates to the Residual, which is why the aggregate statement is the right object: under (7.9) the batch as a whole pays less than its members would have paid, and the light side's saving is guaranteed. How the crossing rate grows with flow, and what that does to the cost curve over time, is the subject of Chapter 12.

Multi-pair consistency

Every pair in Privium is quoted in q, and a trader may hold Intents in several pairs in the same Crossing. Two consistency properties are needed: the outcome must not depend on the order in which the enclave processes pairs, and internal prices must admit no cycle that extracts value from the Pool.

Definition 7.4 (Reservation). At the start of Crossing k, for each trader u the enclave reserves, in the order of the Intents' hashes, the worst-case debit (7.1) of each eligible Intent against $b_{u,\cdot}^{(k-1)}$. An Intent whose reservation does not fit in the remaining unreserved balance $\hat{b}$ is ineligible for this Crossing. Proceeds of a Crossing are credited after all pairs settle and become reservable in Crossing $k+1$, whose cutoff falls at least Δ slots after Commit k.

Theorem 7.9 (Pair independence). With reservations, the per-pair outcomes of Crossing k are independent of the order in which pairs are processed, may be computed in parallel, and leave every balance $b_{u,a}^{(k)} \ge 0$.

Proof. The eligible set of each pair is fixed by Definition 7.4 before any pair is processed, and the per-pair kernel reads only its own eligible set, the mid of its own pair and the seed. No pair's output is an input to another's, so the outcome is a product of independent maps and is order-invariant. For nonnegativity, each Intent's actual debit is bounded by its reservation: a crossed or routed buy pays at most $\ell_i(1+\phi)$ per unit by Theorem 7.6, and a sell gives at most x_i. The reservations of one trader sum to at most the trader's committed balance in each asset, so total debits do not exceed $b_{u,a}^{(k-1)}$, and credits are nonnegative. □

Proposition 7.10 (No internal cycle). Any sequence of crossed fills within one Crossing that starts and ends in the same asset returns at most the starting quantity.

Proof. Every crossed fill exchanges an asset against q at that pair's mid, less fees. A cycle $a_1 \to q \to a_2 \to q \to \cdots \to a_1$ multiplies the quantity by $\prod_j (m_{a_j}/m_{a_{j+1}})$ times a product of factors $(1-\phi)$, and the product of mid ratios around a cycle telescopes to 1. □

Each pair's Residual is a separate swap, and all of them touch the q vault. On Solana a transaction declares the accounts it writes, and transactions writing the same account are serialized [2]; Chapter 9 gives the write-lock layout that lets Residuals in different pairs land in one leader span.

MEV: batched Residuals versus individual swaps

A public swap reveals its size, direction and slippage tolerance before it executes, and a searcher who can order transactions around it extracts value by front-running and back-running it. Daian et al. documented this on Ethereum [3], and Zhou et al. formalized the sandwich attack against constant-product pools and showed that the attacker's optimal profit is bounded by the victim's slippage tolerance [4]. We use that bound as the adversary model.

Definition 7.5 (Sandwich adversary). For a swap of size y with slippage tolerance β, a sandwich adversary chooses a front-run and back-run around it. The victim's execution degrades by at most what its bound allows, so the value transferred from victim to adversary is at most $\beta\, m\, |y|$ plus impact the victim would have paid anyway.

Theorem 7.11 (Extraction bound). Let a batch of active Intents with gross size G_π and Residual R be executed either individually, each with tolerance β, or through a Crossing with Residual bound β. The maximal value a sandwich adversary can extract is at most $\beta\, m\, G_\pi$ in the first case and at most $\beta\, m\, |R| = \beta\, m\, (1-\gamma_\pi)\, G_\pi$ in the second.

Proof. Individually, each swap y_j exposes at most $\beta m |y_j|$ by Definition 7.5, and the bound sums to $\beta m G_\pi$. Through a Crossing, crossed units never enter a public transaction: they are Ledger updates inside the enclave, committed as ciphertext (Chapter 4), so they offer no execution to degrade. The only public swap is the Residual, whose realized price is enforced on chain by the vault check, so its exposure is at most $\beta m |\tilde{R}| \le \beta m |R|$. The identity $|R| = (1-\gamma_\pi) G_\pi$ follows from (7.4). □

The theorem understates the difference in two ways. First, the Residual's bound is set per Crossing from a fresh quote with a floor of β_0, while individual traders often send loose tolerances. Second, a sandwich needs a target worth the cost of the attack, and the Residual reveals only a net figure per pair per Crossing: no wallet, no trader history, nothing a searcher can model as one participant's next move. What the Residual does reveal, and how little it says about any trader, is analyzed in Chapter 8.

Adverse selection at the mid

In a continuous market a fast trader who sees the price move hits a quote that has not yet moved. A Crossing offers no such quote: the batch is sealed before `open_crossing` reads the mid, and the pair crosses only while that mid lies within κ of a market price read after the seal.

Proposition 7.12 (No pick-off at the mid). Let every eligible Intent be sealed before $t^{(k)}$, and let $(m(t))$ be a positive martingale with respect to a filtration $(\mathcal{F}_t)$ that contains a trader's information, with the reference read at a slot $t_r \geq t^{(k)}$ as $P = m(t_r)$. Let the cross execute at $m_\pi^{(k)}$ only on the event $C = \{|m_\pi^{(k)} - m(t_r)| \leq \kappa\, m(t_r)\}$, with $m_\pi^{(k)}$ known by t_r, and let an Intent that does not cross gain nothing at the mid. For any Intent whose side s and size are $\mathcal{F}_{t_0}$-measurable with $t_0 < t^{(k)}$, the expected per-unit gain relative to the cross price, marked at any slot $T \geq t_r$, is at most $\kappa\, m(t_0)$ in absolute value. The rest of the counterparty's per-unit exposure is the increment $m(T) - m(t_r)$, whose root mean square does not depend on t_0.

Proof. The gain per unit is $s\,\mathbf{1}_C\,(m(T) - m_\pi^{(k)}) = s\,\mathbf{1}_C\,(m(T) - m(t_r)) + s\,\mathbf{1}_C\,(m(t_r) - m_\pi^{(k)})$. The event C is $\mathcal{F}_{t_r}$-measurable and s is $\mathcal{F}_{t_0}$-measurable with $t_0 < t_r \leq T$, so the tower property gives $\mathbb{E}[s\,\mathbf{1}_C\,(m(T) - m(t_r)) \mid \mathcal{F}_{t_0}] = \mathbb{E}\big[s\,\mathbf{1}_C\,\mathbb{E}[m(T) - m(t_r) \mid \mathcal{F}_{t_r}] \mid \mathcal{F}_{t_0}\big] = 0$. On C the second term is at most $\kappa\, m(t_r)$ in absolute value, so its conditional expectation is at most $\kappa\,\mathbb{E}[m(t_r) \mid \mathcal{F}_{t_0}] = \kappa\, m(t_0)$ in absolute value. The first term is an increment of the martingale after t_r and is independent of when the Intent was formed. □

The seal before the read, with the band at κ, is what removes the speed race: knowing the mid before the cutoff is worth at most a fraction κ of the price per unit, in expectation. What remains is symmetric price variance over the interval until the fill is marked, the quantity Budish et al. identify as the residual cost of a frequent batch [1].

Worked example

This is an example with illustrative inputs; none of its figures are measurements or market data.

One pair, SOL/USDC, in Crossing k. Mid $m = 150.00$ USDC, confidence within (7.3), slippage bound $\beta = 50$ bps from the quote. Eligible Intents:

Intent	Side	Size (SOL)	Limit (USDC)	Active at mid
A	buy	60	151.00	yes
B	buy	30	150.50	yes
C	buy	10	149.00	no ($\ell < m$)
D	sell	40	149.50	yes
E	sell	20	145.00	yes

Active sizes: $Q^+ = 90$, $Q^- = 60$, so $X = 60$, $R = +30$, $G = 150$ and $\gamma = 2 \cdot 60/150 = 0.8$. Sellers D and E are the light side and cross in full at 150.00, receiving 9,000 USDC before fees. Buyers share 60 SOL pro rata by (7.5): A receives $60 \cdot 60/90 = 40$ and B receives $60 \cdot 30/90 = 20$, each at 150.00.

Remainders are A: 20 and B: 10. The buy-side worst price is $\bar{p}^+ = 150 \cdot 1.005 = 150.75$. A's limit 151.00 clears it; B's limit 150.50 does not, so B's remainder rests for the next Crossing. The routed size is $\tilde{R} = 20$. The Residual spends $A_{\text{in}} = 20 \cdot 150 = 3{,}000$ USDC with minimum output $\lceil 20/1.005 \rceil$, about 19.9005 SOL. Suppose (illustrative) the route returns 19.94 SOL, a realized price of $3000/19.94 \approx 150.45$, inside the bound.

Outcome before fees: A holds $40 + 19.94 = 59.94$ SOL for 9,000 USDC, an average of about 150.15, under its limit by Theorem 7.6; the 0.06 SOL shortfall rests with A's Intent. B holds 20 SOL at 150.00 with 10 resting. C rests untouched. D and E sold in full at the mid. Of 150 SOL of active gross size, 20 reached a public pool. By (7.9), the batch is cheaper than five direct swaps whenever $\phi \leq 0.8\,(f + h)$, and by Theorem 7.11 a sandwich adversary faces at most $20/150$ of the exposure the five swaps would have offered.

References

1. E. Budish, P. Cramton, J. Shim, “The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response”, Quarterly Journal of Economics 130(4), 2015. https://doi.org/10.1093/qje/qjv027
2. N. Lokhande et al., “Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana”, IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
3. P. Daian et al., “Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability”, IEEE Symposium on Security and Privacy, 2020. https://arxiv.org/abs/1904.05234
4. L. Zhou, K. Qin, C. F. Torres, D. V. Le, A. Gervais, “High-Frequency Trading on Decentralized On-Chain Exchanges”, IEEE Symposium on Security and Privacy, 2021. https://arxiv.org/abs/2009.14021
5. D. Easley, N. M. Kiefer, M. O’Hara, “Cream-Skimming or Profit-Sharing? The Curious Role of Purchased Order Flow”, Journal of Finance 51(3), 1996. https://doi.org/10.1111/j.1540-6261.1996.tb02706.x
6. “Swap API” (Metis quote, `otherAmountThreshold`, `slippageBps`; Swap V2 `/build`), Jupiter Developer Docs, checked 2026-11-16. https://developers.jup.ag/docs/swap-api
7. “Best Practices” (confidence intervals and staleness), Pyth Network Docs, checked 2026-11-16. https://docs.pyth.network/price-feeds/best-practices
8. “Reduced Slot Times”, Solana Foundation, 2026. https://solana.com/upgrades/reduced-slot-times
9. W. Thomson, “Axiomatic and Game-Theoretic Analysis of Bankruptcy and Taxation Problems: A Survey”, Mathematical Social Sciences 45(3), 2003. https://doi.org/10.1016/S0165-4896(02)00070-7
10. G. Angeris, H.-T. Kao, R. Chiang, C. Noyes, T. Chitra, “An Analysis of Uniswap Markets”, Cryptoeconomic Systems, 2021. https://arxiv.org/abs/1911.03380

Chapter 8

Unlinkability

Confidentiality hides what a balance holds; unlinkability hides whose it is. A trader in Privium deposits from one wallet, trades through sealed Intents, and exits to another, and the question this chapter answers is how well an observer can connect the two wallets, or connect either of them to the trades in between. We fix three adversaries, define the public transcript they see, and measure what it says about any one trader with Shannon entropy and min-entropy over the anonymity set. We then bound the three channels through which linkage leaks in any shielded pool (amounts, timing, and flow) and give the countermeasures that close each one: denominations, exit delays and Ring exits. The chapter ends with a formal unlinkability game and the theorem Privium achieves in it: the cryptographic layers leak nothing beyond a stated leakage function, and the statistical bounds of the earlier sections measure exactly what that function reveals.

What the chain shows

Privium's public surface is small and fixed. Every Intent is encrypted to the Coprocessor's attested key (Chapter 3 gives the HPKE construction, Chapter 5 the attestation), and Intents reach the enclave through its relay or as opaque blobs posted by a relayer, so no Intent carries a trader's wallet on chain. The Ledger is a Merkle root over ciphertext leaves (Chapter 4). What remains visible is the movement of tokens across the boundary of the Pool and the net flow per pair.

Definition 8.1 (Leakage function). For an execution of Privium up to Crossing k, the leakage $\mathcal{L}^{(k)}$ is the sequence of:

1. *Deposits* (w, a, d, t): source wallet w, asset a, amount d, slot t.
2. *Exits* (w', a, e, t): destination address w', asset, amount, slot.
3. *Residuals* $(k, \pi, \mathrm{spent}, \mathrm{got})$ for every pair with a routed Residual (Chapter 7).
4. *Commit metadata*: the root $\rho^{(k)}$, the Crossing index, the slot the Commit lands in, the number of live leaves, the set of assets it carries (empty for a heartbeat Commit, Chapter 10), the *rewrite pattern* (for each asset the Commit carries, the positions it vacates and the positions it fills), and the success or failure of each public instruction.

Commits follow a *rewrite discipline* built on the leaf of Chapter 4. In every asset a Commit carries, it rewrites the leaves the Crossing changed together with decoys drawn uniformly from the asset's other live leaves, so that at least $m_0 = 64$ leaves of the asset are rewritten (parameter: one aggregated range proof covers 64 leaves at 16.5 bytes per leaf, Chapter 4, and a batch of 64 hides each changed leaf among 63 others). Each Crossing that settles also carries one asset in a fixed rotation over the admitted assets, whether the Crossing changed it or not, so no single carried asset shows that it moved; a heartbeat Commit carries no asset. Every rewritten leaf moves to a fresh uniform position with a fresh owner tag, fresh commitment randomness and fresh ciphertexts. In the Encifher model, on which the Ledger builds, programs hold only handles to ciphertexts [10]; Privium publishes the ciphertexts on Solana itself (Chapter 4). Everything an observer of the chain sees is then computable from $\mathcal{L}^{(k)}$ plus ciphertexts whose distribution does not depend on plaintext. Theorem 8.9 makes this precise.

The table compares that surface with a trader's footprint when trading from a wallet.

Observable	Trading from a wallet	Trading through Privium
Wallet behind each swap	yes	no swap carries a trader wallet
Size and side of each order	yes	only the net Residual per pair per Crossing
Balance history	yes	ciphertext leaves under one root
Entry and exit of funds	the same wallet	deposit and exit, linked only through $\mathcal{L}$
Position across tokens	yes	not observable

Adversary model

We consider three adversaries of increasing reach. All three are computationally bounded in $\lambda = 128$ and see every Solana transaction, including every ciphertext the Commits publish.

- **Chain observer** $\mathcal{A}_{\mathrm{chain}}$: passive; sees the full transcript.
- **Wallet tracker** $\mathcal{A}_{\mathrm{track}}$: additionally holds auxiliary labels on wallets (exchange deposit addresses, known traders) and controls a set $\mathcal{C} \subset \mathcal{U}$ of Pool accounts of its own, through which it may deposit, trade and exit freely. Clustering of addresses by co-spending and change heuristics is the standard toolkit of such an adversary [6].

- **Timing analyst $\mathcal{A}_{\text{time}}$**: additionally exploits slot-level timing of deposits and exits, including the heuristic that a withdrawal close in time to a deposit of matching amount came from it, one of the heuristics Wang et al. used to link deposits to withdrawals in zero-knowledge mixers [5].

The guarantees in this chapter rest on three assumptions, each proved or specified elsewhere: the integrity and confidentiality of the Coprocessor enclave (Chapter 5), an honest threshold of the Council, with fewer than t of n shares in adversarial hands (Chapter 6), and the IND-CCA2 security of the Intent encryption and the security of the Ledger's ciphertexts (Chapters 3 and 4). Traders who disclose through a Viewing key or a Receipt disclose by choice (Chapter 11), and the game below excludes those disclosures for its challenge accounts.

The anonymity set and its measures

Definition 8.2 (Anonymity set). For an Exit e at slot t_e, the anonymity set $\mathcal{N}(e)$ is the set of honest accounts $u \in \mathcal{U} \setminus \mathcal{C}$ that hold a live Ledger leaf at t_e. Write $N(e) = |\mathcal{N}(e)|$.

Because every pair crosses against the quote asset and balances are encrypted, any live account can fund an Exit in any asset: an account that deposited SOL may exit in USDC after a Crossing, and nothing public says it traded. The set is therefore every live honest account, not only those that deposited the exited asset.

Size alone overstates protection when some members are much more likely than others. Let U_e be the random variable "the owner of e", and let $P_e(u) = \Pr[U_e = u \mid \mathcal{L}]$ be the adversary's posterior.

Definition 8.3 (Entropy measures). The Shannon anonymity of e is $\mathsf{H}(U_e \mid \mathcal{L} = L) = -\sum_u P_e(u) \log_2 P_e(u)$, with effective set size 2^{H} [1]. The degree of anonymity is $\mathsf{d}(e) = \mathsf{H} / \log_2 N(e) \in [0, 1]$ [2]. The min-entropy anonymity is $\mathsf{H}_\infty(U_e \mid \mathcal{L} = L) = -\log_2 \max_u P_e(u)$, and the average conditional min-entropy over transcripts is

$$\tilde{\mathsf{H}}_\infty(U_e \mid \mathcal{L}) = -\log_2 \mathbb{E}_L \Big[\max_u \Pr[U_e = u \mid \mathcal{L} = L] \Big]. \tag{8.1}$$

Shannon entropy measures the average uncertainty; min-entropy measures the adversary's best single guess, and is the right measure for a tracker who acts on one guess. Smith argues for min-entropy on exactly these grounds in quantitative information flow [3].

Lemma 8.1 (Ordering of measures). For every e and transcript, $\mathsf{H}_\infty(U_e \mid L) \le \mathsf{H}(U_e \mid L) \le \log_2 N(e)$, with both equalities if and only if the posterior is uniform on $\mathcal{N}(e)$.

Proof. $\mathsf{H} = \mathbb{E}[-\log_2 P_e(U)] \ge \min_u(-\log_2 P_e(u)) = \mathsf{H}_\infty$, with equality iff P_e is constant on its support. The upper bound is Gibbs' inequality against the uniform distribution on $\mathcal{N}(e)$, with equality iff P_e is that distribution. Both conditions coincide when the support is all of $\mathcal{N}(e)$. □

The set as a function of Pool activity

The anonymity set grows with the Pool. Model honest account openings as a stationary process with rate $\lambda_{\mathcal{U}}$ per slot and account lifetimes (slots from first deposit to last exit) with mean $\mathbb{E}[T_{\mathcal{U}}]$.

Proposition 8.2 (Set size). In steady state, $\mathbb{E}[N] = \lambda_{\mathcal{U}} \, \mathbb{E}[T_{\mathcal{U}}]$.

Proof. The number of live honest accounts is the number of customers in a stationary queueing system with arrival rate $\lambda_{\mathcal{U}}$ and mean sojourn time $\mathbb{E}[T_{\mathcal{U}}]$; Little's law gives the time-average number in the system as their product, for any arrival and sojourn distributions with finite means [8]. □

Two consequences follow. The Pool's anonymity set is not the number of trades but the number of resident accounts, and an account that stays in Privium as its trading account, the way a client stays at a broker, raises $\mathbb{E}[T_{\mathcal{U}}]$ and with it everyone's set. Second, only honest accounts count: an adversary who opens accounts of its own does not enlarge the set it has to search, because it knows which members are its own. Proposition 8.2 bounds the ceiling; the next sections bound how close the posterior stays to uniform below it.

Linkage channels

A shielded pool leaks through the amounts and times at its boundary and through the aggregate flow it shows inside. We bound each channel in turn.

Amount correlation

An Exit of exactly the amount a unique deposit brought in links the two with near certainty. Kappos et al. showed that matching the values that enter and leave Zcash's shielded pool links a substantial part of its transactions [4]. The bound we need is a general fact about how much any observation can leak.

Lemma 8.3 (Min-entropy leakage of a finite observation). Let U be secret and O an observation taking at most M values. Then

$$\tilde{\mathsf{H}}_\infty(U \mid O) \;\geq\; \mathsf{H}_\infty(U) - \log_2 M. \tag{8.2}$$

Proof. $\mathbb{E}_O[\max_u \Pr[U=u \mid O]] = \sum_o \max_u \Pr[U=u, O=o] \leq \sum_o \max_u \Pr[U=u] = M \cdot 2^{-\mathsf{H}_\infty(U)}$. Taking $-\log_2$ gives (8.2). This is the bound of Smith [3]: leakage is at most the logarithm of the number of distinguishable outputs. □

Lemma 8.3 is useless for exact amounts, where M is as large as the token's range, and decisive for denominated ones.

Definition 8.4 (Denominations). For asset a with d_a decimals, the denomination unit is $\delta_a = 10^{d_a - 2}$ base units (parameter: one hundredth of a whole token keeps the sub-unit remainder that stays in the Ledger below any meaningful size), and the denomination set is $\mathcal{D}_a = \{\delta_a 2^j : 0 \leq j < J\}$ with $J = 24$ (parameter: 24 binary denominations cover a single Exit of up to $(2^{24} - 1)\delta_a$, about 1.68×10^5 whole tokens). A denominated Exit of amount e is executed as the pieces $\delta_a 2^j$ for each set bit j of $\lfloor e/\delta_a \rfloor$, each piece a separate Exit with its own delay (next section). Deposits through the SDK are split the same way.

Theorem 8.4 (Amount leakage under denominations). Each denominated piece lowers the adversary's average min-entropy about its owner by at most $\log_2 J$ bits. If deposits are denominated as well, the candidate set for a piece of denomination $\delta_a 2^j$ is the set of honest accounts live at its slot, and no exact-match link exists.

Proof. A piece's amount takes at most J values, so (8.2) with $M = J$ gives the first claim. For the second, a piece of denomination $\delta_a 2^j$ is a sum the Ledger can debit from any account whose encrypted balance covers it, and since deposits arrive in the same denominations, every denomination class contains every honest depositor who used that class; balances after trading are not observable, so the amount does not narrow the set below the live accounts. □

The contrast is sharp. With exact amounts, a deposit of 1,234.567 USDC followed by an Exit of 1,234.567 USDC is a fingerprint. With denominations, both are sums of pieces shared by everyone in the same class, and the pieces themselves are scattered in time.

Timing correlation

An Exit soon after a deposit, in a quiet Pool, has few plausible sources. We bound the timing analyst's success in a continuous-time model with slots as the unit.

Model. Honest deposits in the relevant class (one denomination of one asset, or all deposits if amounts are not denominated) form a stationary Poisson process with rate λ per slot. Each deposit's matching Exit occurs after a holding time T drawn independently with density g bounded by $g_{\max}$. The adversary $\mathcal{A}_{\text{time}}$ sees all deposit slots in the class and the slot t_e of a target Exit, and guesses which deposit it came from.

Theorem 8.5 (Timing bound). The adversary's success probability is at most

$$\Pr[\text{link}] \;\leq\; \frac{6\, g_{\max}}{\lambda}. \tag{8.3}$$

Proof. Condition on a target Exit at t_e. By the Palm property of the Poisson process (Slivnyak's theorem), the other deposits form a Poisson process of rate λ independent of the target, whose own deposit lies at $t_e - T^*$ with $T^* \sim g$. All deposits being exchangeable a priori, the posterior that deposit j is the source is proportional to $g(t_e - t_j)$, so the best guess succeeds with probability

$$\frac{\max_j g(t_e - t_j)}{g(T^*) + S}, \qquad S = \sum_{j \neq *} g(t_e - t_j),$$

which is at most $\min(1, g_{\max}/S)$. Campbell's theorem gives $\mathbb{E}[S] = \lambda \int_0^\infty g = \lambda$ and $\mathrm{Var}[S] = \lambda \int_0^\infty g^2 \leq \lambda\, g_{\max}$. By Chebyshev, $\Pr[S < \lambda/2] \leq \Pr[|S - \lambda| \geq \lambda/2] \leq 4 g_{\max}/\lambda$. On the complement, $g_{\max}/S \leq 2 g_{\max}/\lambda$. Hence $\Pr[\text{link}] \leq 4 g_{\max}/\lambda + 2 g_{\max}/\lambda$. □

The bound depends on the holding-time density only through its peak. A Pool whose traders hold for varied times is already hard to time; the danger is a sharp peak, a trader who always exits a fixed number of slots after depositing. An exit delay flattens every peak.

Corollary 8.6 (Exit delay). If each Exit is executed after an additional delay D drawn independently from the exponential distribution with mean μ slots, then for any trader behavior

$$\Pr[\text{link}] \leq \frac{6}{\lambda \mu}, \qquad \tilde{\mathsf{H}}_\infty \geq \log_2 \frac{\lambda \mu}{6}. \tag{8.4}$$

Proof. The holding time becomes $T = T_0 + D$ with T_0 independent of D, so its density is the convolution $g = g_0 * f_D$, and $g(t) = \int g_0(s) f_D(t-s)\, ds \leq \sup f_D \int g_0 = 1/\mu$ for any distribution of T_0, including a point mass. Substitute $g_{\max} \leq 1/\mu$ into (8.3); the entropy bound is (8.1) applied to the success probability. □

The exponential is the natural choice: its peak density is $1/\mu$, and it is memoryless, so an observer who has waited t slots since an Exit request learns nothing about the remaining delay. The Coprocessor samples D inside the enclave at the moment the Exit request is decrypted, so the schedule is unknown to everyone, the requester included. The recommended mean is $\mu = 9{,}000$ slots (parameter): 30 minutes at the slot time of Chapter 9, long enough to cover half an hour of Pool deposits and short enough for a trading account. A trader can request an immediate Exit, which is then bounded by (8.3) with the trader's own holding-time density.

Denominations and delays compose. With denominated amounts, each piece is timed against deposits of its own denomination only, so λ in (8.4) becomes the class rate λ_j; the pieces of one Exit are delayed independently, so they arrive interleaved with other accounts' pieces of other denominations and are not observable as one Exit.

Flow correlation

Inside the Pool the public traces of trading are the Residual, the rewrite pattern and the Crossings' times. For pair π in Crossing k, $\mathcal{L}$ contains the vault deltas of the routed size $\tilde{R}_\pi^{(k)}$ (Chapter 7), the positions rewritten in each carried asset, and nothing about the crossed size X_π, the number of Intents, or their owners.

Apart from the heartbeat, which carries no asset and settles nothing, a Crossing goes on chain only when it settles something (Chapter 10), so its slot shows that the Pool had work in that batch, and its carried set shows in which assets, alongside the one in rotation. What the work was, a trade, a cancel, a deposit credit, an Exit piece, an expiry or a resting order the market reached, and whose it was, stays sealed: the schedule of Crossings follows the slots at which Intents reached intake, their expiries and limits against the live price, and the boundary events of $\mathcal{L}$, and takes no account identity as input.

Proposition 8.7 (Residual and rewrite leakage). Under the rewrite discipline and the confidentiality of the enclave: (a) for every honest account u and every Crossing, the adversary's posterior that u had an Intent in pair π given $\mathcal{L}$ is the same for any two honest accounts with identical public deposit and exit records; (b) for any leaf, the adversary's largest posterior that a given position holds it never increases from one Crossing to the next, and is at most $1/m_0$ from the first rewrite of the leaf onward.

Proof. (a) The Residual is a deterministic function of the multiset of eligible Intents' sizes, sides and limits, the mid and the seed (Chapter 7); it does not take account identities as input. The slot and carried set of each Crossing are functions of the times and assets of the work it settles, which carry no account identity either. A rewrite batch is a set of vacated and filled positions with fresh tags and ciphertexts, which carry no plaintext and no account identity under the assumptions of Theorem 8.9. Two honest accounts with identical public records are therefore exchangeable given $\mathcal{L}$, so their posteriors coincide. (b) Within a batch of $m \geq m_0$ leaves, the leaves are relocated to uniform fresh positions and their new public parts are independent of the old ones (Theorem 4.3), so the map from vacated to filled positions is uniform in the adversary's view. A filled position therefore receives the average of the posterior mass on the vacated positions, at most $\max \cdot m/m = \max$, and a position outside the batch keeps its mass. After the leaf is in a batch, its mass is spread over $m \geq m_0$ positions, each at most $1/m_0$. □

A Residual reveals that someone in the Pool traded a net amount in a pair; it never reveals who, and a Crossing in which buyers and sellers cancel leaves no public trace beyond its Commit's slot and rewrite batches that look like decoys. Every deposit credit is itself a rewrite batch, so from the first Crossing that touches a leaf the rewrite pattern locates it only to within m_0 positions. The tracker's tools that read a wallet's history and react to its next move [7] have no wallet to read.

Countermeasures in the product

The three countermeasures are defaults in the SDK and the Coprocessor, not settings a trader has to find.

Denominations. The SDK splits deposits and Exits into $\mathcal{D}_a$ pieces by Definition 8.4; the sub-δ_a remainder stays in the trader's Ledger balance.

Exit delays. Each piece is scheduled by the enclave with an independent exponential delay of mean μ, giving (8.4) per denomination class.

Ring exits. An Exit can land in a Helius Ring instead of a wallet. The default Ring encrypts assets and amounts while wallets stay visible; Custom Rings can be configured anonymous, encrypting counterparties and balances, with mainnet release targeted for Q4 2026 [9]. An Exit through a Ring adds a second, independent mixing stage, and the two stages compose.

Lemma 8.8 (Composition). Let a funds path pass from a deposit wallet w through a Privium Exit into a Ring and out to a wallet w', and let the observations of the two stages be independent given the intermediate Ring deposit Z. If the best success probability of linking w to its Pool Exit is ε_P and that of linking a Ring deposit to its Ring withdrawal is ε_R, then the probability of linking w to w' is at most $\min(\varepsilon_P, \varepsilon_R)$.

Proof. Give the adversary, in addition to its view, the identity of Z's Ring withdrawal. This can only raise its success. Given that, the Ring stage is fully revealed and linking w to w' is exactly linking w to Z, which succeeds with probability at most ε_P. Symmetrically, revealing the link from w to Z reduces the task to the Ring stage, bounded by ε_R. Each bound holds for the original adversary, which knows less. □

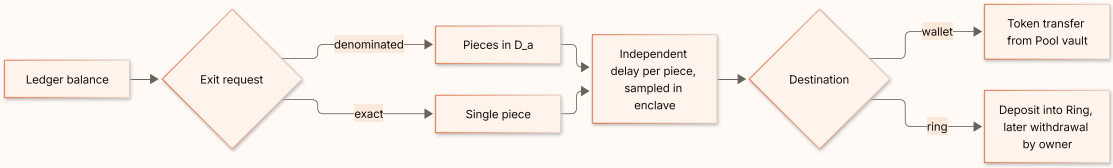
The listing shows the SDK's Exit with the defaults written out, and the denomination split.

```ts
export interface ExitOptions {
  denominate?: boolean;              // default true: split into D_a pieces
  delay?: "default" | "none";        // default: enclave samples D ~ Exp(mu) per piece
  via?: "wallet" | "ring";           // ring: land in a Helius Ring
}

export function denominate(amount: bigint, unit: bigint, J: number): bigint[] {
  let units = amount / unit;
  const pieces: bigint[] = [];
  for (let j = 0; j < J && units > 0n; j++, units >>= 1n) {
    if (units & 1n) pieces.push(unit << BigInt(j));
  }
  if (units > 0n) throw new Error("amount exceeds largest denomination run; split the Exit");
  return pieces;
}

export async function exit(
  client: PriviumClient,
  asset: PublicKey,
  amount: bigint,
  to: PublicKey,
  opts: ExitOptions = {},
): Promise<ExitTicket> {
  const unit = client.denominationUnit(asset);
  const pieces = opts.denominate === false ? [amount] : denominate(amount, unit, client.params.J);
  const request = {
    asset, to, pieces,
    delay: opts.delay ?? "default",
    via: opts.via ?? "wallet",
    nonce: client.nextNonce(),
  };
  const sealed = await client.sealToCoprocessor(request); // HPKE to the attested key
  return client.relay.submit(sealed);
}
```

The request travels like an Intent: sealed to the Coprocessor and relayed, so neither the destination nor the split is visible until each piece executes. The ticket lets the SDK track pieces as they land.



The unlinkability game

The statistical bounds measure what $\mathcal{L}$ reveals. What remains is to prove that the rest of Privium's transcript, every ciphertext, signature and root, reveals nothing more. We state this as an indistinguishability game in which the adversary must tell which of two accounts it chose funded an Exit.

Definition 8.5 (Game $\mathsf{Exp}^{\mathrm{unl}}_{\mathcal{A}}(\lambda)$).

1. *Setup.* The challenger runs Privium's setup: the Coprocessor generates (sk_C, pk_C) inside the enclave and registers it (Chapter 5); the Council runs key generation (Chapter 6). The adversary receives all public keys and the key shares of any $t - 1$ Council members of its choice.
2. *Oracles.* The adversary may open accounts, deposit, submit Intents and request Exits for any account, advance slots (which runs Crossings and Commits), and corrupt any account, receiving its keys and Viewing key. It sees every transaction, every ciphertext the Commits publish, and every Commit.
3. *Challenge.* The adversary names two honest accounts u_0, u_1 and an Exit request (asset, amount, destination, options). The challenger draws $b \in \{0, 1\}$ and executes the Exit from u_b.
4. *Admissibility.* The challenger runs every subsequent query in both worlds, $b = 0$ and $b = 1$, and answers it only if the leakage $\mathcal{L}$ it produces is identical in both; otherwise the query is refused. u_0, u_1 may not be corrupted, and Receipts and Viewing-key disclosures of u_0, u_1 are not available.
5. *Output.* The adversary outputs b'. Its advantage is $\mathsf{Adv}^{\mathrm{unl}}_{\mathcal{A}} = |2\Pr[b' = b] - 1|$.

Admissibility rules out the trivial wins: if u_0 could not afford the Exit, or a later Intent of u_0 would be funded in one world and not the other, the difference would appear in $\mathcal{L}$, and that is the leakage the statistical sections measure, not a break of the cryptography.

Theorem 8.9 (Unlinkability). Let the Coprocessor be realized by an enclave whose integrity and confidentiality hold except with advantage $\varepsilon_{\mathrm{TEE}}$ (Chapter 5), let the Intent encryption be IND-CCA2 with advantage $\varepsilon_{\mathrm{HPKE}}$, let the Ledger ciphertexts be IND-CPA with advantage $\varepsilon_{\mathrm{AE}}$ under nonce discipline (Chapters 3 and 4), let Commits follow the rewrite discipline, and let the adversary hold fewer than t Council shares. For any adversary making at most q_I Intent and Exit requests and observing at most q_L leaf ciphertexts,

$$\mathsf{Adv}^{\mathrm{unl}}_{\mathcal{A}}(\lambda) \;\le\; 2\big(\varepsilon_{\mathrm{TEE}} + q_I\,\varepsilon_{\mathrm{HPKE}} + q_L\,\varepsilon_{\mathrm{AE}}\big) + \mathsf{negl}(\lambda). \tag{8.5}$$

Proof. We move from the real game with $b = 0$ to a game whose view is a function of $\mathcal{L}$ alone, then back to $b = 1$ by the same steps in reverse. Each hop changes the adversary's output distribution by at most the stated term.

Hybrid 0. The real game with $b = 0$.

Hybrid 1. Replace the Coprocessor by an ideal functionality that receives plaintext Intents and Exit requests over a private channel, computes Crossings and Exits by the specification of Chapter 7, and outputs only signed Residual orders, Exit transfers and Commits. The adversary's view changes by at most $\varepsilon_{\mathrm{TEE}}$, by the integrity and confidentiality of the enclave: the enclave's outputs are the specified function of its inputs, and its memory and sealed keys are not observable.

Hybrid 2. Replace every Intent and Exit ciphertext from an honest account by an encryption of the all-zero message of the same padded length under ek_C, while the functionality still receives the true plaintexts. A distinguisher between consecutive sub-hybrids yields an IND-CCA2 adversary against the Intent encryption: the reduction answers decryption queries for adversarial ciphertexts through its decryption oracle, which is exactly what the adversary's own Intents require, and embeds the challenge ciphertext at one position. A hybrid over q_I positions costs $q_I\,\varepsilon_{\mathrm{HPKE}}$.

Hybrid 3. Replace every honest account's leaf ciphertext in every Commit by an encryption of zero under fresh randomness. Leaf keys are held inside the ideal functionality, so a distinguisher yields an IND-CPA adversary against the Ledger encryption; nonce discipline ensures each encryption uses a fresh nonce, the condition under which AES-256-GCM's confidentiality bound applies. A hybrid over q_L ciphertexts costs $q_L\,\varepsilon_{\mathrm{AE}}$. The rewrite pattern is an element of $\mathcal{L}$ under the rewrite discipline, and the leaves are now independent of balances.

Hybrid 4. Replace the adversary's $t-1$ Council shares by uniformly random field elements. With fewer than t shares of a Shamir sharing, the shares are uniformly distributed and independent of the secret (Chapter 6), so this change is perfect. The Council's public outputs in the execution, which arise only in an Escape, are simulatable from $\mathcal{L}$ and the decryption proofs of Chapter 6, whose zero-knowledge simulators are perfect or statistical; we absorb this in $\mathsf{negl}(\lambda)$.

In Hybrid 4 every component of the view is either an element of $\mathcal{L}$, a ciphertext of a fixed message, a signature by the ideal functionality on a message determined by $\mathcal{L}$ (Residual orders and Commit roots over zero-ciphertexts), or uniform randomness. The view is therefore a function of $\mathcal{L}$ and independent randomness. By admissibility, $\mathcal{L}$ is identical for $b = 0$ and $b = 1$, so Hybrid 4 is identically distributed for both values of b. Reversing Hybrids 3 through 0 with $b = 1$ gives the factor 2 in (8.5). □

Theorem 8.9 is the formal statement of what the earlier sections assumed: against $\mathcal{A}_{\mathrm{chain}}$, $\mathcal{A}_{\mathrm{track}}$ and $\mathcal{A}_{\mathrm{time}}$, Privium leaks exactly $\mathcal{L}$, and every quantitative bound in this chapter is a bound on what $\mathcal{L}$ says. Corruption of accounts in $\mathcal{C}$ is already in the game, which is why the anonymity set of Definition 8.2 counts honest accounts only.

Corollary 8.10 (Unlinkability with countermeasures). Under the assumptions of Theorem 8.9, for a denominated, delayed Exit piece of denomination class j, any adversary's probability of identifying its source deposit is at most

$$\frac{6}{\lambda_j \mu} + \frac{1}{m_0} + 2\big(\varepsilon_{\mathrm{TEE}} + q_I\,\varepsilon_{\mathrm{HPKE}} + q_L\,\varepsilon_{\mathrm{AE}}\big) + \mathsf{negl}(\lambda), \tag{8.6}$$

and, for a Ring exit, at most the minimum of this quantity and the Ring's own linking probability.

Proof. By Theorem 8.9 the adversary's view is, up to the cryptographic terms, a function of $\mathcal{L}$. Within $\mathcal{L}$ the piece's amount identifies only its class (Theorem 8.4), and within the class Corollary 8.6 bounds timing linkage by $6/(\lambda_j \mu)$; Proposition 8.7 (a) shows the Residuals add no information distinguishing honest accounts with equal public records. The rewrite pattern helps only if the adversary follows the piece's leaf from the batch that credited the deposit to the batch that debits the Exit, which by Proposition 8.7 (b) it does with probability at most $1/m_0$; a union bound adds that term. Lemma 8.8 gives the Ring statement. □

Example

This is an example with illustrative inputs; none of its figures are measurements of Privium or of Solana.

Take one denomination class of USDC in which honest deposits arrive at $\lambda_j = 0.05$ per slot, and the recommended mean delay $\mu = 9{,}000$ slots. Then $\lambda_j \mu = 450$, and (8.4) gives a timing adversary at most $6/450 \approx 1.3\%$ chance of naming the source deposit

of a piece, or at least $\log_2 75 \approx 6.2$ bits of average min-entropy; the rewrite term of (8.6) adds at most $1/64 \approx 1.6\%$ at $m_0 = 64$. Doubling either the deposit rate or the mean delay adds one bit.

For contrast, suppose the same trader exits an exact, unique amount with no delay, one slot after depositing it. The amount alone singles out the deposit, so the posterior has min-entropy zero, and no Pool size helps. The same trader with denominations and the default delay receives (8.6), and routing the Exit through an anonymous Ring caps the link by the Ring's own bound as well (Lemma 8.8). The difference between those two outcomes is a pair of defaults the trader does not have to set.

References

1. A. Serjantov, G. Danezis, "Towards an Information Theoretic Metric for Anonymity", Privacy Enhancing Technologies (PET), 2002. https://doi.org/10.1007/3-540-36467-6_4
2. C. Díaz, S. Seys, J. Claessens, B. Preneel, "Towards Measuring Anonymity", Privacy Enhancing Technologies (PET), 2002. https://doi.org/10.1007/3-540-36467-6_5
3. G. Smith, "On the Foundations of Quantitative Information Flow", FoSSaCS, 2009. https://doi.org/10.1007/978-3-642-00596-1_21
4. G. Kappos, H. Yousaf, M. Maller, S. Meiklejohn, "An Empirical Analysis of Anonymity in Zcash", USENIX Security Symposium, 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/kappos
5. Z. Wang, S. Chaliasos, K. Qin, L. Zhou, L. Gao, P. Berrang, B. Livshits, A. Gervais, "On How Zero-Knowledge Proof Blockchain Mixers Improve, and Worsen User Privacy", ACM Web Conference (WWW), 2023. https://arxiv.org/abs/2201.09035
6. S. Meiklejohn et al., "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names", ACM Internet Measurement Conference, 2013. https://doi.org/10.1145/2504730.2504747
7. "Photon vs Axiom: Solana Memecoin Terminal Comparison (2026)", MadeOnSol, 2026. https://madeonsol.com/blog/photon-vs-axiom-solana-trading-terminal
8. J. D. C. Little, "A Proof for the Queuing Formula: L = λW", Operations Research 9(3), 1961. https://doi.org/10.1287/opre.9.3.383
9. "Solana Privacy Rings", Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings
10. N. Lokhande et al., "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504

Chapter 9

Solana program engineering

The Pool is one Anchor program that custodies every deposited token, accepts each Commit, routes each Residual through Jupiter and pays every Exit and Escape. This chapter specifies it at audit depth: account layouts and seeds, each instruction with the accounts it locks and the checks it runs, the compute and byte budgets that set how much one Commit carries, and the write-lock design that keeps deposits, Intents and Commits parallel under Sealevel. The Ledger semantics are in Chapter 4, enclave registration in Chapter 5, the Escape protocol in Chapter 6 and the clearing math in Chapter 7. Solana constants are cited to current Solana and Agave sources; figures derived for Privium are marked "est.".

Program architecture

One program owns four state families: the Pool (`PoolConfig`, one `AssetState` per mint, vault and deposit queue shards), the Ledger (`LedgerRoot`, one `CrossingState` per open Crossing), the `Registry` of admitted measurements and live enclave keys with their validity intervals (Chapter 5), and the `CouncilConfig` holding the Council key PC, the n member verification keys S_j and the threshold t (Chapter 6). Splitting them across programs would put a CPI on the Commit path for every cross-family read, and a CPI costs 1,000 CU plus 1 CU per 250 serialized bytes [1] and spends a level of a limited call depth (see the Jupiter section).

Every vault is a token account owned by the Pool authority, a program derived address with no private key. The only code that can sign for it is `invoke_signed` inside this program, and every such path is gated by an enclave signature, a Council threshold signature or a Merkle proof against a committed root.

Accounts, PDAs and seeds

Fixed-layout state uses Anchor zero-copy accounts (`#[account(zero_copy)]`, `repr(C)`), so handlers read fields in place instead of decoding whole accounts. Sizes include the 8-byte discriminator.

Account	Seeds	Bytes	Count
`PoolConfig`	`["pool"]`	160	1
Pool authority	`["authority"]`	signer only	1
`LedgerRoot`	`["ledger"]`	2,696	1
`Registry`	`["registry"]`	656	1
`CouncilConfig`	`["council"]`	560	1
`AssetState`	`["asset", mint]`	200	per admitted mint
Vault shard 0	associated token account of the authority	165 Token, 170+ Token-2022 [16]	per mint
Vault shard $j \geq 1$	`["vault", mint, j]`	as above	$S - 1$ per mint
`DepositQueue`	`["deposits", mint, j]`	61,504	S per mint
`CrossingState`	`["crossing", k]`	736	per open Crossing
`ExitRecord` / `EscapeRecord`	`["exit", k, index]` / `["escaped", leaf, mint]`	9	per payout
`EscapeState`	`["escape"]`	913	1, created by `trigger_escape`
`EscapeRequest`	`["escape-req", leaf, mint]`	216	per Escape request

The shard count is $S = 2$ (parameter): two shards double each asset's write-lock headroom (Equation 9.4) while a Commit stays at five accounts per asset (Equation 9.3). Shard 0 is the authority's associated token account, so Jupiter's router can use it as swap source and destination directly.

```
#[account(zero_copy)]
#[repr(C)]
pub struct LedgerRoot {
    pub k: u64,                  // last committed Crossing
    pub last_commit_slot: u64,
    pub escape_from_slot: u64,   // 0 while commits are live
    pub root: [u8; 32],          // p^(k)
    pub da_commitment: [u8; 32], // hash of the Crossing's ciphertext diff
    pub mids_hash: [u8; 32],
    pub exit_ring_head: u32,
    pub registry_epoch: u32,
    pub exit_roots: [ExitRoot; 64], // { k: u64, root: [u8; 32] }
}

#[account(zero_copy)]
#[repr(C)]
pub struct AssetState {
    pub mint: Pubkey,
    pub token_program: Pubkey,
    pub oracle_feed: [u8; 32],   // Pyth feed id; zero when the pair always routes
    pub ledger_total: u64,       // T_a, attested at the last Commit
    pub pending_exits: u64,      // X_a, committed, not yet released
    pub gamma: [u8; 32],         // Gamma_a, sum of asset-a leaf commitments (Equation 4.7)
    pub consumed: [u64; 4],      // deposits credited per shard
    pub decimals: u8,
    pub bump: u8,
    pub flags: u8,               // admitted Token-2022 extensions
    pub _pad: [u8; 5],
}
```

`PoolConfig` holds the bumps, S, the batch interval Δ (Chapter 7), the liveness window W (Chapter 6), the Crossing lease (Chapter 10), the fee rate ϕ and the fee recipient, the token's launch wallet, whose associated token account in each asset receives every Commit's fees (Chapter 12). Only the admin key the configuration names changes the rate or the recipient. `LedgerRoot` keeps the last 64 exits roots (parameter), so an Exit proves inclusion against the Crossing that created it without one account per Crossing; an Exit whose root leaves the ring unreleased is placed again in a later exits root, which the enclave knows from the on-chain `ExitRecord` accounts.

A `DepositQueue` is a ring of 512 entries (parameter, several Crossings of deposits at the Equation 9.4 ceiling) of 120 bytes each: amount, cumulative amount, slot, the source token account (where an Escape refunds it), the hash of the deposit's encrypted note, and the running chain hash, for $8 + 56 + 512 \cdot 120 = 61{,}504$ bytes. An account may grow by at most 10,240 bytes inside one CPI (`MAX_PERMITTED_DATA_INCREASE` [1]), so Anchor's `init`, which allocates through a CPI to the System program, cannot create it. The queue is allocated by a top-level `CreateAccount` in the same transaction, within the 10 MiB account ceiling [1], and initialized under Anchor's `zero` constraint.

Rent for d data bytes is $(128 + d)\varpi$ lamports, with ϖ moving from 6,960 to 696 lamports per byte in the five steps of SIMD-0437 [13]. At $\varpi = 6{,}960$ a queue holds $61{,}632 \cdot 6{,}960 = 428{,}958{,}720$ lamports (est.), a tenth of that at the final step, all refundable on close.

Instructions

Accounts are W (writable) or R (read-only).

Instruction	Authorized by	Accounts	Checks
`admit_asset`	governance	W `AssetState`, vault shards; R mint	Extension set admitted (Token-2022 section)
`register_enclave`	attestation	W `Registry`	Quote and report-data binding (Chapter 5)
`rotate_council`	Council t-of-n	W `CouncilConfig`	Threshold signatures on the new set (Chapter 6)
`deposit`	trader	W vault j, queue j, source; R `AssetState`, `LedgerRoot`	No Escape; queue not full; credit = vault delta
`post_intent`	any fee payer	W fee payer only	Blob length
`open_crossing`	enclave	W `CrossingState`; R `LedgerRoot`, `Registry`, per mid an `AssetState` and its Pyth price account	$k = \texttt{LedgerRoot.k} + 1$, at least Δ slots after `last_commit_slot`; sponsored price accounts; validity recorded
`route_residual`	enclave	W `CrossingState`, vault 0 of two mints, route accounts; R `AssetState` ×2	Route hash; deadline; spent ≤ signed input; Crossing's total spent of the input ≤ its T_a; received ≥ signed minimum
`log_diff`	relayer named at `open_crossing`	W `CrossingState`	Next chunk index; accumulator (4.10) not sealed
`commit`	enclave	W `LedgerRoot`, `AssetState` ×A, `CrossingState`, vault 0 ×A, fee recipient account ×A; R other shards	Order, prior root, opener, mids, fills, deposits, data availability; pays ΔF_a to the fee recipient; conservation (Equations 9.2 and 4.7)
`release_exit`	anyone	W `AssetState`, vault, recipient, `ExitRecord`; R `LedgerRoot`	Inclusion in a ring root; first release
`set_fee_rate`, `set_fee_recipient`	admin	W `PoolConfig`	Admin signature; rate at most 10,000 bps
`trigger_escape`	anyone	W `LedgerRoot`, `EscapeState`; R `CrossingState`	Slot $>$ $\texttt{last_commit_slot} + W$; records the open Crossing's fills
`refund_deposit`	anyone	W queue j, vault j, source	Escape active; entry past the stored cut; first refund
`escape_request`	owner	W `EscapeRequest`; R `LedgerRoot`	Leaf in frozen root; owner tag opens to (Y_u, ς_u)
`escape_share`	Council member	W `EscapeRequest`; R `CouncilConfig`	Share proof (6.7) against S_j; combine at t
`escape_claim`	owner	W `EscapeRequest`, vaults, recipients, `EscapeRecord`; R `AssetState`, `EscapeState`	Claim proof under Y_u; first claim; settlement rule (6.10)

`post_intent` is the on-chain Intent path and it writes nothing but the fee payer. The Intent's HPKE ciphertext travels as instruction data, the program checks its length, and the indexer reads it from the transaction. Transactions with different fee payers share no writable account, so Sealevel runs them all in parallel.

`open_crossing` is the Crossing's lease and price anchor. It records the opening enclave key (only that key may route and commit k until the lease passes; Chapter 10) and takes one `AssetState` and one Pyth price account per mid. The price account must be the one Pyth sponsors and keeps current for the asset's feed [23]: the program derives the push oracle's address from

the seeds (shard 0 as a little-endian `u16`, `oracle_feed`) [28] and requires that exact account, owned by the Pyth receiver and holding a `PriceUpdateV2` at verification level `Full` whose feed id equals `oracle_feed` [29], so a relayer can substitute neither another account nor another feed (`Price`). It then sets each price's validity against its own `Clock`: positive, published no more than `max_price_age` seconds before it and no more than 1 second after, and with a confidence within κ of the price, the check of Equation 7.3 per feed. A price that fails is recorded as invalid rather than failing the open. The program stores price, confidence, exponent, publish time and validity per feed and their hash h_m, so the mids $m_\pi^{(k)}$ the enclave crosses at are the ones the chain verified, and `route_residual` refuses a swap in a mint whose recorded mid is invalid.

Deposit

```
pub fn deposit(ctx: Context<Deposit>, shard: u8, amount: u64, note: Vec<u8>) → Result<()> {
    require!(note.len() ≤ MAX_NOTE, PriviumError::NoteTooLong);
    require!(ctx.accounts.ledger.load()?.escape_from_slot == 0, PriviumError::EscapeActive);
    let asset = ctx.accounts.asset.load()?;
    let mut q = ctx.accounts.queue.load_mut()?;
    require!(q.head - asset.consumed[shard as usize] < QUEUE_CAP, PriviumError::QueueFull);

    let before = ctx.accounts.vault.amount;
    token_interface::transfer_checked(ctx.accounts.transfer_ctx(), amount, asset.decimals)?;
    ctx.accounts.vault.reload()?;
    let credited = ctx.accounts.vault.amount.checked_sub(before).ok_or(PriviumError::Math)?;

    let slot = Clock::get()?.slot;
    let note_hash = hashv(&[b"privium/note/v1", &note]).to_bytes();
    q.cum_amount = q.cum_amount.checked_add(credited).ok_or(PriviumError::Math)?;
    q.chain_hash = hashv(&[&q.chain_hash, &credited.to_le_bytes(), &slot.to_le_bytes(), &note_hash])
        .to_bytes();
    let i = (q.head % QUEUE_CAP) as usize;
    q.entries[i] = DepositEntry {
        amount: credited, cum_amount: q.cum_amount, slot,
        source: ctx.accounts.source.key(), note_hash, chain_hash: q.chain_hash,
    };
    q.head += 1;
    Ok(())
}
```

The credit is the measured vault delta, never the argument: under a transfer fee the vault receives less than was sent [17]. The note, which tells the Coprocessor which Ledger account to credit (Chapter 4), stays in transaction data and enters the chain hash through its digest.

Commit

The Commit has no payload of its own. Its message M_k of Equation 4.6 sits once, inside the Ed25519 precompile instruction before it, in this byte layout:

$$M_k = \texttt{privium/commit/v1} \,\|\, \text{program id} \,\|\, e \,\|\, k \,\|\, \rho^{(k-1)} \,\|\, \rho^{(k)} \,\|\, \delta^{(k)} \,\|\, E^{(k)} \,\|\, n_E \,\|\, h_m \,\|\, h_F \,\|\, h_D \,\|\, \big(\text{mint}_a, T_a, \Delta X_a, \Delta F_a, c_{a,0}, \ldots, c_{a,J-1}, \Delta\Gamma_a, R_a\big)_{a \in \mathcal{A}_k} \tag{9.1}$$

with e the signing key's epoch, $\delta^{(k)}$ the data availability accumulator (4.10) over the Crossing's ciphertext diff, $E^{(k)}$ and n_E the exits root and count, h_m and h_F the hashes of the mids and Residual fills in `CrossingState`, h_D the digest of the deposit chain hashes at the consumed positions, $\mathcal{A}_k$ the touched assets, $T_a = \sum_u b_{u,a}^{(k)}$ the attested Ledger total, ΔX_a the exits the Crossing adds, ΔF_a the fees it pays, $c_{a,j}$ the deposits consumed from shard j, and $(\Delta\Gamma_a, R_a)$ the commitment delta and aggregate opening of Equation 4.5. After applying the deltas, the program checks for each $a \in \mathcal{A}_k$:

$$T_a + X_a + \sum_{j<S} \left(\Sigma_{a,j}^{\text{head}} - \Sigma_{a,j}^{(c)}\right) = \sum_{j<S} V_{a,j} = V_a^{(k)} \tag{9.2}$$

evaluated after the Commit has paid ΔF_a from vault shard 0 to the fee recipient's token account, where $\Sigma_{a,j}^{\text{head}}$ is shard j's cumulative deposited amount, $\Sigma_{a,j}^{(c)}$ the cumulative amount at the consumed entry, and $V_{a,j}$ the shard balance. This is the on-chain form of the conservation theorem of Chapter 4, exact in base units: at every committed root, every vault token is a committed balance, a committed Exit awaiting release or a deposit not yet credited; no fee stays in a vault past the Commit that charges it. Between Commits the open Crossing's recorded fills also move the vaults; its Commit absorbs them into T_a, so none is outstanding at a committed root, and if commits stop with fills recorded the Escape settles them (Chapter 6). Releases landing between the enclave's snapshot and the Commit keep (9.2) exact, because the enclave signs deltas and `release_exit` decrements X_a together with the vault; late deposits enter both sides through Σ^{head}. The commitment half of the check, Equation 4.7, runs in the same pass.

```rust
pub fn commit<'info>(ctx: Context<'_, '_, 'info, 'info, Commit<'info>>, k: u64) -> Result<()> {
    let slot = Clock::get()?.slot;
    let msg = enclave_message(&ctx.accounts.instructions, &ctx.accounts.registry.load()?, slot)?;
    let c = CommitMsg::parse(&msg)?;
    let mut ledger = ctx.accounts.ledger.load_mut()?;
    require!(ledger.escape_from_slot == 0, PriviumError::EscapeActive);
    require!(c.program_id == crate::ID && c.k == k && k == ledger.k + 1, PriviumError::OutOfOrder);
    require!(c.prev_root == ledger.root, PriviumError::StaleRoot);

    let mut crossing = ctx.accounts.crossing.load_mut()?;
    require!(crossing.k == k && c.signer == crossing.opener, PriviumError::NotOpener);
    require!(c.mids_hash == crossing.mids_hash, PriviumError::MidsMismatch);
    require!(c.fills_hash == crossing.fills_hash(), PriviumError::FillsMismatch);
    require!(c.da == crossing.da_digest && !crossing.da_sealed, PriviumError::DaMismatch);
    crossing.da_sealed = true;

    let pool = ctx.accounts.pool.load()?;
    let per_asset = 4 + 2 * SHARDS; // state, mint, vaults, queues, fee recipient, token program
    require!(ctx.remaining_accounts.len() == per_asset * c.assets.len(), PriviumError::Accounts);
    let mut chain = Vec::with_capacity(SHARDS * c.assets.len());
    let mut acc = ConserveAcc::new(&hash(&msg));
    for (a, group) in c.assets.iter().zip(ctx.remaining_accounts.chunks_exact(per_asset)) {
        conserve(a, group, &pool, &ctx.accounts.authority, &mut chain, &mut acc)?; // Chapter 4
    }
    acc.finish()?;                                          // Equation 4.7, one MSM
    let slices: Vec<&[u8]> = chain.iter().map(|h: &[u8; 32]| h.as_slice()).collect();
    require!(hashv(&slices).to_bytes() == c.deposits_digest, PriviumError::DepositsMismatch);

    ledger.k = k;
    ledger.root = c.root;
    ledger.da_commitment = c.da;
    ledger.mids_hash = c.mids_hash;
    ledger.last_commit_slot = slot;
    let h = ledger.exit_ring_head as usize % 64;
    ledger.exit_roots[h] = ExitRoot { k, root: c.exits_root };
    ledger.exit_ring_head = ledger.exit_ring_head.wrapping_add(1);
    Ok(())
}
```

`conserve`, listed in Chapter 4, matches the `AssetState` mint to the message, pays ΔF_a from vault shard 0 to the fee recipient's associated token account under the Pool authority's seeds, checks each vault shard is a token account of that mint owned by the authority, reads each queue entry at index $c_{a,j} - 1$ for its chain hash and $\Sigma^{(c)}_{a,j}$, applies the deltas, evaluates Equation 9.2 and stages the asset's terms of Equation 4.7. It derives one address per asset, the fee recipient's associated token account from the recipient, the token program and the mint, and none of the Pool's own: each Pool account was created by this program at its canonical address, and the owner check, discriminator and stored mint and shard fields identify it uniquely. At 1,500 CU per `create_program_address` [1], skipping five derivations per asset saves 45,000 CU (est.) on a six-asset Commit.

The set $\mathcal{A}_k$ may be empty. A heartbeat Commit (Chapter 10) carries no asset entry and takes no remaining account: the program checks its order, prior root, opener, mids, fills and data availability as for any Commit, Equation 4.7 is the empty sum, and h_D covers no cut. It advances the index, re-anchors the root and stores its slot, from which the liveness window of Chapter 6 runs again.

The digest h_D binds the deposits the enclave credited to the chain the program wrote. The enclave reads deposits through an untrusted RPC host (Chapter 10); a fabricated deposit yields a chain hash that matches no on-chain entry, and the Commit fails.

Exit and Escape

An Exit leaf is $\mathsf{H}(\texttt{0x02} \,\|\, k \,\|\, \text{index} \,\|\, \text{mint} \,\|\, \text{recipient} \,\|\, \text{amount} \,\|\, \text{release slot})$ under the root $E^{(k)}$, so the delay the enclave samples for each piece (Chapter 8) is enforced on chain. `release_exit` is permissionless: it verifies the path against the ring entry for k, requires the current slot to be at or past the release slot, creates the `ExitRecord` (whose existence blocks a second release), decrements X_a and pays the recipient, which is any address the trader named, a Helius Ring included (Chapter 8).

`trigger_escape` sets `escape_from_slot` once W slots pass without a Commit and writes `EscapeState`: the frozen index $k^\star$ and, when Crossing $k^\star + 1$ is open, the fills its `CrossingState` recorded, leg by leg (input mint, output mint, spent, got). Commits, opens, routes and deposits then fail. `refund_deposit` returns each queue entry past the stored cut to the source the entry records, and `release_exit` keeps paying committed Exits. An owner opens an `EscapeRequest` with the leaf, its Merkle path to the frozen root and the opening (Y_u, ς_u) of its owner tag, which the program checks against the leaf's version. Council members post re-encryption shares with `escape_share`, each verified against the member's key S_j in `CouncilConfig`, and the t-th valid share stores the combined pair. `escape_claim` verifies the owner's proof for the amount, creates the `EscapeRecord` for (leaf, mint), whose existence blocks a second claim, and pays by the settlement rule (6.10): in full in the leaf's asset when

the open Crossing spent none of it, otherwise partly in the output assets of the legs that did. Chapter 6 gives the protocol and its safety theorem.

Signature verification by instruction introspection

Precompiles run as native code outside the sBPF VM and cannot be invoked by CPI [3]. A program learns that a signature was checked by reading the transaction's top-level instructions through the Instructions sysvar, which exposes top-level instructions only [4]. The Ed25519 program (`Ed25519SigVerify111111111111111111111111111`) takes a signature count and, per signature, seven `u16` offsets and instruction indices for signature, key and message, where `u16::MAX` means the precompile instruction itself; the transaction fails if any signature is invalid [3].

The flexibility is the classic hole: a precompile may verify a signature over bytes from some other instruction, so a program that checks only that an Ed25519 instruction names the right key can be fed a valid signature over an unrelated message. The Pool accepts exactly one shape:

```
pub fn enclave_message(ix_sysvar: &AccountInfo, reg: &Registry, slot: u64) → Result<Vec<u8>> {
    let here = load_current_index_checked(ix_sysvar)? as usize;
    require!(here > 0, PriviumError::MissingSig);
    let ix = load_instruction_at_checked(here - 1, ix_sysvar)?;
    require!(ix.program_id == ed25519_program::ID, PriviumError::MissingSig);
    let d = &ix.data;
    require!(d.len() ⩾ 112 && d[0] == 1, PriviumError::BadSigLayout); // one signature

    let f = |o: usize| u16::from_le_bytes([d[2 + o], d[3 + o]]) as usize;
    let this = u16::MAX as usize;
    require!(f(2) == this && f(6) == this && f(12) == this, PriviumError::BadSigLayout);
    require!(f(0) == 16 && f(4) == 80 && f(8) == 112, PriviumError::BadSigLayout);
    require!(112 + f(10) == d.len(), PriviumError::BadSigLayout);

    let pk: [u8; 32] = d[80..112].try_into().unwrap();
    let key = reg.find(&pk).ok_or(PriviumError::UnknownEnclave)?;
    require!(key.valid_from ⩽ slot && slot < key.valid_until, PriviumError::KeyExpired);
    Ok(d[112..].to_vec())
}
```

Fixed offsets (2-byte header, 14-byte offsets record, signature, key, message to the end) mean the program parses one known layout instead of attacker-chosen offsets, and pinning the precompile to index $here - 1$ gives each signed instruction its own. The domain tag and program id in Equation 9.1 stop a signature made for `route_residual` , `open_crossing` or another deployment from passing as a Commit. The same function with a t-signature layout and `CouncilConfig` keys serves `rotate_council` ; Escape shares and claims are sigma proofs that the program checks with the curve25519 syscalls instead.

Ed25519 carries the per-Commit signature because it is the cheapest precompile in the cost model: 2,280 CU per verification against 4,800 for secp256r1 and 6,690 for secp256k1 [10]. The secp256r1 program, which checks up to eight P-256 signatures per instruction with low-S enforced [3], is used at enclave registration for the attestation chain (Chapter 5).

The Residual route through Jupiter

The Coprocessor gets each Residual's route from Jupiter's Router path, `GET /swap/v2/build` , which returns raw Metis-routed instructions with no Jupiter swap fee, for integrators that compose or wrap the swap [21]. The program invokes the Jupiter aggregator (`JUP6LkbZbjS1jKKwapdHNy74zcZ3tLUZoi5QNyVTaV4`) with the route's accounts as remaining accounts, the pattern of Jupiter's CPI example [22], and adds three guards.

```rust
pub fn route_residual<'info>(
    ctx: Context<'_, '_, 'info, 'info, RouteResidual<'info>>, data: Vec<u8>,
) -> Result<()> {
    let slot = Clock::get()?.slot;
    let msg = enclave_message(&ctx.accounts.instructions, &ctx.accounts.registry.load()?, slot)?;
    let r = RouteMsg::parse(&msg)?;
    let mut crossing = ctx.accounts.crossing.load_mut()?;
    require!(r.k == crossing.k && r.signer == crossing.opener, PriviumError::OutOfOrder);
    require!(slot <= r.deadline_slot, PriviumError::ResidualExpired);

    let authority = ctx.accounts.authority.key();
    let mut metas = Vec::with_capacity(ctx.remaining_accounts.len());
    let mut keys: Vec<&[u8]> = vec![JUPITER_ID.as_ref(), &data];
    for a in ctx.remaining_accounts {
        let is_signer = *a.key == authority;
        metas.push(AccountMeta { pubkey: *a.key, is_signer, is_writable: a.is_writable });
        keys.push(a.key.as_ref());
    }
    require!(hashv(&keys).to_bytes() == r.route_hash, PriviumError::RouteMismatch);

    let (in0, out0) = (ctx.accounts.vault_in.amount, ctx.accounts.vault_out.amount);
    let seeds: &[&[u8]] = &[b"authority", &[ctx.accounts.pool.load()?.authority_bump]];
    let ix = Instruction { program_id: JUPITER_ID, accounts: metas, data };
    invoke_signed(&ix, ctx.remaining_accounts, &[seeds])?;

    ctx.accounts.vault_in.reload()?;
    ctx.accounts.vault_out.reload()?;
    let spent = in0.checked_sub(ctx.accounts.vault_in.amount).ok_or(PriviumError::Math)?;
    let got = ctx.accounts.vault_out.amount.checked_sub(out0).ok_or(PriviumError::Math)?;
    require!(spent <= r.in_amount && got >= r.min_out, PriviumError::Slippage);
    require!(crossing.spent_of(&r.in_mint) + spent <= ctx.accounts.asset_in.load()?.ledger_total,
        PriviumError::RouteCap);                              // routes spend only committed balances
    crossing.record_fill(r.pair, spent, got)
}
```

The route is bound to the enclave: it signs $\mathsf{H}(\text{Jupiter id} \,\|\, \text{data} \,\|\, \text{keys})$ with $(k, \pi, x_{\text{in}}, y_{\min})$ and the program recomputes the hash over what it invokes, so the relayer cannot substitute a route and the authority's signature reaches only accounts the enclave approved; the message is the `ResidualOrder` of Chapter 7 under the domain tag `privium/route/v1`. The outcome is measured: input spent must be at most x_{in} and output received at least $y_{\min}$, the bound Chapter 7 derives from the oracle mid. The fill is recorded in `CrossingState`, the enclave settles on recorded fills, and h_F in the Commit binds them, so the Ledger credits exactly what the market delivered. The routes of one Crossing never spend more of an asset than its committed total T_a, so committed Exits and uncredited deposits stay in the vault whatever the route does, which the Escape's solvency relies on (Chapter 6).

The authority enters the outer transaction as a non-signer; the program marks it signer in the CPI metadata and `invoke_signed` supplies its seeds. The `/build` request names the authority as `taker`, the relayer as `payer` so the PDA needs no SOL, output shard 0 as `destinationTokenAccount`, and `wrapAndUnwrapSol=false`, since vaults hold wrapped SOL as an ordinary mint [21].

The CPI runs the route one level deeper than a wallet swap. The maximum instruction stack depth is 5, height 1 being the top-level instruction [1][5]. Inside `route_residual`, Jupiter runs at height 2, each AMM at 3 and the Token program at 4, leaving one level for an AMM that makes a further CPI. SIMD-0268 raises the limit to 9 and is in development for Agave 4.4 [6]. A transaction also has 64 instructions in total, CPIs included (`MAX_INSTRUCTION_TRACE_LENGTH` [1]). The Coprocessor simulates each route before signing and rebuilds without the offending DEX when either limit is hit (Chapter 10).

Token and Token-2022 handling

Each vault uses the mint's own token program, recorded in `AssetState.token_program`, and every transfer goes through `token_interface::transfer_checked`, which serves both programs and is the form Token-2022 extensions require. On the Token program, p-token (SIMD-0266, live on mainnet) runs `TransferChecked` in 105 CU, down from 6,200 [15]. `admit_asset` admits a mint when each of its extensions has a treatment:

Extension	Treatment
`TransferFeeConfig`	Credit measured vault deltas on deposit and route [17]
`TransferHook`	Hook program on the admitted list; senders append the extra accounts every transfer of the mint requires [18]
`ConfidentialTransfer`	Vaults hold the public balance; private entry and exit happen in the Ring or confidential account around the Pool [19]
`InterestBearingConfig`, `ScaledUiAmount`	Raw base units; scaling is display only
`PermanentDelegate`, `NonTransferable`	Not admitted: Equation 9.2 needs vault balances that move only through Pool instructions

Confidential Balances run on the ZK ElGamal proof program, restored on mainnet in June 2026 [20]. The Pool verifies no ZK ElGamal proof itself: Ledger confidentiality rests on the enclave and the Council (Chapters 4 to 6), and vaults move only public amounts.

Compute budget

A transaction may request up to 1,400,000 CU; one that requests nothing gets 200,000 CU per non-builtin instruction [1]. The estimates below use published unit costs: 1,500 CU per `create_program_address`, 1,000 CU per CPI plus 1 CU per 250 bytes, `sol_sha256` at 85 CU plus, per slice, the larger of 10 CU and half the slice length, and for ristretto255 169 CU per point validation, 521 per addition and a multiscalar multiplication at 2,303 CU plus 788 per point beyond the first [1][11][12]. A Merkle node $\mathsf{H}(\mathrm{tag} \,\|\, L \,\|\, R)$ therefore costs $85 + 10 + 16 + 16 = 127$ CU. Anchor dispatch and account validation get an allowance of 5,000 CU per instruction (est.), replaced by simulated figures before every send (Chapter 10).

Instruction	Components	CU (est.)
`deposit`	allowance; 3 PDA checks 4,500; transfer CPI ≈ 1,110; hashes ≈ 320; writes ≈ 500	12,000
`post_intent`	allowance	5,000
`commit`, $A = 2$	measured on the built program, both fee transfers and the fee recipient's address derivations included	73,214 to 78,518, measured
`route_residual`	allowance; introspection ≈ 1,500; route hash ≈ 1,000; CPI ≈ 2,000; reloads ≈ 500	10,000 + route
`release_exit`, depth 12	allowance; path $12 \cdot 127$; record creation 2,500; transfer ≈ 1,110	10,200
`escape_share`	allowance; (6.7) as one 11-point MSM $2{,}303 + 10 \cdot 788$; 6 point validations; challenge ≈ 300; at the t-th share, (6.8) as two 5-point MSMs 10,910	16,500, or 27,400 at t
`escape_claim`	allowance; claim proof as one 6-point MSM $2{,}303 + 5 \cdot 788$; point work ≈ 1,500; record creation 2,500; transfer ≈ 1,110 per paid mint	16,400

The cost model adds 2,280 CU per Ed25519 verification [10], so a two-asset Commit, its fees paid, costs at most about 80,800 CU, near 5.8% of the transaction ceiling and below the 157,000 CU Encifher reports for a whole confidential swap [27]. It is cheap because the chain verifies and never recomputes: tree updates, netting and allocation happen once in the enclave, and the program checks one signature, one vault equation per asset and one multiscalar multiplication for all assets. The Jupiter route is the large item; the relayer simulates at 1,400,000 and sets 1.2 times the units consumed, as Jupiter prescribes for `/build` [21].

Transaction size and lookup tables

Legacy and v0 transactions hold 1,232 bytes. The v1 format, active on mainnet since epoch 1035 in September 2026, holds 4,096 bytes and inlines every address, with no lookup tables [2][7]. A lookup table stores up to 256 addresses that v0

references by 1-byte index, non-signers only [2]. Both formats lock at most 64 accounts; SIMD-0596, in development, raises v1 to 96 [1][8]. The Commit is laid out for v1.

The fixed part of a v1 Commit is 880 bytes (est.): 42 of prefix, header, config mask, blockhash and counts; ten addresses (fee payer, `PoolConfig`, `LedgerRoot`, `Registry`, `CrossingState`, Pool authority, Instructions sysvar, event authority, Ed25519 program, Pool program) at 320; 16 of config values; 8 of instruction headers; 112 of precompile header, signature and key plus a 294-byte fixed message; 24 of Commit indices and data; a 64-byte signature. Each asset adds eight addresses (its `AssetState`, mint, two vault shards, two deposit queues, the fee recipient's token account and the token program), 136 message bytes and eight indices, 400 bytes:

$$880 + 400A \le 4{,}096 \quad \text{and} \quad 10 + 8A \le 64 \implies A_{\max} = 6 \text{ (est.)} \tag{9.3}$$

The account limit binds first, at six assets, where a Commit is 3,280 bytes; the byte limit alone would allow eight, the most one Commit message carries. At the 96 accounts of SIMD-0596 the byte limit binds, at eight (est.). A Crossing touching more assets commits as consecutive Crossings, each closed under the pairs it settles (Chapter 7).

A v1 transaction must state its compute limit and loaded-accounts data size: left unset, each is zero and the transaction fails, and ComputeBudget instructions in v1 run as 150 CU no-ops [2]. The relayer writes both into the message config from a simulation, rounding the data size up to the next 32 KiB page, the cost model's unit [2].

Sealevel parallelism and write-lock design

Sealevel runs transactions in parallel when their write sets are disjoint, and Encifher shows that declared read and write sets order encrypted operations without a custom scheduler [27]. The Pool keeps its high-volume instructions on disjoint writable accounts.

Instruction	Writes	Reads	Serializes with
`deposit` (shard j)	vault j, queue j, source	`AssetState`, `LedgerRoot`	deposits on shard j; the Commit
`post_intent`	fee payer	none	nothing
`route_residual`	`CrossingState`, two vaults, AMM pools	`AssetState` ×2	swaps on the same pools
`log_diff`	`CrossingState`	none	the Crossing's other stages
`commit`	`LedgerRoot`, `AssetState` ×A, `CrossingState`, vault 0 and the fee recipient's account ×A	other shards of A assets	everything on those assets, once per Crossing
`release_exit`	`AssetState`, vault, recipient	`LedgerRoot`	releases of the same asset

The binding limit for a hot asset is per account: one writable account absorbs at most 12,000,000 CU per block, inside a block limit of 100,000,000 CU since SIMD-0286 activated in July 2026 [9][10]. With the deposit cost c_{dep} of the compute table,

$$N_{\text{dep}} \le S \cdot \frac{12{,}000{,}000}{c_{\text{dep}}} = 2 \cdot \frac{12{,}000{,}000}{12{,}000} = 2{,}000 \text{ per asset per block (est.)} \tag{9.4}$$

or 10,000 deposits per asset per second at the 200 ms slot time live on mainnet since SIMD-0525 [14] (est.); every other chapter converts slots to time with this figure. The SDK picks the shard uniformly at random, so load spreads without coordination. Releases of one asset serialize on `AssetState` at up to $12{,}000{,}000/10{,}200 \approx 1{,}176$ per block (est.). The Commit's read locks order it against deposits on its shards once per Crossing; deposits landing first are counted in Σ^{head} and credited at the next Crossing.

Upgrade authority policy

The Pool is deployed under the upgradeable loader: the upgrade authority can replace the bytecode from a buffer, and setting the authority to `None` makes the program immutable for good [24][25]. The authority is a multisig whose execution enforces a time lock longer than W plus the release crank's slots, so every pending upgrade is visible on chain for longer than any trader needs to Exit, or to Escape, under the code they deposited with. Each buffer's hash is reproducible from the tagged source with the Solana Verify CLI in Docker, and its verification record lives in a PDA anyone can recheck [26].

Layouts carry a `version` byte and reserved space so upgrades extend state without migration. Once the instruction set has run unchanged through a Council rotation and an enclave re-attestation (Chapters 5 and 6), the authority is removed with `solana program set-upgrade-authority <program-id> --final` [25]. From then on the enclave changes by re-attestation in the Registry and the Council by threshold rotation, with the program fixed.

References

1. "Constants Reference", Solana Foundation, 2026. https://solana.com/docs/core/constants-reference
2. "Versioned Transactions: legacy, v0 and v1", Solana Foundation, 2026. https://solana.com/docs/core/transactions/versioned-transactions
3. "Precompiled Programs", Solana Foundation, 2026. https://solana.com/docs/core/programs/precompiles
4. "Instruction Introspection", Solana Foundation, 2026. https://solana.com/docs/core/instructions/instruction-introspection
5. "CPI Execution and Privileges", Solana Foundation, 2026. https://solana.com/docs/core/cpi/cpi-execution
6. "Deeper CPI Nesting" (SIMD-0268), Solana Foundation, October 2026. https://solana.com/upgrades/cpi-nesting-depth
7. "Larger Transaction Sizes" (transaction v1), Solana Foundation, September 2026. https://solana.com/upgrades/larger-transaction-sizes
8. "More Accounts per Transaction" (SIMD-0596), Solana Foundation, October 2026. https://solana.com/upgrades/v1-account-limit
9. "100M CU Blocks" (SIMD-0286), Solana Foundation, July 2026. https://solana.com/upgrades/100m-cu-blocks
10. `cost-model/src/block_cost_limits.rs`, Agave v3.1.8, Anza. https://github.com/anza-xyz/agave/blob/v3.1.8/cost-model/src/block_cost_limits.rs
11. `program-runtime/src/execution_budget.rs`, Agave v3.1.8, Anza. https://github.com/anza-xyz/agave/blob/v3.1.8/program-runtime/src/execution_budget.rs
12. `syscalls/src/lib.rs` (hash syscall metering), Agave v3.1.8, Anza. https://github.com/anza-xyz/agave/blob/v3.1.8/syscalls/src/lib.rs
13. "Reduced Rent" (SIMD-0437), Solana Foundation, September 2026. https://solana.com/upgrades/reduced-rent
14. "Reduced Slot Times" (SIMD-0525), Solana Foundation, October 2026. https://solana.com/upgrades/reduced-slot-times
15. "Optimized Token Program" (p-token, SIMD-0266), Solana Foundation, 2026. https://solana.com/upgrades/p-token
16. "Create a Token Account", Solana Foundation, 2026. https://solana.com/docs/tokens/basics/create-token-account
17. "Transfer Fees", Solana Foundation, 2026. https://solana.com/docs/tokens/extensions/transfer-fees
18. "Transfer Hook Integration Guide", Solana Foundation, 2026. https://solana.com/docs/tokens/extensions/transfer-hook-integration
19. "Confidential Balances", Solana Foundation, 2026. https://solana.com/docs/tokens/extensions/confidential-transfer
20. "Confidential transfers return to Solana mainnet after year-long security pause", Solana Compass, June 18, 2026. https://solanacompass.com/news/confidential-transfers-return-to-solana-mainnet-after-year-long-security-pause
21. "Build" (Swap API V2 Router path), Jupiter, 2026. https://developers.jup.ag/docs/swap/build
22. `sol-swap-cpi`, Jupiter, GitHub. https://github.com/jup-ag/sol-swap-cpi
23. "Sponsored Feeds" (Solana), Pyth Network, 2026. https://docs.pyth.network/pro/sponsored-feeds#solana
24. "Program Deployment", Solana Foundation, 2026. https://solana.com/docs/core/programs/program-deployment
25. "Deploying Programs", Solana Foundation, 2026. https://solana.com/docs/programs/deploying
26. "Verified Builds", Solana Foundation, 2026. https://solana.com/docs/programs/verified-builds
27. N. Lokhande et al., "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
28. `programs/pyth-push-oracle/src/lib.rs` (price feed account seeds), Pyth Network, GitHub. https://github.com/pyth-network/pyth-crosschain/blob/main/target_chains/solana/programs/pyth-push-oracle/src/lib.rs
29. `pyth_solana_receiver_sdk/src/price_update.rs` (`PriceUpdateV2`, `VerificationLevel`), Pyth Network, GitHub. https://github.com/pyth-network/pyth-crosschain/blob/main/target_chains/solana/pyth_solana_receiver_sdk/src/price_update.rs

Chapter 10

The Coprocessor and the off-chain stack

The Coprocessor is the service that turns encrypted Intents into a committed Ledger root whenever there is something to settle, as often as once per batch interval. It runs as a Rust service inside a TEE, and around it sits an untrusted host stack: a relayer that lands transactions, a reader for Pyth's price accounts on Solana, clients for Jupiter's Price API and Router, a ciphertext store, an indexer, and the TypeScript SDK through which a trader places an order in one call. This chapter specifies that stack: how the service is cut between trusted and untrusted code, how one Crossing runs, how every input from the host is bound on chain so the enclave never has to trust it, how Commits land and recover, and how the enclave image is built so anyone can reproduce its measurement. Attestation and key custody are in Chapter 5, the Escape in Chapter 6, the Crossing math in Chapter 7 and the program in Chapter 9.

Service architecture

Encifher established the pattern Privium follows: on-chain programs hold handles to ciphertexts, ciphertexts live on a public data availability layer, and a coprocessor inside an enclave decrypts, computes on plaintext and commits attested, Merkle-anchored results, with decryption trust split across a threshold committee [1]. Privium specializes it to one workload, the Crossing, and draws the trust boundary tightly around the code that sees plaintext.

Component	Domain	Holds	Role
`intake`	enclave	HPKE private key	Opens Intents, checks signature, nonce and expiry
`engine`	enclave	plaintext Ledger	Nets, clears and settles a Crossing (Chapter 7)
`ledger`	enclave	Merkle tree over ciphertext leaves	Re-encrypts touched leaves, computes $\rho^{(k)}$
`signer`	enclave	sk_C	Signs `open_crossing`, `route_residual` and Commit messages
`relayer`	host	a fee-payer key holding SOL only	Builds, simulates, sends and confirms transactions
`oracle`	host	the Jupiter API key it shares with `router`	Reads Pyth's sponsored price accounts and Jupiter's reference prices
`router`	host	Jupiter API key	Requests `/swap/v2/build` routes
`store`	host	nothing secret	Posts ciphertext diffs as `log_diff` chunks; keeps Crossing journals and a diff mirror
`indexer`	host	nothing secret	Decodes program activity; serves leaves and proofs

The host components carry no authority over funds. Every instruction that moves Pool tokens needs an enclave signature (Chapter 9), and the relayer's key can only pay fees. The host can delay the enclave or feed it wrong data, so each input it supplies is bound to something the chain checks:

Input from the host	Binding the program checks
Deposit entries read over RPC	Digest h_D of the chain hashes at the consumed positions (Equation 9.1)
Ciphertext diff	Accumulator $\delta^{(k)}$ that `log_diff` builds on chain (Equation 4.10); the Commit must repeat it
Oracle prices	`open_crossing` derives each Pyth price account itself, verifies it and stores the prices' hash h_m; the Commit must repeat it
Jupiter route	The enclave signs the route hash and $y_{\min}$; the program measures the fill
Residual fills	Recorded on chain by `route_residual`; the Commit carries their hash h_F
Previous root and Crossing index	`commit` requires $\rho^{(k-1)}$ and k to match `LedgerRoot`

A host that lies produces a Commit the program rejects. The worst a host can do is withhold service, and the answer to that is a second, independently hosted enclave (see Deployment) and, past W slots, the Escape.

The workspace mirrors the boundary, so the enclave image links only crates that are meant to see plaintext:

```
[workspace]
resolver = "2"
members = [
  "crates/types",      # Intent, LedgerLeaf, CommitMsg, RouteMsg: shared wire formats
  "crates/engine",     # pure Crossing function, no I/O, no clock, no floats
  "crates/enclave",    # intake, ledger, signer, sealing; links engine
  "crates/relayer",    # host: transactions, retries, fee payer
  "crates/feeds",      # host: Pyth account reader, Jupiter clients
  "crates/store",      # host: DA writer and reader
  "crates/indexer",    # host: program decoder, leaf and proof API
]

[profile.release]
codegen-units = 1
lto = "fat"
panic = "abort"
```

Every dependency is pinned to an exact version in `Cargo.lock` and vendored, because the enclave's measurement is a hash of the image and an unpinned resolver makes two honest builds differ (see the last section).

The Crossing loop

A Crossing is five steps: collect, open, cross, route and commit. It goes on chain only when it has something to settle, and the Coprocessor otherwise sends a heartbeat Commit that carries no asset. The loop below is the enclave's view of it; each `await` on a host service crosses the boundary through a narrow, typed channel.

```rust
const SCAN_SLOTS: u64 = 25;       // resting orders against the live reference
const HEARTBEAT_SLOTS: u64 = 750; // the longest gap between Commits

pub async fn run(mut c: Coprocessor) -> Result<()> {
    let mut last_scan = 0;
    loop {
        let head = c.chain.ledger_root().await?;          // host read, checked at Commit
        let k = head.k + 1;
        let cutoff = c.chain.wait_slot(head.last_commit_slot + c.cfg.batch_interval).await?;
        let deposits = c.chain.deposits_since(&c.ledger.cursors()).await?;

        let mut due = c.intake.has_due_work(cutoff, &deposits); // no price needed
        if !due && cutoff ≥ last_scan + SCAN_SLOTS {
            last_scan = cutoff;
            let refs = c.oracle.reference(&c.intake.resting_assets()).await?; // Price API V3
            due = c.intake.resting_may_fill(cutoff, &refs);   // Equation 10.1
        }
        if !due && cutoff < head.last_commit_slot + HEARTBEAT_SLOTS {
            c.chain.wait_slot(cutoff + 2).await?;           // idle: nothing goes on chain
            continue;
        }

        let batch = c.intake.drain(k, cutoff);              // received before the cutoff
        c.store.put_journal(k, &batch.sealed()).await?;   // replayable by any registered peer

        let mids = c.oracle.read_and_open(k, &batch.pairs(), &c.signer).await?; // verified on chain
        let refs = c.oracle.reference(&batch.pairs()).await?;          // Jupiter, gate only
        let plan = c.engine.cross(&c.ledger, &batch, &mids.within(&refs))?; // Chapter 7

        let mut fills = Vec::with_capacity(plan.residuals.len());
        for r in &plan.residuals {
            fills.push(c.router.route(k, r, &mids, &c.signer).await?); // recorded on chain
        }

        let heartbeat = !due;                               // carries no asset
        let next = c.engine.settle(&c.ledger, &plan, &fills, &deposits, heartbeat)?;
        let diff = c.ledger.encrypt_diff(&next)?;           // AES-256-GCM leaves
        let da = c.store.log_diff(k, &diff).await?;         // δ^(k) from the on-chain accumulator
        let msg = next.commit_message(k, head.root, da, &mids, &fills, &deposits);
        c.relayer.commit(k, c.signer.sign(&msg), msg).await?;
        c.ledger.apply(next);                               // only after the Commit confirms
    }
}
```

A Crossing has something to settle when the enclave holds any of the work below at the cutoff. All but the last row it decides from its own state, with no price.

Work	Counts when
Trade, cancel and Exit Intents	intake accepted one since the last cutoff
Deposits	a queue holds entries past the stored cuts, to credit or, when the note does not open, to refund
Exit pieces	a funded piece has reached its release slot
Expiries	an open order's expiry τ lies below the cutoff
A resting order the market reached	its limit satisfies (10.1)

Status and Receipt requests settle nothing and start no Crossing.

The last row needs a price. Every 25 slots (parameter: 5 s at the slot time of [10], a twelfth of the one-minute heartbeat on which Pyth refreshes the mids a resting order clears against [7]), the host reads Jupiter's Price API V3 [8] for the assets of the resting orders, and the enclave counts the Crossing as due when a resting order i on pair π, with reference ratio P_π of its base and quote USD prices, satisfies

$$s_i \left(\ell_i - P_\pi\right) \;\geq\; -(\kappa + \beta_0)\, P_\pi. \tag{10.1}$$

The band holds every limit a crossable mid can reach, since Chapter 7 crosses a pair only within κ of the same reference, widened by the floor β_0 of the route bound. The scan decides only whether a Crossing begins. The Crossing then reads its mids on chain and fills by Chapter 7 alone, so the reference can start a Crossing or leave an order resting until the next scan, and never sets a price.

With nothing to settle, nothing goes on chain until 750 slots after the last Commit (parameter: 150 s at the slot time of [10], so twelve heartbeat intervals fit in the liveness window W of Chapter 6). The Coprocessor then sends a heartbeat: a Crossing that drains no Intent, reads no price, routes nothing and carries no asset, which the program accepts with an empty asset list (Chapter 9). It is three transactions, `open_crossing`, one `log_diff` chunk and the Commit (est.: with no asset term, Equation 4.11 leaves the 80-byte header and any rewritten trader records, inside one 3,800-byte chunk). A day holds 432,000 slots at 200 ms [10], so an idle Pool lands 576 heartbeats, 1,728 transactions, a day (est.). Each heartbeat re-anchors the root and restarts the liveness window, so the gap between Commits measures the enclave's liveness and never trading activity.

The cutoff is the slot at which the loop begins Crossing k, never less than Δ after Commit $k-1$ (Chapter 7). While work waits, it falls Δ after the last Commit; after an idle stretch, at the slot the loop finds work. Intents received before the cutoff are the Crossing's candidates. One received later keeps the work flag set, and Crossing $k+1$ begins as soon as Δ allows. Intake stamps each Intent with the confirmed chain slot at which it arrives, not the loop's last read, so the rule holds while a Crossing's proofs are still being computed.

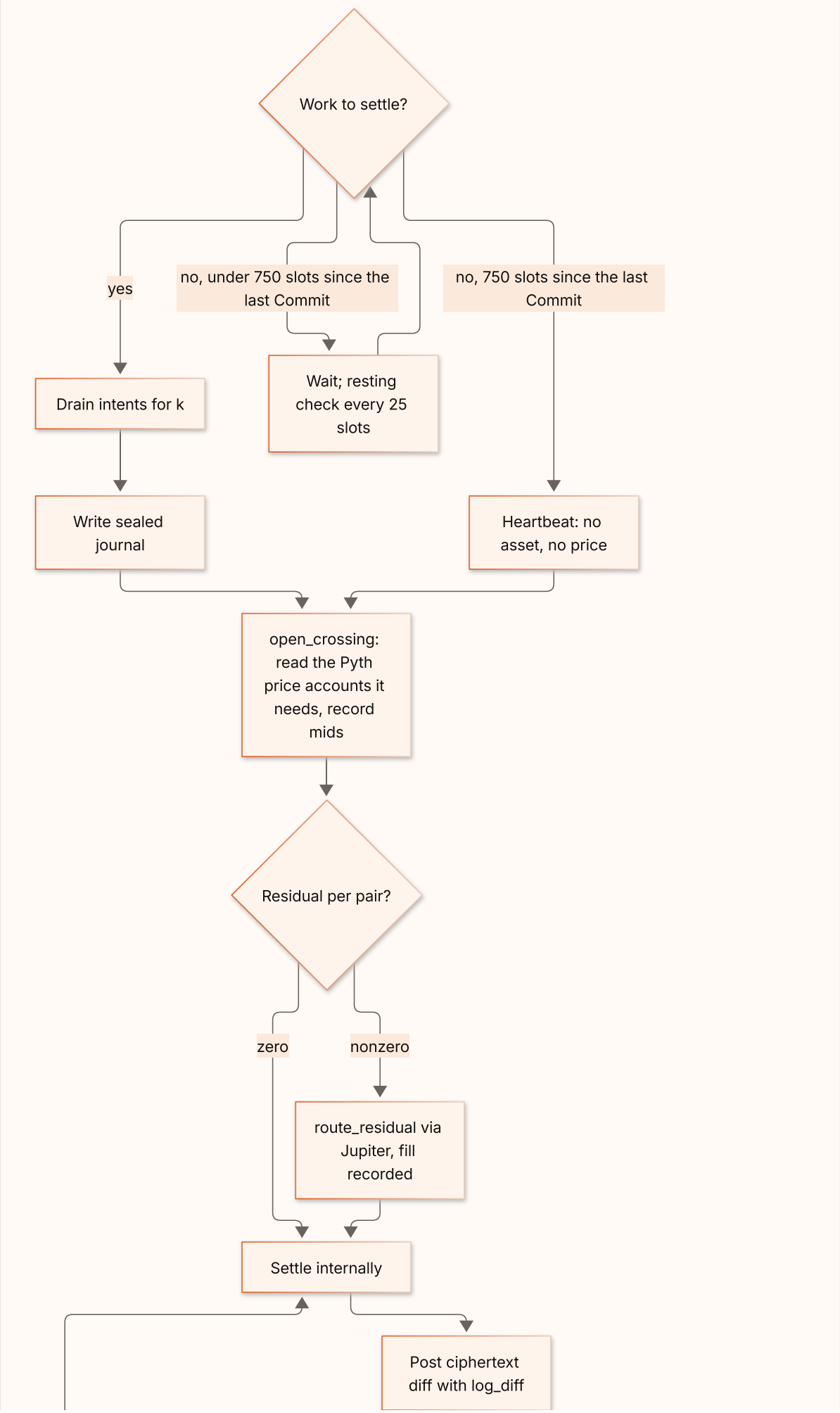
Three properties of this loop carry the rest of the design.

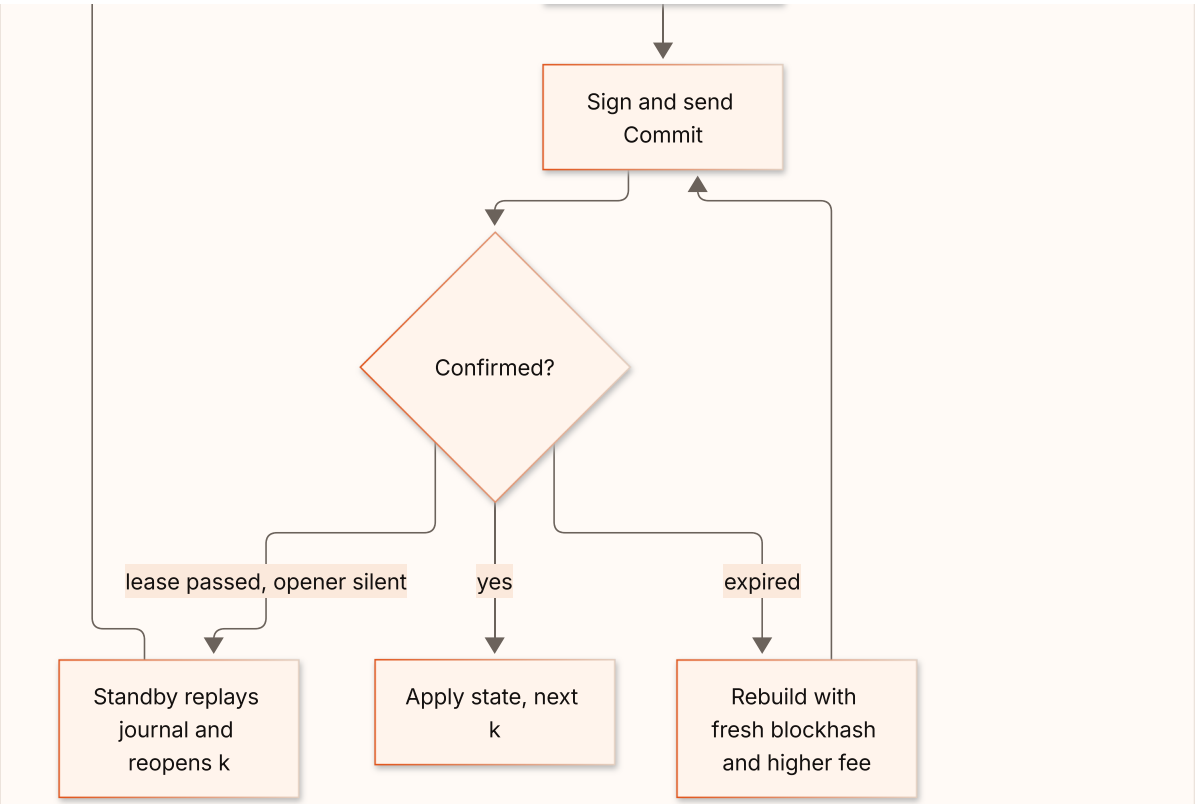
The engine is a pure function of $(\rho^{(k-1)}$, the plaintext Ledger, the Intent batch, the mids, the fills, the deposits). It reads no clock and no randomness, iterates over ordered maps only, and uses integer arithmetic in base units. Any enclave holding the same inputs computes the same $\rho^{(k)}$, which is what makes the journal and the standby useful.

The ciphertext diff is on chain before the Commit is signed. `log_diff` returns $\delta^{(k)}$, the accumulator (4.10) the program built from the confirmed chunks, and the Commit must match it, so every root the program accepts has its ciphertexts in Solana's history. The Escape (Chapter 6) and any trader's balance proof read from there.

The in-memory Ledger advances only after the Commit confirms. If the Commit fails for good, the enclave still holds the state that matches `LedgerRoot`, and the Crossing is retried with the fills already recorded.

The on-chain part of a Crossing is three dependent transaction stages: `open_crossing`, then the `route_residual` transactions, then the Commit. The routes of one Crossing all write its `CrossingState`, so the scheduler runs them one after another, but they need no confirmation between them and a leader can place them in the same block. Each stage waits for the previous one's confirmation, which Alpenglow targets at about 150 ms [11], inside one slot at the slot time Chapter 9 cites. The shortest Crossing is therefore about three slots (est.), plus the Jupiter round trips and the `log_diff` chunks, which run while the previous stage confirms where their inputs allow. `open_crossing` accepts Crossing $k+1$ no sooner than Δ slots after Commit k lands (Chapter 9), so Crossings never overlap, and a Pool with work always waiting runs one Crossing every Δ slots plus a Crossing's own duration.


Work to settle?
yes
no, under 750 slots since the last Commit
no, 750 slots since the last Commit
Drain intents for k
Wait; resting check every 25 slots
Write sealed journal
Heartbeat: no asset, no price
open_crossing: read the Pyth price accounts it needs, record mids
Residual per pair?
zero
nonzero
route_residual via Jupiter, fill recorded
Settle internally
Post ciphertext diff with log_diff



Intent intake

An Intent $i = (u, \pi, s, x, \ell, \tau, \nu)$ arrives as an HPKE ciphertext (RFC 9180) under the enclave's attested HPKE key ek_C, published in the `Registry` and bound to the measurement as described in Chapter 5 [13]. The suite is DHKEM(X25519, HKDF-SHA256), HKDF-SHA256 and AES-256-GCM, and the HPKE `info` string is (3.8) of Chapter 3: the program id, the key epoch and the hash of its quote, so a ciphertext made for one deployment or key epoch does not open under another. Intents reach the enclave either over a TLS session that terminates inside the enclave or from `post_intent` transactions picked up by the indexer; both paths deliver the same bytes to `intake`.

```
pub fn open_intent(&mut self, ct: &IntentCiphertext, slot: u64) → Result<Accepted, Reject> {
    let info = [b"privium/intent/v1".as_slice(), &PROGRAM_ID, &self.epoch.to_le_bytes(), &self.quote_hash].concat();
    let pt = hpke::single_shot_open::<Aes256Gcm, HkdfSha256, X25519HkdfSha256>(
        &OpModeR::Base, &self.hpke_sk, &ct.encapped_key, &info, &ct.body, &[],
    ).map_err(|_| Reject::Undecryptable)?;

    let signed: SignedIntent = borsh::from_slice(&pt).map_err(|_| Reject::Malformed)?;
    let i = &signed.intent;
    let acct = self.ledger.account(i.u).ok_or(Reject::UnknownAccount)?;
    acct.auth_key.verify_strict(&i.signing_bytes(), &signed.sig).map_err(|_| Reject::BadSig)?;

    if i.tau ⩽ slot { return Err(Reject::Expired); }
    if i.x == 0 || !self.pairs.contains(&i.pi) { return Err(Reject::Malformed); }
    if !acct.nonce_fresh(i.nu) { return Err(Reject::Nonce); }  // Chapter 4 nonce discipline

    let need = i.max_debit();          // x·ℓ for buys (ℓ capped for market orders), x for sells
    if acct.available(i.debit_asset()) < need { return Err(Reject::Balance); }
    self.ledger.reserve(i.u, i.debit_asset(), need);
    self.ledger.consume_nonce(i.u, i.nu);
    Ok(Accepted { id: i.id(), crossing_hint: self.next_k })
}
```

The trader's Ledger authorization key, not the wallet, signs the Intent, so placing an order needs no wallet prompt and leaves no wallet signature anywhere. The reservation makes a trader's concurrent Intents in one Crossing unable to overspend; reservations not filled by expiry τ are released at the next Crossing. Rejections are returned on the session only to the sender. The symmetric layer is not the bottleneck: Encifher measures over one million AES-256-GCM operations per second in its enclave [1].

Ciphertext store and data availability

The store handles two kinds of object per Crossing:

- the **journal** $J^{(k)}$: the Intent ciphertexts the enclave accepted, sealed again under a key derived from the Ledger data key that every admitted enclave holds (Chapter 5), written to the host store before `open_crossing`;
- the **diff** Diff_k of Definition 4.4: the header, the rewritten leaves with their positions, the per-asset proofs and the entry records, posted on chain as `log_diff` chunks before the Commit.

The diff holds the leaves the Crossing changed and the decoys of Chapter 8, so its size follows Equation 4.11, and Solana's transaction history is its data availability layer. Anyone can rebuild every root from genesis by replaying diffs in order and checking each result against the on-chain $\rho^{(k)}$ and $\delta^{(k)}$, which is how the indexer validates its own copy before serving it. A trader fetches their own leaf and its Merkle path from any mirror and decrypts it with their key; the leaf proves itself against the on-chain root, so the mirror needs no trust.

Oracle ingestion

Pyth sponsors price feed accounts on Solana for a set of commonly used feeds. For each feed id, the account at shard 0 of Pyth's push oracle holds a `PriceUpdateV2` that Pyth's own pusher keeps current, for most feeds on a one-minute heartbeat and on every 0.5% move of the price [7]. Privium reads those accounts as they stand and posts no price update of its own. The `oracle` component derives each admitted asset's account from the feed id in its `AssetState`, reads the accounts a Crossing needs in one `getMultipleAccounts` call after the cutoff, and hands them to the enclave, which signs their price inputs into the `open_crossing` message (Chapter 9). The accounts are live, so a push can land between that read and the open; the program then finds values other than the signed ones and rejects the open with `MidsMismatch`, and the host reads again. Once a Crossing is open its mids are fixed on chain, and a standby that reopens it keeps them.

`open_crossing` derives and checks each account and records whether its price is valid (Chapter 9). The maximum price age is 75 seconds (parameter): above the one-minute heartbeat with room for each push to land, so a live feed qualifies at every Crossing, and 15 seconds past the first missed beat of a feed whose pusher stops. A price past that age does not fail the open: the program records it as invalid, and the pairs that need it rest for that Crossing. The enclave reads the recorded mids and their validity back from the `CrossingState`, never from the host, and the confidence check of Chapter 7 runs on the same stored values.

A pushed price can trail the market by up to its heartbeat, so before it crosses, the enclave holds each valid mid against a live reference: Jupiter's Price API V3, which prices a token from its last swapped price across Solana venues after filtering outliers [8]. The host requests the USD prices of the pair's base and quote mints, the enclave divides them, and the pair crosses only while its mid lies within κ of that ratio (Chapter 7). The reference only decides whether a pair crosses: the clearing price is always the mid the program verified, so a host that withholds or bends the reference can make a pair skip a Crossing and nothing more. A pair whose `AssetState` names no feed has no verified mid and routes its whole flow as the Residual, its route bound set from the Jupiter reference price of its token.

Jupiter route building

Residuals use Jupiter's Router path, `GET https://api.jup.ag/swap/v2/build`, which returns raw instructions and address lookup tables from Metis routing with no Jupiter swap fee [2]. Requests carry the team's key in the `x-api-key` header; the paid tiers allow 10 to 150 requests per second, against 0.5 for keyless calls [4], and one Crossing makes one request per pair with a nonzero Residual plus rebuilds.

```rust
pub async fn build_route(&self, r: &Residual, mids: &Mids) -> Result<SignedRoute> {
    let y_min = r.min_out(mids);                        // Chapter 7 bound in base units
    let slippage_bps = r.slippage_bps_for(y_min);       // the same bound, as Jupiter's knob
    let q = [
        ("inputMint", r.in_mint.to_string()),
        ("outputMint", r.out_mint.to_string()),
        ("amount", r.in_amount.to_string()),
        ("taker", POOL_AUTHORITY.to_string()),
        ("payer", self.fee_payer.to_string()),
        ("destinationTokenAccount", vault0(&r.out_mint).to_string()),
        ("wrapAndUnwrapSol", "false".into()),
        ("slippageBps", slippage_bps.to_string()),
        ("maxAccounts", ROUTE_ACCOUNT_BUDGET.to_string()),
        ("transactionVersion", "1".into()),
        ("excludeDexes", self.excluded.join(",")),
    ];
    let b: BuildResponse = self.http.get(BUILD_URL).query(&q)
        .header("x-api-key", &self.api_key).send().await?.error_for_status()?.json().await?;

    let ix = b.swap_instruction;
    if ix.program_id ≠ JUPITER_ID { return Err(RouteError::UnexpectedProgram); }
    let threshold: u64 = b.other_amount_threshold.parse()?;
    if threshold < y_min { return Err(RouteError::BelowBound); }
    self.check_accounts(&ix)?;                          // no authority-owned account but vault 0s
    Ok(SignedRoute { k: r.k, pair: r.pair, in_amount: r.in_amount, min_out: y_min, ix,
                     setup: b.setup_instructions, compute_unit_price: b.compute_unit_price })
}
```

With `transactionVersion=1` the response returns no lookup tables and no compute budget instructions, and gives a `computeUnitPrice` field for the transaction config instead [3]. `ROUTE_ACCOUNT_BUDGET` is 64 minus the accounts `route_residual` adds of its own (fee payer, `PoolConfig`, `Registry`, `CrossingState`, two `AssetState`, the authority, the Jupiter and Pool programs, the Ed25519 program, the Instructions sysvar and the token program): 52 (est.). The enclave signs the route only after `check_accounts` confirms that the only token accounts owned by the authority in the list are the two shard-0 vaults, and that $y_{\min}$ is at least the Chapter 7 bound; the program then enforces the same bound on the measured fill.

Before signing, the relayer simulates the transaction with the compute limit at 1,400,000 and sets the limit to 1.2 times the units consumed, as Jupiter prescribes for `/build`, clamping the returned price to a configured ceiling [5]. A simulation that fails at the CPI depth or instruction-trace limits of Chapter 9 names the failing program; the router maps it to its DEX label with `GET /swap/v2/program-id-to-label` [6], adds it to `excludeDexes` and rebuilds.

Commit submission and retries

All Pool transactions are v1, which carries its limits in the message config: the compute unit limit, the loaded accounts data size and a priority fee stated as a total in lamports [9]. A v1 transaction with either limit unset gets zero and fails immediately [9], so the relayer always sets both from a simulation.

```rust
pub async fn commit(&self, k: u64, sig: Signature, msg: CommitMsg) -> Result<Confirmed> {
    let ixs = [ed25519_ix(&self.enclave_pk, &sig, &msg.bytes()), commit_ix(k, &msg)];
    let mut fee = self.fees.start(k);
    for attempt in 0..self.cfg.max_rebuilds {
        let bh = self.rpc.latest_blockhash().await?;
        let sim = self.rpc.simulate_v1(&ixs, bh, Limits::MAX).await?;
        if let Some(e) = sim.err() { return Err(classify(e)); } // StaleRoot, OutOfOrder: stop
        let limits = Limits {
            compute_units: (sim.units_consumed * 12 / 10) as u32,
            loaded_bytes: round_up_32k(sim.loaded_accounts_data_size),
            priority_fee_lamports: fee.next(attempt),
        };
        let tx = self.build_v1(&ixs, bh, limits)?.sign(&self.fee_payer)?;
        let until = bh.last_valid_block_height;
        while self.rpc.block_height().await? ≤ until {
            self.send_all(&tx).await;                   // same bytes, idempotent
            if let Some(c) = self.rpc.confirmed(&tx.signature()).await? { return Ok(c); }
            self.wait_slots(2).await;
        }
    }
    Err(CommitError::Exhausted)
}
```

A blockhash expires 150 blocks after it was issued [12], about 30 seconds at the slot time Chapter 9 cites. Within that lifetime the relayer resends the same signed bytes, which is safe because a transaction signature can land once. After expiry it rebuilds

with a fresh blockhash and a higher priority fee, which is also safe, because the Commit itself is idempotent at the program level: it requires $k = \texttt{LedgerRoot.k} + 1$ and $\rho^{(k-1)}$ equal to the stored root, so a second copy of an already-landed Commit fails with `OutOfOrder` and the relayer treats that error as confirmation after reading `LedgerRoot`. Under Alpenglow, live since epoch 1064 with finality targeted at about 150 ms [11], the relayer applies the new state on finalized confirmation without a long wait. The relayer does not use durable nonces for Commits; durable nonces are being replaced by on-chain offline-signer programs [15], and a Commit has no need for an offline signing window.

The lease covers a silent opener. `open_crossing` records the opening enclave key, and only that key may route and commit Crossing k until `lease_slots` have passed. The lease is 300 slots (parameter): two full blockhash lifetimes for the opener to land its Commit, and well inside the liveness window W of Chapter 6. Past the lease, a standby enclave reads the journal $J^{(k)}$, reopens k with the mids and fills already on chain, and replays the pure engine to the same settlement. Residual fills already executed are part of its input, so no swap is repeated.

Indexer

The indexer decodes every Pool transaction into typed events: deposits with their note blobs, `post_intent` ciphertexts, Crossing opens with their mids, recorded fills, Commits with their roots and DA commitments, Exit releases and Escape claims. It serves three consumers: the enclave (deposits since a cursor, Intent blobs since a slot), the SDK (a trader's leaf and Merkle path, Exit release status) and the public (per-Crossing net flow, which the chain already shows).

Because Pool transactions are v1, every read path passes `maxSupportedTransactionVersion: 1`. Without it `getTransaction` fails with error -32015 and block reads cannot decode the transaction, and clients that read compute limits or priority fees from ComputeBudget instructions find nothing, since v1 keeps them in the message config [9]. The indexer checks each replayed root against the program's `LedgerRoot` and each diff against $\delta^{(k)}$, and stops serving a Crossing whose data does not reproduce the on-chain root.

The TypeScript SDK

The SDK puts a trade behind one call:

```ts
import { Privium } from "@privium/sdk";

const privium = await Privium.connect({ wallet, rpc: process.env.RPC_URL! });

await privium.deposit("USDC", 500);
const fill = await privium.trade("SOL/USDC", "buy", 2.5);          // market order
await privium.trade("SOL/USDC", "sell", 1, { limit: 210, expiry: "30s" });
await privium.exit("SOL", 1.5, { to: freshAddress });
```

Underneath, `trade` builds and seals an Intent and waits for the Crossing that settles it:

```ts
export async function trade(
  this: Privium, pair: string, side: "buy" | "sell", size: number, opts: TradeOpts = {},
): Promise<Fill> {
  const { base, quote } = await this.markets.resolve(pair);           // mints, decimals, feeds
  const slot = await this.rpc.getSlot().send();
  const intent: Intent = {
    u: this.account.id,
    pi: [base.mint, quote.mint],
    s: side === "buy" ? 1 : -1,
    x: toBaseUnits(size, base.decimals),
    l: opts.limit === undefined ? marketLimit(side) : toPriceUnits(opts.limit, base, quote),
    tau: slot + slotsFrom(opts.expiry ?? "60s"),
    nu: await this.account.nextNonce(),
  };
  const sig = await this.account.authKey.sign(intentSigningBytes(intent));
  const enclave = await this.registry.activeKey();                    // attested (Chapter 5)
  const ct = await hpkeSeal(enclave.hpkePk, encodeSignedIntent(intent, sig), intentInfo(enclave));
  const receipt = this.opts.onChain
    ? await this.postIntent(ct)                                       // post_intent, any payer
    : await this.intake.submit(enclave, ct);                          // session to the enclave
  return this.watchFill(receipt, intent);                             // reads own leaf, decrypts
}
```

`watchFill` polls the indexer for the trader's leaf at each new Commit, verifies its Merkle path against the on-chain root and decrypts it locally; the fill price and size come out of the trader's own balance change. Prices and sizes cross the API as decimals and are converted to base units once, with the mint's decimals; nothing downstream sees a float. `slotsFrom` converts

durations at the current slot time read from the cluster, because slot length changed in 2026 and code that hardcodes 400 ms per slot drifts [10].

Wallet flows

A trader's wallet appears on chain for deposits and nowhere else. Everything after the deposit is signed by keys derived from one wallet signature:

```
const DOMAIN = "Privium account v1\nThis signature derives your Privium keys. It moves no funds.";

export async function deriveAccount(wallet  MessageSigner): Promise<AccountKeys> {
  const sig = await wallet.signMessage(new TextEncoder().encode(DOMAIN)); // Ed25519, deterministic
  const ikm = await crypto.subtle.importKey("raw", sig, "HKDF", false, ["deriveBits"]);
  const derive = async (label  string) ⇒ new Uint8Array(await crypto.subtle.deriveBits(
    { name: "HKDF", hash: "SHA-256", salt: new Uint8Array(32), info: utf8(label) }, ikm, 256));
  return {
    authSeed: await derive("privium/auth/v1"),     // A_u: Ed25519 seed that signs Intents
    viewSeed: await derive("privium/view/v1"),     // vk_u: viewing root (Chapter 11)
    escapeSeed: await derive("privium/escape/v1"), // y_u: escape key (Chapter 6)
    salt: await derive("privium/salt/v1"),         // varsigma_u: owner-tag salt (Chapter 4)
    noteSeed: await derive("privium/note/v1"),     // X25519 seed for deposit notes
  };
}
```

Ed25519 signatures are deterministic [16], so the same wallet always yields the same keys and a trader recovers their account on any device from the wallet alone. The flows are:

Flow	Wallet prompts	What lands on chain
Open account	one message signature	nothing
Deposit	one transaction	a token transfer into a vault shard and a queue entry
Trade	none	nothing from the trader; the Pool's net Residual per Crossing
Exit	none	a release from a vault to the address the trader named
Escape	none	a request, Council re-encryption shares and the trader's claim proof, paid to the trader's address (Chapter 6)

An Exit can pay a Helius Ring deposit address, so the funds enter an encrypted balance on arrival [17]; Chapter 8 measures what that does to linkage.

Observability

The enclave emits structured events from a fixed schema and nothing else; there is no free-form logging inside the boundary, so no code path can print plaintext. The schema carries only values the chain already shows or that describe the machine: Crossing index, stage latencies (drain, open, cross, route, DA write, Commit), transaction outcomes and error classes, enclave health and key epoch. Counts of Intents and rejections are exported per Crossing rounded up to a power of two, enough to size capacity without giving a timing analyst (Chapter 8) a per-batch count. The host exports the rest: RPC latency and error rates, Jupiter response times, DA write latency, the relayer's fee spend and SOL balance. Alerts fire on the conditions that precede a missed Commit: a lease more than half spent, a DA write past its budget, a relayer balance below one day of fees.

Deployment and reproducible enclave builds

Two Coprocessor instances run on separate hosts with separate cloud attestation roots where the TEE allows it, each with its own relayer fee payer. Both are registered in the `Registry`, which holds up to four keys so rotation can overlap (Chapter 5). One opens each Crossing; the other follows the journal and the chain and takes over at lease expiry. Because the engine is deterministic and every input is either on chain or in the journal, the standby needs no live channel to the primary.

The enclave image is built so that anyone can reproduce its measurement from the tagged source. The toolchain is pinned in `rust-toolchain.toml`, every crate is vendored and locked, timestamps come from the commit, and the image is assembled by a hermetic build that records its inputs:

```
#!/usr/bin/env bash
set -euo pipefail
git checkout "v${RELEASE}"
export SOURCE_DATE_EPOCH="$(git log -1 --format=%ct)"
export CARGO_HOME="$PWD/.cargo-home" RUSTFLAGS="-C debuginfo=0 --remap-path-prefix=$PWD=/src"
cargo vendor --locked vendor/ > .cargo/config.toml
cargo build --release --locked --offline -p privium-enclave
nix build .#enclave-image --option sandbox true            # kernel, initrd, rootfs, cmdline
./tools/measure result/image > measurement.json            # values the quote must report
sha256sum result/image measurement.json
solana-verify build --library-name privium_pool            # on-chain program, Docker build
```

The release publishes the image hash, the expected measurement values and the program hash side by side. The program hash is checked with the Solana Verify CLI, whose record sits in a PDA anyone can recheck [14]; the measurement is what `register_enclave` compares against the attestation quote, through the chain of trust described in Chapter 5. A trader who wants to know what code holds their balances rebuilds both and compares two hashes against two on-chain records.

References

1. N. Lokhande et al., “Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana”, IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
2. “Build” (Swap API V2 Router path), Jupiter, 2026. https://developers.jup.ag/docs/swap/build
3. “Transaction Versions (v0 and v1)”, Jupiter, 2026. https://developers.jup.ag/docs/swap/advanced/transaction-versions
4. “Rate Limits”, Jupiter Developer Platform, 2026. https://developers.jup.ag/docs/portal/rate-limits
5. “Compute Units and Priority Fees”, Jupiter, 2026. https://developers.jup.ag/docs/swap/advanced/compute-units
6. “Program ID to Label”, Jupiter, 2026. https://developers.jup.ag/docs/api-reference/swap/program-id-to-label
7. “Sponsored Feeds” (Solana), Pyth Network, 2026. https://docs.pyth.network/pro/sponsored-feeds#solana
8. “Solana Price API” (Price API V3), Jupiter, 2026. https://developers.jup.ag/docs/price
9. “Versioned Transactions: legacy, v0 and v1”, Solana Foundation, 2026. https://solana.com/docs/core/transactions/versioned-transactions
10. “Reduced Slot Times” (SIMD-0525), Solana Foundation, October 2026. https://solana.com/upgrades/reduced-slot-times
11. “Alpenglow”, Solana Foundation, October 2026. https://solana.com/upgrades/alpenglow
12. “Constants Reference”, Solana Foundation, 2026. https://solana.com/docs/core/constants-reference
13. R. Barnes, K. Bhargavan, B. Lipp, C. Wood, “Hybrid Public Key Encryption”, RFC 9180, IRTF, February 2022. https://www.rfc-editor.org/rfc/rfc9180
14. “Verified Builds”, Solana Foundation, 2026. https://solana.com/docs/programs/verified-builds
15. “Durable Nonce Deprecation”, Solana Foundation, October 2026. https://solana.com/upgrades/durable-nonce-deprecation
16. S. Josefsson, I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA)”, RFC 8032, IETF, January 2017. https://www.rfc-editor.org/rfc/rfc8032
17. “Solana Privacy Rings”, Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings

Chapter 11

Selective disclosure and Receipts

A confidential account still has to be shown to the people who need to see it: an accountant closing a tax year, a fund administrator, a follower checking a caller's record. Privium gives the trader two instruments for this. Viewing keys open the trader's own Ledger entries to a chosen reader, scoped by asset, by period or down to a single entry. Receipts go the other way: instead of opening the data, the trader asks the Coprocessor for a signed statement about it (realized PnL over a period, volume, a return above a threshold), which anyone can verify against the chain and which names no wallet. This chapter defines both, with the key hierarchy, the PnL rules and the properties each provides, and the controls institutions attach to their accounts, including auditor keys in the form Helius Rings introduced [1].

Two ways to disclose

Disclosure on a public chain is all or nothing: a wallet's history is either published in full or kept in a wallet nobody knows about. Rings changed the default for balances. "Only the user can decrypt their balances with a viewing key," and the user can share ongoing read access or a transaction-specific viewing key that opens one transaction without the rest of the history [1]. Token-2022 confidential transfers reached the same place from the issuer's side: a mint may carry a global auditor ElGamal key, and when it does, every transfer instruction carries an additional encryption of the amount under that key [2].

Privium carries both ideas into the trading account and adds a third that only an attested coprocessor makes possible. The three instruments differ in what the reader learns.

Instrument	Reader learns	Reader needs	Trust the reader extends
Viewing key	the plaintext of the Ledger entries the key covers	the key and the Crossing diffs on Solana	none
Auditor key	every entry of an account, for as long as the account's policy names the auditor	the auditor's own secret key and the Crossing diffs	none
Receipt	one statement computed over the trader's entries, and nothing else	the Receipt, the Registry, the Ledger root history	the integrity of the enclave whose key signed it

A viewing key discloses data; a Receipt discloses a fact about data. A tax accountant wants the first. A follower wants the second, and is better served by it: a Receipt for, say, "realized PnL of $+41{,}200$ USDC over Crossings k_0 to k_1" lets the follower check the number without learning which tokens were traded, when, or from which deposit address.

The viewing key hierarchy

Where the owner component lives

Chapter 4 defines the Ledger leaf. For this chapter one property of it matters: every leaf of trader u carries an owner component, the leaf's plaintext encrypted under AE with the leaf key $k_{u,a} = \mathsf{F}(vk_{u,a}, \texttt{"balance"})$, which only u and the Coprocessor can derive. The Council handle in the same leaf serves the Escape (Chapter 6) and is never used for disclosure. Alongside balance leaves, the Coprocessor writes one entry record per event that touches u (a deposit credit, a fill, a fee, an Exit debit), encrypted under the entry keys below and carried in the Crossing diff that the Commit seals through its data availability digest (Chapter 4, Definition 4.4). Viewing keys are keys to these owner components.

Deriving the root

A trader's viewing root is derived from the wallet that opened the account, so it can be rebuilt on any device without a backup:

$$\sigma_u = \mathsf{Sign}_{sk_u}(m_{\mathrm{acct}}), \qquad vk_u = \mathsf{HKDF}\big(\sigma_u,\ \texttt{"privium/view/v1"}\big) \tag{11.1}$$

where m_{acct} is the account message of Chapter 10, the one wallet signature from which the SDK also derives the authority key A_u and the escape key y_u under other labels. Ed25519 signatures are deterministic (RFC 8032 derives the nonce from the key and the message [3]), so the same wallet always yields the same σ_u, and the domain-separated message cannot be replayed as a transaction signature. The confidential-balances documentation derives encryption keys the same way, "deterministically derived from an owner signing key" and never reusing the signing key itself [2]. When the account is opened, the SDK encrypts vk_u to the Coprocessor's attested HPKE key ek_C (Chapter 3, Chapter 5) and the enclave keeps it in the trader's record

(Chapter 4). The Coprocessor already reads every balance in plaintext inside the enclave to compute a Crossing, so holding vk_u adds nothing to what it can read; it lets the enclave encrypt each new entry to keys the trader can later hand out.

Asset keys and period trees

From the root, keys branch first by asset and then by time. Let F be HMAC-SHA-256 used as a PRF. The asset key is

$$vk_{u,a} = \mathsf{F}\big(vk_u,\ \texttt{"asset"} \,\|\, a\big). \tag{11.2}$$

Time is divided into disclosure epochs of E Crossings each, $E = 1$ (parameter: one epoch per Crossing makes a period key exact to the Crossing; coarser epochs only shorten the tree). Under each asset key sits a binary tree of depth D in the style of Goldreich, Goldwasser and Micali [4]: the root node is $vk_{u,a}$, and each node v has children

$$v_0 = \mathsf{F}(v, 0), \qquad v_1 = \mathsf{F}(v, 1). \tag{11.3}$$

The leaf reached by the D-bit path equal to epoch number e is the epoch key $vk_{u,a,e}$. With $D = 40$ (parameter: 2^{40} epochs outlast any realistic run of Crossings at any batch interval Chapter 7 considers), the tree never fills.

The key that encrypts entry j of trader u in asset a during epoch e is

$$k_{u,a,e,j} = \mathsf{F}\big(vk_{u,a,e},\ \texttt{"entry"} \,\|\, j\big), \tag{11.4}$$

and the owner component of the entry is $\mathsf{AE.Enc}(k_{u,a,e,j}, \mathit{entry}, ad)$ with associated data $ad = (\mathsf{id}_{\mathrm{prog}}, a, e, j, \mathsf{H}(u))$, so a ciphertext cannot be moved to another position and still decrypt. The entry itself is referenced the way Encifher references all encrypted state, by a 128-bit handle [5], and its ciphertext is published on Solana in the Crossing diff.

Scoping a disclosure

A disclosure is a set of nodes. To open asset a for epochs $[e_0, e_1]$, the trader sends the minimal set of tree nodes whose subtrees cover exactly that range. To open one entry, the trader sends $k_{u,a,e,j}$ alone, the per-transaction disclosure Rings offers [1]. To open everything, the trader sends vk_u.

Scope	Key handed over	Size
Whole account, all time	vk_u	32 bytes
One asset, all time	$vk_{u,a}$	32 bytes
One asset, epochs $[e_0, e_1]$	covering nodes of the period tree	at most $2D$ nodes
One entry	$k_{u,a,e,j}$	32 bytes

Lemma 11.1 (Range cover). For any $0 \le e_0 \le e_1 < 2^D$ there is a set N of at most $2D - 2$ tree nodes whose subtrees' leaves are exactly $\{e_0, \dots, e_1\}$.

Proof. Walk the paths from the root to e_0 and to e_1. Below the node where they split, take every right sibling hanging off the path to e_0, every left sibling hanging off the path to e_1, and the two endpoints, merging any two chosen siblings into their parent. Each path contributes at most one node per level over at most $D - 1$ levels, so $|N| \le 2(D - 1)$; every leaf strictly between e_0 and e_1 lies in exactly one chosen subtree, and each chosen subtree lies on the inner side of a boundary path. □

The guarantee the trader cares about is that the covering set opens nothing outside the range.

Theorem 11.2 (Scoped disclosure). Let N be the covering set for $[e_0, e_1]$ in asset a. If F is a secure PRF and AE is IND-CCA2 secure, then for any probabilistic polynomial-time reader $\mathcal{A}$ given N and every ciphertext in the store, the advantage of $\mathcal{A}$ in distinguishing the owner components of any entry outside $(a, [e_0, e_1])$ from encryptions of random entries of equal length is $\mathsf{negl}(\lambda)$.

Proof sketch. No ancestor of a node outside the subtrees of N is in N, and no such ancestor is computable from N, since F only maps downward. Replace the tree hybrid by hybrid, top-down: in hybrid H_d, every node at depth $\le d$ not in N or below a node of N is replaced by a uniform string. A distinguisher between H_{d-1} and H_d breaks the PRF at a node at depth $d - 1$ whose key is already uniform in H_{d-1}, so each step costs at most the PRF advantage, and there are D steps per asset and one step for the asset layer. In the final hybrid every entry key outside the range is uniform and independent of $\mathcal{A}$'s view, and an encryption under a uniform unseen key is indistinguishable from an encryption of a random entry under IND-CCA2. The total advantage is at most $(D + 1)\,\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{F}} + J \cdot \mathsf{Adv}^{\mathsf{cca}}_{\mathsf{AE}}$ for J entries outside the range, which is negligible. □

A disclosure is therefore bounded in time by construction. A reader holding the node set for last quarter keeps reading last quarter and nothing that comes after it, because future epochs sit under nodes the reader was never given.

Rotation

A trader who has handed out vk_u or an asset key closes future entries to that reader by rotating the root: the trader derives the next root from (11.1) with the label `privium/view/vr` for a version counter r, sends it to the enclave in a `RotateViewing` Intent, and from the next Crossing entries are encrypted under the new tree, with r in their associated data. Entries already written stay readable to the old key's holder; the next epoch begins under a key that holder never had.

Reading with a viewing key

The SDK's reader takes a key bundle (the node set plus asset and root version, small enough for a QR code), fetches the covered entry ciphertexts from the Crossing diffs, derives each entry key from whichever bundle node is its ancestor, decrypts, and checks each diff against the accumulator δ_k that an accepted Commit sealed (Chapter 4). A reader who reconstructs a history is therefore reading entries the Solana program accepted, not files a server chose to return.

Receipts

What a Receipt states

A Receipt is a statement the Coprocessor computes over one trader's Ledger entries for a span of Crossings, signs with sk_C, and returns to the trader. The trader chooses which fields it carries.

Definition 11.1 (Receipt). A Receipt is a tuple

$$\mathcal{R} = \big(ver,\ \mathrm{id}_{\mathrm{prog}},\ \psi,\ k_0,\ k_1,\ \rho^{(k1)},\ \Sigma,\ pk_\psi,\ pk_C,\ att_ref,\ s_R\big) \quad \text{with signature} \quad \sigma_C = \mathsf{Sign}_{sk_C}\big(\texttt{"privium/receipt/v1"} \,\|\, \mathsf{enc}(\mathcal{R})\big),$$

where ψ is the Receipt's pseudonym, $[k_0, k_1]$ the span of Crossings it covers, $\rho^{(k1)}$ the Ledger root committed at the end of the span, Σ a list of claims, pk_ψ a public key the trader controls, pk_C the signing key, att_ref the `EnclaveKey` record that binds pk_C to an attested measurement (Chapter 5), and s_R the slot at which the enclave signed. `enc` is a canonical, length-prefixed encoding.

A claim in Σ is either an exact value or a predicate:

Claim	Exact form	Predicate form
Realized PnL in q	$\mathrm{RPnL} = v$	$\mathrm{RPnL} \geq \theta$
Return on capital	$r\mathrm{TW} = v$	$r\mathrm{TW} \geq \theta$
Traded notional in q	$\mathrm{Vol} = v$	$\mathrm{Vol} \geq \theta$
Fills	$n_{\mathrm{fill}} = v$	$n_{\mathrm{fill}} \geq \theta$
Fees paid in q	$\mathrm{Fee} = v$	none
Ending equity in q	$\mathrm{Eq} = v$	$\mathrm{Eq} \geq \theta$
Assets traded	the set of a	the count of a

Predicate claims are the reason a Receipt can say less than a viewing key ever could. For example, a trader applying to a fund can prove "time-weighted return of at least 30% over the last 2,000 Crossings" without disclosing the actual return, the capital behind it, or the instruments. The enclave evaluates the predicate on plaintext and signs only its truth; the threshold θ is the trader's choice and appears in the Receipt.

The pseudonym and the holder key

A Receipt names no wallet. Its pseudonym is

$$\psi = \mathsf{F}\big(vk_u,\ \texttt{"receipt-id"} \,\|\, label\big), \tag{11.5}$$

where $label$ is a string the trader picks. Using the same label across Receipts makes them carry the same ψ, which is what a public caller wants: a stable handle whose track record accumulates. Using a fresh label makes each Receipt stand alone,

which is what a trader applying to two funds wants. The holder key pk_ψ is derived the same way, $sk_\psi = \mathsf{F}(vk_u, \texttt{"receipt-key"} \,\|\, label)$ mapped to an Ed25519 scalar, so the trader can answer a verifier's challenge by signing a nonce with sk_ψ and show that the Receipt is theirs to present, while a copied Receipt in someone else's hands cannot answer.

Producing a Receipt

The trader submits a `ReceiptRequest` Intent, encrypted to the Coprocessor's HPKE key like any other Intent, carrying the span, the claim selection, the label and a proof of account control (a signature by u's wallet over the request, verified inside the enclave and never published). The enclave recomputes the claims from the plaintext entries of u between k_0 and k_1 using the rules in the next section, checks that $\rho^{(k_1)}$ is a root that has been accepted on chain, signs, and returns the Receipt through the same encrypted channel. No transaction is sent and nothing about the request reaches the chain.

```
use ed25519_dalek::{Signer, SigningKey};

pub const RECEIPT_DOMAIN: &[u8] = b"privium/receipt/v1";

#[derive(Clone, borsh::BorshSerialize)]
pub enum Claim {
    RealizedPnl { exact: Option<i128>, at_least: Option<i128> }, // quote base units
    TimeWeightedReturn { exact_bp: Option<i64>, at_least_bp: Option<i64> },
    Volume { exact: Option<u128>, at_least: Option<u128> },
    Fills { exact: Option<u32>, at_least: Option<u32> },
    Fees { exact: u128 },
    Equity { exact: Option<u128>, at_least: Option<u128> },
    Assets { mints: Option<Vec<[u8; 32]>>, count: u16 },
}

#[derive(Clone, borsh::BorshSerialize)]
pub struct ReceiptBody {
    pub ver: u8,
    pub pool_id: [u8; 32],
    pub psi: [u8; 32],
    pub k0: u64,
    pub k1: u64,
    pub root_k1: [u8; 32],
    pub claims: Vec<Claim>,
    pub holder_pk: [u8; 32],
    pub coproc_pk: [u8; 32],
    pub att_ref: [u8; 32], // EnclaveKey record address (Chapter 5)
    pub signed_slot: u64,  // s_R
}

pub struct Receipt { pub body: ReceiptBody, pub sig: [u8; 64] }

pub fn sign_receipt(sk_c: &SigningKey, body: ReceiptBody) -> Receipt {
    let enc = borsh::to_vec(&body).expect("canonical encoding");
    let mut msg = Vec::with_capacity(RECEIPT_DOMAIN.len() + enc.len());
    msg.extend_from_slice(RECEIPT_DOMAIN);
    msg.extend_from_slice(&enc);
    let sig = sk_c.sign(&msg).to_bytes();
    Receipt { body, sig }
}
```

The domain prefix keeps a Receipt signature from ever verifying as a Commit signature, which the same sk_C produces over a different prefix (Chapter 5 and Chapter 9 give the Commit message). Borsh serialization is canonical for these types, so verifier and signer agree on the bytes without a second encoding layer.

Verifying a Receipt

Anyone can verify a Receipt with public data and no account of their own:

1. Decode the body and recompute the signed message with the domain prefix.
2. Read the `EnclaveKey` record named by att_ref from Solana and check that it records pk_C as an admitted Coprocessor key whose validity interval contains s_R, and that s_R is not earlier than the Commit of Crossing k_1 (registration by remote attestation is Chapter 5).
3. Verify σ_C under pk_C with Ed25519.
4. Check that $\rho^{(k_1)}$ is the root accepted by the Commit of Crossing k_1, from the Ledger root account's history or the Commit transaction itself (Chapter 9).
5. If the Receipt is being presented live, send a fresh nonce and check the presenter's signature under pk_ψ.

The SDK wraps this as `privium.receipts.verify(receipt)`, and a Receipt rendered as a link carries its encoded bytes in the fragment, so a verifier page checks it entirely in the browser.

Theorem 11.3 (Receipt soundness). Suppose Ed25519 is EUF-CMA secure, the enclave whose measurement is recorded in the Registry entry for pk_C runs the measured code with sk_C confined to it (enclave integrity, Chapter 5), and the Ledger roots accepted on chain satisfy the conservation and transition rules of Chapter 4. Then for any Receipt that passes verification, every claim in Σ is true of the entries committed under $\rho^{(k_1)}$ for the trader whose viewing root yields ψ, evaluated by the rules of the next section.

Proof sketch. Let $\mathcal{A}$ output a verifying Receipt with a false claim. Inside the measured code, sk_C signs a message with the Receipt prefix only on the path that recomputes Σ from the trader's entries and checks $\rho^{(k_1)}$, so by enclave integrity the enclave never signed this body. The Commit path signs under a different prefix, so no Commit signature is a valid Receipt signature on any body. Hence $\mathcal{A}$'s signature is a forgery on a message the signing oracle (the enclave, exposed through both paths) never signed, and $\mathcal{A}$ breaks EUF-CMA with the same probability. Step 4 of verification ties the Receipt to a root the program accepted, so the entries the claims describe are the ones the program's checks bound, not an alternative history. □ A Receipt rests on exactly what every Commit rests on.

Theorem 11.4 (Receipt unlinkability). Let $\mathcal{R}_0$ and $\mathcal{R}_1$ be Receipts issued with distinct labels, by the same trader or by two different traders. If F is a secure PRF, a party holding both Receipts and all public chain data distinguishes "same trader" from "different traders" with advantage at most $\mathsf{negl}(\lambda)$ plus the advantage obtainable from the claims Σ_0, Σ_1 and spans themselves.

Proof sketch. Replace $\mathsf{F}(vk_u, \cdot)$ with a random function; the change is undetectable by PRF security because vk_u never leaves the trader and the enclave. In the hybrid, ψ and pk_ψ are fresh uniform values for each distinct label, independent of u. The remaining fields, $k_0, k_1, \rho^{(k_1)}, pk_C$, *att_ref*, are functions of the span and of public chain state and do not depend on u. What remains is the information in the claims, which the trader chose to disclose. □

The claims are the trader's own lever. Chapter 8 quantifies how exact amounts correlate with deposits and Exits; predicate claims, or exact claims rounded to a resolution ϵ_R the request names, add nothing to that correlation.

PnL rules

The arithmetic behind a Receipt is fixed in the measured code, so two traders with the same history receive the same numbers. The Coprocessor computes PnL in the quote asset q, per Crossing, from the trader's own entries.

Positions and cost basis

For trader u and asset $a \neq q$, let $Q_a^{(k)}$ be the position in base units after Crossing k (equal to $b_{u,a}^{(k)}$) and $c_a^{(k)}$ the average cost per unit in q. A fill in Crossing k of size x at price p (the clearing price $p_\pi^{(k)}$ for a crossed fill, the executed Residual price for a routed fill, Chapter 7) with side s changes the position by $s\,x$. Write $Q = Q_a^{(k-1)}$ and $c = c_a^{(k-1)}$. Spot accounts hold no negative positions, so $Q \geq 0$ and a sell satisfies $x \leq Q$.

A buy ($s = +1$) raises the position and averages the cost:

$$Q' = Q + x, \qquad c' = \frac{Q\,c + x\,p}{Q + x}. \tag{11.6}$$

A sell ($s = -1$) realizes the difference between price and basis on the quantity sold and leaves the basis of the rest unchanged:

$$Q' = Q - x, \qquad c' = c, \qquad \Delta\mathrm{RPnL} = x\,(p - c). \tag{11.7}$$

Every fill also pays the protocol fee in q (Chapter 12), which realized PnL subtracts:

$$\mathrm{RPnL}^{(k_0,k_1)} = \sum_{k=k_0}^{k_1} \Big(\sum_{\text{sells in } k} x\,(p - c) \;-\; \sum_{\text{fills in } k} \phi\,x\,p \Big). \tag{11.8}$$

Average cost is the default (parameter: it is path independent within a Crossing and needs one number per asset). A request may select first-in-first-out instead, and the Receipt records the method.

Deposits, Exits and the mark

A deposit of asset a brings no trade history with it. The enclave sets its basis at the oracle mid of the Crossing that credits it, $m_{(a,q)}^{(k)}$, so PnL inside Privium is measured from the moment the asset entered. An Exit removes quantity at the current average cost and realizes nothing: moving tokens out is not a trade. Both rules keep PnL a function of trading decisions alone.

Unrealized PnL and equity at the end of a span use the oracle mid of Crossing k_1:

$$\mathrm{UPnL}^{(k1)} = \sum_{a \neq q} Q_a^{(k1)} \big(m_{(a,q)}^{(k1)} - c_a^{(k1)} \big), \qquad \mathrm{Eq}^{(k1)} = b_{u,q}^{(k1)} + \sum_{a \neq q} Q_a^{(k1)}\, m_{(a,q)}^{(k1)}. \tag{11.9}$$

The mid is the same price Chapter 7 clears at, from the same Pyth feed the Coprocessor ingests (Chapter 10), so a Receipt's mark is the mark at which the trader could have crossed in that Crossing.

Return on capital

A raw PnL figure says little without the capital behind it, and capital changes with deposits and Exits. Receipts report the time-weighted return, which removes the effect of external flows. Let $k_0 = t_0 < t_1 < \cdots < t_M = k_1$ be the Crossings at which u's account had a deposit or Exit, together with the endpoints. For each sub-span, let $\mathrm{Eq}^{-}_{t_j}$ be equity just before the flow at t_j and $\mathrm{Eq}^{+}_{t_{j-1}}$ equity just after the flow at t_{j-1}. Then

$$1 + r_{\mathrm{TW}} = \prod_{j=1}^{M} \frac{\mathrm{Eq}^{-}_{t_j}}{\mathrm{Eq}^{+}_{t_{j-1}}}. \tag{11.10}$$

Lemma 11.5 (Flow invariance). Suppose that on each sub-span j the trader's decisions earn a return g_j on the equity the sub-span starts with, so that $\mathrm{Eq}^{-}_{t_j} = (1+g_j)\,\mathrm{Eq}^{+}_{t_{j-1}}$. Then $r_{\mathrm{TW}} = \prod_j (1+g_j) - 1$, whatever the sizes of the deposits and Exits at $t_1, \ldots, t_{M-1}$.

Proof. Substitute the hypothesis into (11.10): each factor equals $1+g_j$, and the flows enter only through $\mathrm{Eq}^{+}_{t_{j-1}}$, which cancels within its own factor. □

The lemma is what makes return comparable across traders of different size: a trader who doubles the account halfway through a span and keeps trading the same way reports the same return as one who never added a token.

Fee and volume lines

Traded notional is $\mathrm{Vol} = \sum x\,p$ over all fills in the span and fees paid are $\mathrm{Fee} = \sum \phi\, x\, p$, crossed and routed fills alike. The crossed share of a trader's volume is available as a further claim: the trader's own crossing rate (Chapter 12).

Auditor keys and institutional accounts

The auditor key

A desk or a fund answers to someone who must see everything without asking the trader. Custom Rings meet this with "a separate auditor key" that grants read access to wallet balances inside the Ring under the policy the Ring configures [1], and Token-2022 meets it at the mint level with an auditor ElGamal key under which every confidential transfer carries an extra amount ciphertext [2]. Privium attaches the same role to an account.

An institutional account carries a policy record $\mathcal{P}_u$, a Ledger leaf of kind 3 (Chapter 4) anchored like any other leaf. Among its fields is an optional auditor public key pk_A, an X25519 key for HPKE. When pk_A is set, the Coprocessor writes, next to each owner component, an auditor component: the entry encrypted with HPKE to pk_A, with the same associated data. The auditor reads with its own secret key and needs nothing from the trader. Because the auditor component is a separate ciphertext, the auditor's access is independent of the trader's viewing keys: rotating vk_u does not close the auditor, and changing pk_A does.

Changing the policy is itself governed by the policy. $\mathcal{P}_u$ names a set of policy signers and a threshold t_A; a `SetPolicy` Intent carries t_A signatures over the new policy and the current policy's hash, which the enclave checks before applying. A desk whose policy requires its compliance officer's signature to change the auditor cannot have the auditor removed by a trader acting alone.

Proposition 11.6 (Auditor completeness). For an account whose policy has carried pk_A from Crossing k_A onward, every entry of that account written at or after k_A has an auditor component under pk_A, and the entry is published in the diff of its Crossing.

Proof sketch. The measured entry-writing path reads $\mathcal{P}_u$ from the same plaintext state it updates and emits the auditor component in the same step as the owner component, and both enter the Crossing diff that the Commit seals. By enclave integrity no entry is written by another path, and by the transition rules of Chapter 4 no root is accepted whose leaves were not produced by that path. □

Compliance hooks

Institutional policy goes beyond reading. The controls Custom Rings expose to institutions, allowlists and blocklists, freezes and additional required signatures enforced on Solana [1], are what a desk expects of its own account, and Privium's policy record carries them as rules the enclave applies before it accepts an Intent:

Hook	Field of $\mathcal{P}_u$	Applied at
Exit destinations	allowlist of addresses, or a screening attestation required per Exit	Exit Intent acceptance
Counter-signature	set of co-signers and threshold for Exits above a size	Exit Intent acceptance
Instrument set	allowlist of assets the account may trade	Intent acceptance in a Crossing
Position limits	maximum Q_a per asset, maximum notional per Crossing	Intent acceptance in a Crossing
Freeze	a flag the policy signers set, holding all Intents except `SetPolicy`	every Intent
Auditor	pk_A	every entry written

Each rule is checked inside the enclave on the plaintext Intent, so a desk's limits are enforced with the same guarantee as its balances, without its limits or its positions becoming public. A rejected Intent produces an entry of its own, visible to the auditor, so the audit trail includes what a policy stopped as well as what it allowed.

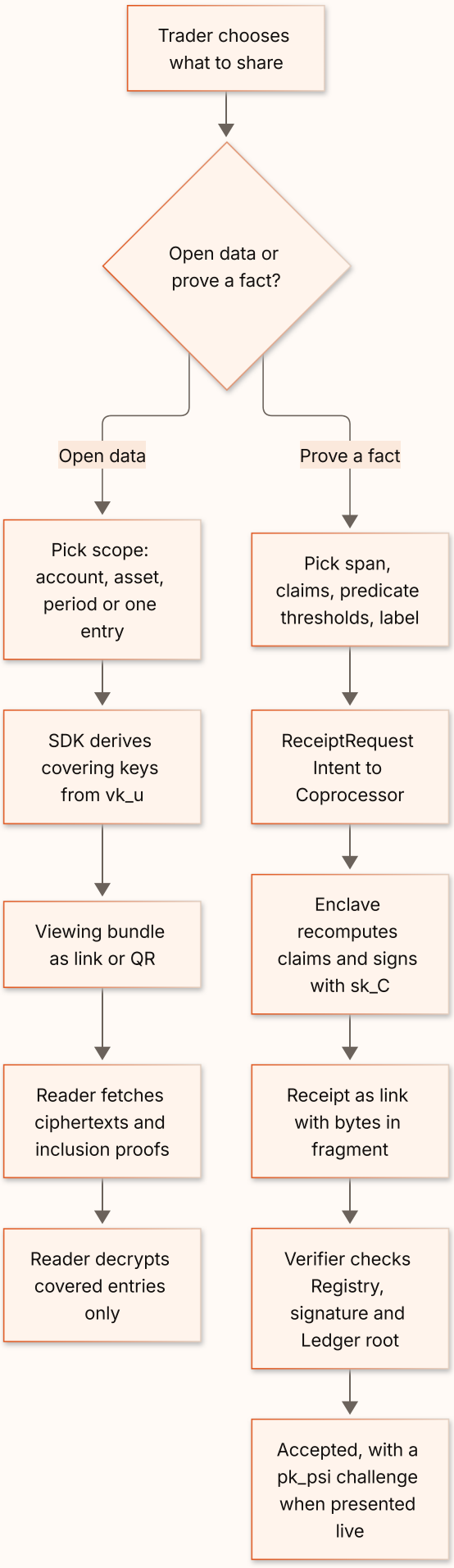
Screening sits at the account's edges, the only points where it touches a public address: a deposit's source and an Exit's destination. Helius lists Range, Chainalysis and Elliptic among the providers supporting wallet screening for Rings [1]. A policy that requires screening names the provider's attestation key, and the enclave accepts an Exit only with a signed screening result for its destination.

Exits through a Ring

An institution that wants the exit itself confidential routes it through a Ring (Chapter 8). A Custom Ring, confidential or anonymous, carries its own auditor key and policy [1], so the books stay continuous: the auditor reads the Privium account through pk_A, the Ring balance through the Ring's auditor key, and matches them on the Ring destination the Exit entry records.

The sharing flow

Disclosure is something a trader does in a few seconds from the account screen. The two paths split at the first choice: open data or prove a fact.


Trader chooses what to share
Open data or prove a fact?
Open data
Prove a fact
Pick scope: account, asset, period or one entry
SDK derives covering keys from vk_u
Viewing bundle as link or QR
Reader fetches ciphertexts and inclusion proofs
Reader decrypts covered entries only
Pick span, claims, predicate thresholds, label
ReceiptRequest Intent to Coprocessor
Enclave recomputes claims and signs with sk_C
Receipt as link with bytes in fragment
Verifier checks Registry, signature and Ledger root
Accepted, with a pk_psi challenge when presented live

From the SDK, both paths are one call:

```
const bundle = await privium.disclose({ asset: "SOL", crossings: [k0, k1] });
const receipt = await privium.receipt({
  crossings: [k0, k1],
  claims: { realizedPnl: { atLeast: 10_000n * 10n ** 6n }, fills: "exact" },
  label: "public-calls",
});
```

`disclose` returns a bundle that opens exactly the stated scope; `receipt` returns a signed Receipt and its link. A trader who publishes calls under one label builds a record anyone can check Receipt by Receipt, under a pseudonym that never touches the deposit wallet.

References

1. "Solana Privacy Rings", Helius, 2026. https://www.helius.dev/blog/solana-privacy-rings
2. "Confidential Balances: Overview", Solana Program Library documentation, Anza. https://www.solana-program.com/docs/confidential-balances/overview
3. S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, IETF, January 2017. https://www.rfc-editor.org/rfc/rfc8032
4. O. Goldreich, S. Goldwasser and S. Micali, "How to Construct Random Functions", Journal of the ACM 33(4), 1986. https://dl.acm.org/doi/10.1145/6490.6503
5. N. Lokhande et al., "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504

Chapter 12

Economics and $PRV

Privium charges one fee, a rate ϕ on the notional of every fill, and its economics come down to a comparison the trader makes on each order: what this trade costs through the Pool against what it costs as a swap from the trader's own wallet. This chapter writes that comparison as math. It separates the three ways a unit of an order can be executed (crossed inside the Pool, routed as part of the Residual, or swapped directly), derives the crossing rate as a function of the flow a Crossing sees, and shows how the cost of trading through Privium falls as that flow grows. It then follows the fee out of the Ledger to the token's launch wallet, through a market buyback of $PRV and a burn, and describes how the token launched. It closes with the build plan as milestones.

The fee

One rate on every fill

Every fill a trader receives in a Crossing pays

$$\mathrm{fee}(x, p) = \phi\, x\, p \tag{12.1}$$

in the quote asset q, where x is the filled size in base units and p the fill price: the clearing price $p_\pi^{(k)}$ for a crossed fill and the executed Residual price for a routed one (Chapter 7 defines both). The rate is the same on both kinds of fill. A trader never needs to know in advance how much of an order will cross, and the Receipt's fee line (Chapter 11) is the plain sum of (12.1) over the trader's fills.

Privium sets ϕ at 0.10% (10 bps). Every result below is stated for a general ϕ, and the inequalities that compare Privium with a direct swap state the range of ϕ for which they hold; at 10 bps a fully crossed order is cheaper than a direct swap whenever the venue fee, impact and ordering cost it avoids exceed 0.10% (Corollary 12.2).

Both sides of a crossed fill pay the fee, because both receive a fill. The Pool earns $2\phi\, x\, p$ on a crossed quantity x and $\phi\, x\, p$ on each routed unit, so per unit of matched flow the protocol earns the same rate whether the counterparty was another trader or the market.

Where the fee is taken

The fee is computed inside the Coprocessor as part of the Crossing's state transition and debited from each trader's encrypted quote balance $b_{u,q}^{(k)}$ in the same step as the fill. It leaves the Ledger and the Pool in the same Commit: the Commit attests the Crossing's fees $F_q^{(k)}$ and pays them from the quote vault to the fee recipient that the Pool configuration names (Chapter 9). The Residual is routed through Jupiter with no integrator fee attached: Jupiter's documentation states that "By default, there are zero protocol fees on Jupiter Swap," and that an integrator may add one with `platformFeeBps` and a `feeAccount` [1]. Privium leaves both unset and takes its rate in the Ledger, so the routed leg pays exactly what the public route charges and the fee arithmetic lives in one place.

Because the Commit pays the fee out before it checks conservation (Chapter 4), no fee term remains in the vault identity. For the quote asset at every committed root,

$$\sum_{u \in \mathcal{U}} b_{u,q}^{(k)} \;+\; X_q^{\mathrm{pend}} \;+\; P_q^{\mathrm{dep}} \;=\; V_q^{(k)}, \tag{12.2}$$

with X_q^{pend} the committed Exits not yet released and P_q^{dep} the deposits not yet credited (Equation 9.2). Each trader's fee is rounded down to whole units of q, and $F_q^{(k)}$ is the exact sum of those fees, so the vaults back trader balances to the last unit and the fee recipient receives exactly the fees the Commit attests.

Network costs

Each Crossing lands at least one Commit transaction and one Residual swap per routed pair, and their network fees are paid by the Coprocessor's fee payer. Those costs are per Crossing, not per trader, so they fall per unit of traded notional as the batch grows. Write c_{net} for the network cost of one Crossing in q and $\mathcal{V}^{(k)}$ for its total filled notional; the amortized network cost per unit is $c_{\mathrm{net}}/\mathcal{V}^{(k)}$. For scale, Encifher settles a confidential swap in a single Solana transaction at 157,000 compute units [2]; Chapter 9 gives Privium's own compute budget per instruction (est.). The analysis below folds c_{net} into the operator's side of the ledger and keeps the trader's cost to ϕ plus execution.

The cost of a trade, three ways

Execution cost relative to the mid

Measure every cost as a fraction of notional, relative to the oracle mid $m = m_\pi^{(k)}$ at the time the order is executed. For a pair π on public liquidity, define:

- f_D: the liquidity provider fee rate charged by the venues Jupiter routes through, as a fraction of notional.
- $I(x)$: the per-unit price impact and spread of swapping x on public liquidity, so that a buy of x pays $m\,(1 + I(x))$ per unit on average before fees. I is nondecreasing, $I(0) = 0$.
- $\xi(x)$: the expected per-unit value extracted from a public swap of x by ordering (sandwiching and back-running), bounded above by the slippage tolerance the swap carries.

These are properties of Solana's public liquidity at a given moment, not of Privium; Privium's costs are expressed in the same terms so the two can be compared.

Direct swap

A trader who swaps x from their own wallet pays

$$C_{\mathrm{dir}}(x) = f_D + I(x) + \xi(x). \tag{12.3}$$

The third term is the one a public order exposes. A swap of size x from a known wallet is visible as it is forwarded to the leader and in the wallet's history before and after; Chapter 8 treats the information it leaks and Chapter 7 the ordering value it offers.

Crossed fill

A unit that crosses inside the Pool is filled at the clearing price, which Chapter 7 sets at the oracle mid subject to the limits of the intents involved. It touches no venue, so it pays no liquidity fee and has no impact, and it is never a public transaction of its own, so it offers nothing to order against. Its cost is the fee alone:

$$C_{\mathrm{cross}} = \phi. \tag{12.4}$$

Routed fill

A unit that is not crossed is part of the Residual $R = R_\pi^{(k)}$, the net quantity the Pool swaps on the trader's side of the book. The Residual pays the public costs of a swap of size $|R|$, shared pro rata among the units it carries, plus the fee. Its ordering exposure is bounded by the slippage bound $\beta = \beta_\pi^{(k)}$ that the enclave signs into the routed swap (Chapter 7):

$$C_{\mathrm{route}}(R) = \phi + f_D + I(|R|) + \xi_R, \qquad \xi_R \le \beta. \tag{12.5}$$

An order's expected cost

Under the allocation rule of Chapter 7, an intent on side s of pair π has a crossed fraction θ_s and a routed fraction $1 - \theta_s$. With pro rata allocation, if X_+ and X_- are the total active buy and sell sizes in the Crossing (Q^+ and Q^- of Chapter 7),

$$\theta_s = \min\!\left(1,\ \frac{X_{-s}}{X_s}\right), \tag{12.6}$$

so the smaller side crosses entirely and the larger side crosses in proportion. The intent's cost per unit is

$$C_{\mathrm{P}}(\theta, R) = \phi + (1 - \theta)\big(f_D + I(|R|) + \xi_R\big). \tag{12.7}$$

Theorem 12.1 (When the Pool is cheaper). An order of size x costs less through Privium than as a direct swap if and only if

$$\phi \;<\; \Gamma(x, \theta, R) \;:=\; \theta\,\big(f_D + I(x) + \xi(x)\big) \;+\; (1 - \theta)\big(I(x) - I(|R|) + \xi(x) - \xi_R\big). \tag{12.8}$$

Proof. Subtract (12.7) from (12.3) and regroup: $C_{\mathrm{dir}} - C_{\mathrm{P}} = f_D + I(x) + \xi(x) - \phi - (1-\theta)(f_D + I(|R|) + \xi_R)$. Writing $f_D + I(x) + \xi(x) = \theta(f_D + I(x) + \xi(x)) + (1-\theta)(f_D + I(x) + \xi(x))$ and cancelling $(1-\theta)f_D$ gives $C_{\mathrm{dir}} - C_{\mathrm{P}} = \Gamma - \phi$. □

Γ is the gross saving the Pool produces before its fee, and Theorem 12.1 says the fee is paid out of it. The two terms have clear readings. The first is everything a crossed unit avoids. The second is what a routed unit gains from being routed as part of a batch rather than alone: the difference in impact between the trader's own size and the Residual, and the difference between the ordering exposure of a public wallet swap and that of a Pool swap bounded by β.

Corollary 12.2 (Full crossing). For a fully crossed order ($\theta = 1$), Privium is cheaper whenever $\phi < f_D + I(x) + \xi(x)$, the whole cost of the direct swap.

Corollary 12.3 (Small orders in a large Residual). For an order with $\theta < 1$, if the Residual is no larger than the order, $|R| \le x$, then $I(|R|) \le I(x)$ and the second term of Γ is at least $(1-\theta)(\xi(x) - \xi_R)$, so the routed part costs no more in impact than the direct swap would have.

The group view

Theorem 12.1 is the view of one order. The same comparison summed over everyone in a Crossing shows where the saving comes from. Let the Crossing on pair π hold buy intents of sizes $x_1, \dots, x_{n_+}$ and sell intents of sizes $y_1, \dots, y_{n_-}$, with totals X_+ and X_-, and let $R = X_+ - X_-$.

Proposition 12.4 (Group saving on liquidity fees). If each intent had instead been sent as its own public swap, the group would have paid liquidity fees of $f_D\, m\,(X_+ + X_-)$. Through the Pool it pays $f_D\, m\,|R|$. The saving on liquidity fees alone is

$$f_D\, m\,\big(X_+ + X_- - |R|\big) \;=\; 2 f_D\, m \min(X_+, X_-), \tag{12.9}$$

which is f_D times the crossed notional.

Proof. $X_+ + X_- - |X_+ - X_-| = 2\min(X_+, X_-)$, and only the Residual reaches a venue. □

The impact side of the group view depends on how public liquidity behaves between swaps. When arbitrage returns a pool to the external price between consecutive public swaps, each of the $n_+ + n_-$ swaps pays its own impact, $\sum_i x_i I(x_i) + \sum_j y_j I(y_j)$, against the Pool's single $|R|\, I(|R|)$; the Pool's impact is lower whenever

$$|R|\, I(|R|) \;\le\; \sum_i x_i I(x_i) + \sum_j y_j I(y_j). \tag{12.10}$$

When no arbitrage intervenes, a constant-product pool's state depends only on the net quantity swapped, so the group's total impact is the same either way and the Pool's saving is (12.9) plus the ordering value. In both regimes the Pool removes the ordering exposure of every crossed unit and bounds that of every routed unit by β.

Waiting for the batch

An intent waits for the next cutoff: Δ slots after the Commit in flight lands, or at once in an idle Pool (Chapter 7). Under the standard model in which the oracle mid is a martingale over the interval, the expected price change during the wait is zero, and the expected cost of waiting is zero; what the wait adds is variance, which a limit price ℓ bounds (Chapter 7 sets Δ and treats limits). Expected cost (12.7) is therefore the complete expected cost of an intent under that model.

The crossing rate

Definition

The quantity that drives (12.7) across all traders is the share of filled volume that crosses, the crossing rate γ_π of Chapter 7. For one pair in Crossing k,

$$\gamma_\pi^{(k)} = \frac{2\min(X_+, X_-)}{X_+ + X_-} = 1 - \frac{|R_\pi^{(k)}|}{X_+ + X_-}. \tag{12.11}$$

Averaged over traders by volume, the expected cost per unit of notional traded through Privium is

$$\bar{C}_{\mathrm{P}} = \phi + (1-\gamma)\big(f_D + \bar{I}_R + \bar{\xi}_R\big), \tag{12.12}$$

where $\bar{I}_R$ and $\bar{\xi}_R$ are the impact and ordering terms of the Residual. Equation (12.12) follows from (12.7) by weighting each intent's routed fraction by its size: the routed volume is exactly $|R|$, which is $(1-\gamma)$ of the total.

A model of flow

To see how γ moves with activity, model the intents on pair π arriving in a Crossing as a Poisson number N with mean $\Lambda = \lambda\Delta$, where λ is the arrival rate of intents per slot. Each intent independently is a buy with probability $\frac{1}{2}(1+\eta)$ and a sell otherwise, where $\eta \in [-1, 1]$ is the flow imbalance, and has size drawn independently from a distribution with mean $\bar{x}$ and coefficient of variation c_x, so $\mathbb{E}[x^2] = \bar{x}^2(1 + c_x^2)$.

The net quantity $D = X_+ - X_-$ is then a compound Poisson sum of signed sizes, with

$$\mathbb{E}[D] = \eta\,\Lambda\,\bar{x}, \qquad \mathrm{Var}[D] = \Lambda\,\mathbb{E}[x^2] = \Lambda\,\bar{x}^2(1 + c_x^2), \qquad \mathbb{E}[X_+ + X_-] = \Lambda\,\bar{x}. \tag{12.13}$$

The variance follows from the compound Poisson identity $\mathrm{Var}[\sum_{i \le N} Z_i] = \mathbb{E}[N]\,\mathbb{E}[Z^2]$ with $Z = \pm x$, since $Z^2 = x^2$ regardless of sign.

For large Λ, D is approximately normal with these moments, and $\mathbb{E}|D|$ is the mean of a folded normal. Writing $\mu_D = \eta\Lambda\bar{x}$ and $s_D = \bar{x}\sqrt{\Lambda(1 + c_x^2)}$,

$$\mathbb{E}|D| = s_D\sqrt{\tfrac{2}{\pi}}\,e^{-\mu_D^2/(2s_D^2)} + \mu_D\big(1 - 2\Phi(-\mu_D/s_D)\big), \tag{12.14}$$

with Φ the standard normal distribution function. Taking the ratio of expectations as the rate at large Λ, the expected crossing rate is

$$\gamma(\Lambda, \eta) \approx 1 - \frac{\mathbb{E}|D|}{\Lambda\,\bar{x}}. \tag{12.15}$$

Proposition 12.5 (Balanced flow). With $\eta = 0$,

$$\gamma(\Lambda, 0) \approx 1 - \sqrt{\frac{2(1 + c_x^2)}{\pi\,\Lambda}}, \tag{12.16}$$

which increases in Λ and tends to 1 at rate $\Lambda^{-1/2}$.

Proof. With $\mu_D = 0$, (12.14) reduces to $s_D\sqrt{2/\pi}$; divide by $\Lambda\bar{x}$. The expression is increasing in Λ because $\Lambda^{-1/2}$ is decreasing. □

Proposition 12.6 (Imbalanced flow). With $\eta \neq 0$, $\gamma(\Lambda, \eta) \le 1 - |\eta|$ for every Λ, and $\gamma(\Lambda, \eta) \to 1 - |\eta|$ as $\Lambda \to \infty$.

Proof. By Jensen's inequality $\mathbb{E}|D| \ge |\mathbb{E}D| = |\eta|\Lambda\bar{x}$, which gives the bound. As Λ grows, $\mu_D/s_D = \eta\sqrt{\Lambda/(1 + c_x^2)}$ grows without bound in absolute value, the exponential term of (12.14) vanishes and $1 - 2\Phi(-\mu_D/s_D) \to \mathrm{sign}(\eta)$, so $\mathbb{E}|D|/(\Lambda\bar{x}) \to |\eta|$. □

The two propositions say what the crossing rate depends on. Randomness in who arrives on which side costs a term that shrinks as $\Lambda^{-1/2}$, and so disappears with activity. Persistent one-sidedness costs $|\eta|$, which no amount of activity removes, and which is exactly the flow that has to reach the market: if more traders want to buy than sell, the difference is the Residual.

A worked example

The following figures are an example of the model, not measurements. Take balanced flow, $\eta = 0$, and sizes with $c_x = 1$ (an exponential-like spread of order sizes). Then (12.16) gives $\gamma \approx 1 - \sqrt{4/(\pi\Lambda)}$:

Intents per Crossing on the pair, Λ	$\sqrt{4/(\pi\Lambda)}$	Crossing rate γ
25	0.226	0.77
100	0.113	0.89
400	0.056	0.94
1,600	0.028	0.97

Each fourfold increase in Λ halves the routed share. With $\eta = 0.2$ the limit is $\gamma \to 0.8$, and the table's values approach it from below.

Activity lowers everyone's cost

Equation (12.12) together with Proposition 12.5 links the cost of trading through Privium to the activity the Pool sees:

$$\frac{\partial \bar{C}^{\mathrm{P}}}{\partial \Lambda} = -\frac{\partial \gamma}{\partial \Lambda}\big(f_D + \bar{I}_R + \bar{\xi}_R\big) \;<\; 0 \quad \text{for } \eta = 0, \tag{12.17}$$

holding the Residual's per-unit costs fixed. The same holds at any η until γ reaches its limit $1 - |\eta|$. $\Lambda = \lambda\Delta$ rises in two ways. More traders in the Pool raise λ, and every new trader lowers the expected cost for every existing one. A longer batch interval raises Δ, which trades crossing rate against waiting; Chapter 7 sets Δ with that trade in view. Because Λ counts intents per pair, the crossing rate is highest on the pairs that carry the most flow, which are also the pairs where a trader's orders are most watched.

Fee flow

The fee recipient

The Pool configuration names a fee recipient, the token's launch wallet L. Each Commit pays the Crossing's fees $F_q^{(k)}$ from the quote asset's vault shard 0 to L's associated token account for q, in the instruction that attests them. The Pool authority signs the transfer: a program-derived address, for which no private key exists, so only the Pool program signs for it, through `invoke_signed` [3]. The program derives L's token account from L, the token program and the mint, so a Commit pays no other account (Chapter 9). The fee therefore never rests in an account of the Pool, and every fee payment is a token transfer inside a Commit transaction that anyone can read on chain. Only the admin key the configuration names changes L.

Buyback and burn

A crank on Privium's keeper signs as L. On each interval it swaps the quote balance L has received from Commits for $PRV through Jupiter, and burns exactly what each swap delivered. Each swap spends an amount A of the quote asset, capped by a per-run limit $A_{\max}$ (parameter: the cap keeps each purchase small against the token's public liquidity so the buyback pays little impact, and repeated runs spread purchases over time), and requires the output to meet a minimum

$$out_{\min} = \frac{A}{\hat{P}}\,(1 - \beta_B), \tag{12.19}$$

where $\hat{P}$ is the price of $PRV in q that the crank's Jupiter quote gives and β_B is the buyback slippage bound (parameter: the same role β plays for the Residual, bounding what the swap can lose to ordering). The crank reads L's $PRV balance before and after the swap and burns the increase with the SPL Token `Burn` instruction, signed by L. The burn amount is the measured delivery, not the route's claim, so it is exactly what the fees bought: the developer's own first buy at launch, held in the same wallet, is never part of it.

The program carries no buyback instruction. It pays the fee to L and stops there, and the buyback runs as L's own transactions, each a Jupiter swap followed by a burn of its measured output.

Supply over time

Let $S_0 = 1{,}000{,}000{,}000$ be the fixed supply at launch. Over a period j the protocol accrues fees $\mathcal{F}_j = \phi\,\mathcal{V}_j$ in q, where $\mathcal{V}_j$ is the period's total filled notional counted once per fill as in (12.1). If the period's buybacks execute at an average price $\bar{P}_j$, the tokens burned in the period and the supply after J periods are

$$\mathcal{B}_j = \frac{\phi\,\mathcal{V}_j}{\bar{P}_j}, \qquad S_J = S_0 - \sum_{j=1}^{J} \frac{\phi\,\mathcal{V}_j}{\bar{P}_j}. \tag{12.20}$$

Each unit of notional traded through Privium removes $\phi/\bar{P}_j$ tokens from supply. By (12.1) and the two-sided fee on crossed fills, a crossed unit of base asset contributes $2\phi p$ to $\mathcal{F}_j$ and a routed unit ϕp, so for a pair with crossing rate γ the fee per unit of net market demand $|R|$ is

$$\frac{\mathcal{F}}{m\,|R|} = \phi\,\frac{X_+ + X_-}{|R|} = \frac{\phi}{1-\gamma}, \tag{12.21}$$

up to the difference between fill prices and the mid. The higher the crossing rate, the more fee each unit of flow that reaches the market carries with it, which is the protocol's side of the same activity that lowers each trader's cost in (12.17).

Proposition 12.7 (Monotone supply). S_J is nonincreasing in J, and every token removed is removed by an on-chain burn signed by L for the output of a buyback swap.

Proof. $\mathcal{F}_j \geq 0$ and $\bar{P}_j > 0$, so each $\mathcal{B}_j \geq 0$. The mint and freeze authorities of $PRV are revoked, so no instruction can increase supply, and a burn only decreases it. □

$PRV

$PRV is the token of the protocol, ticker PRV, mint `9FSjffoh8Yk8jdpr9EEEuxTD6mphYayfboxZMKWVPRV`, with a fixed supply of 1,000,000,000 at 6 decimals. It launched as a fair launch: the whole supply went to the public launch, where everyone bought on the same terms. Its mint and freeze authorities are revoked, which fixes the supply for good, and its metadata is immutable. The developer allocation is the developer's own first buy at launch, made on the same terms as every other buyer. $PRV trades on a Raydium CPMM pool against ZEC, `7PscVHXMuzSDK1Abz2rP3BrunEjNYdJtc5aNxS1htnDQ`.

The token's role is the one the previous section describes in full. Every Commit pays the protocol's fee share in q to the launch wallet, and the crank turns it into market purchases of $PRV, each followed by a burn of exactly what it bought. Anyone can check this flow: each Commit's fee transfer to L, each swap L signs and the burn after it, and the mint's supply are all public

transactions and accounts on Solana, and (12.20) reconciles them period by period against the filled notional the Commits record.

The fee rate that drives this is ϕ, 0.10%. Equation (12.20) is linear in ϕ and in $\mathcal{V}_j$, and Theorem 12.1 bounds ϕ by the saving the Pool produces, so the fee flow that reaches the token grows with the volume that finds Privium cheaper than a direct swap.

The build plan

Privium is built in two stages: the first build, a mainnet system that carries the complete custody and trading loop on two assets, and then the full product. Each milestone below lists what ships and the check that marks it done.

First build

Milestone 1 (Pool and Ledger). The Pool program with per-asset vaults for SOL and USDC, the Ledger root account, the Registry and the Council configuration (Chapter 9); the Coprocessor service with encrypted balances in the Ledger (Chapter 4) and an attested signing key registered on chain (Chapter 5). Done when a deposit credits an encrypted balance, a Commit lands with a verified enclave signature, and the conservation check of Chapter 4 passes at every root.

Milestone 2 (Crossings). Market-order Intents encrypted to the Coprocessor, Crossings at interval Δ with internal netting at the oracle mid, the Residual routed via Jupiter with a signed slippage bound (Chapter 7), and the fee of (12.1) paid to the fee recipient at every Commit. Done when a Crossing with opposing intents settles both at the mid, routes only $|R|$, and the fee recipient's receipts match the fees the Commits attested.

Milestone 3 (Exit and Escape). Exits to any address, the Council's distributed key generation, and the Escape path with liveness window W (Chapter 6). Done when an Exit releases tokens to a fresh address and an Escape rehearsal with Commits halted pays every owner their last committed balance.

Milestone 4 (Mainnet). The first build on mainnet with SOL and USDC, the TypeScript SDK with `privium.trade(...)` and the indexer (Chapter 10), and the reproducible enclave build whose measurement matches the registered one. Done when traders deposit, trade and exit on mainnet and the measurement published with the build reproduces from source.

Full product

Milestone 5 (Token universe and limits). Every token Jupiter routes as a tradable asset against the quote, Token-2022 mints included (Chapter 9), and limit orders with expiry. Done when a limit Intent on a long-tail pair rests across Crossings, fills when its limit is met, and expires at τ.

Milestone 6 (Disclosure). Viewing keys with the asset and period hierarchy, single-entry disclosure, and Receipts with exact and predicate claims, with the SDK's `disclose`, `receipt` and `receipts.verify` (Chapter 11). Done when a Receipt verifies in a browser from public chain data alone.

Milestone 7 (Ring exits). Exits through a Helius Ring, default and Custom (Chapter 8). Done when an Exit lands as a Ring deposit and the trader withdraws it from the Ring.

Milestone 8 (Institutional accounts). Policy records with auditor keys, Exit allowlists, counter-signatures, instrument sets, position limits and screening hooks (Chapter 11). Done when an auditor reads every entry of a policy account with its own key and a policy-blocked Exit appears in the auditor's trail.

Milestone 9 (Buyback). The buyback crank, signing as the launch wallet, with $A_{\max}$, β_B and its interval in the keeper's configuration. Done when the first run swaps the fees L received through Jupiter, burns the measured output and the mint's supply falls by that amount.

Milestone 10 (Council and enclave breadth). Proactive resharing across Council rotations (Chapter 6) and a second enclave platform registered by its own attestation path (Chapter 5). Done when a rotation completes with the Ledger's threshold key unchanged and Commits from both platforms verify against the Registry.

The milestones are ordered by dependency. Milestones 1 to 4 are the loop a trader needs: in, trade, out, and a guaranteed way out. Milestones 5 to 10 widen what the account holds, who can read it, where it can exit to, and what its fees do, each on the foundation the first build lays.

References

1. "Add Fees To Swap", Jupiter Developer Docs, Jupiter. https://developers.jup.ag/docs/swap-api/add-fees-to-swap
2. N. Lokhande et al., "Encifher: A Trusted-Execution Coprocessor for Confidential Computation on Solana", IACR ePrint 2026/1504, July 2026. https://eprint.iacr.org/2026/1504
3. "Program Derived Address (PDA)", Solana Documentation, Solana Foundation. https://solana.com/docs/core/pda