TECHNICAL WHITEPAPER

ZKPerp — Privacy-First *Perpetual Futures*

A zero-knowledge perpetual DEX built natively on the Aleo blockchain. Position sizes, entry prices, leverage, and PnL are cryptographically private by default — hidden from all other market participants, including the protocol itself.

PROTOCOL

`zkperp_core_v30.aleo` · `zkperp_core_eth_v29c.aleo` · `zkperp_core_sol_v29c.aleo` · `zkperp_oracle_v4.aleo` · `zkperp_compliance_v9.aleo` · `zkperp_amm_v6.aleo` · `zkdarkpool_v9.aleo` · `test_usdcx_stablecoin.aleo`

NETWORK	AUTHOR	VERSION
Aleo Testnet	Henk-Wim de Boer	v30 · May 2026

- Aleo Testnet Live
- Zero-Knowledge Native
- Leo 4.0
- MIT License
- 2-of-3 On-Chain Oracle Quorum
- Commit-Verified Close + Liquidate (v30)
- ZK Dark Pool Live
- KYC Compliance Layer

CONTENTS

§0 Abstract

ZKPerp is a decentralized perpetual futures exchange built natively on the Aleo blockchain — a Layer-1 network designed from the ground up around zero-knowledge proofs. Unlike existing perpetual DEX protocols (GMX, dYdX, Hyperliquid), where every position size, entry price, leverage ratio, and PnL figure is permanently visible on-chain, ZKPerp stores all position data in Aleo private records: encrypted UTXO-style objects readable only by the position owner's viewing key.

This is not a privacy layer bolted on top of a transparent system. ZKPerp is built in Leo 4.0 — Aleo's native language — and leverages the platform's fundamental property: the Aleo VM can verify the mathematical correctness of any computation — that a position was opened at a valid price, with valid collateral, producing a valid PnL — entirely through a ZK proof, without the contract ever learning the underlying values.

ZKPerp closes the most significant architectural gaps in privacy-preserving perpetuals: the **position commitment scheme** (eliminates the inflated-payout attack vector on close), a **fully on-chain 2-of-3 oracle quorum** (no off-chain coordinator), a **privacy-preserving KYC compliance layer**, and an institutional-grade **ZK Dark Pool**. Separately, the project maintains a complementary standalone **Concentrated Liquidity AMM** for USDCx/ALEO spot. The liquidation path extends the same commitment verification — `finalize_liquidate` recomputes the BHP256 commitment from a keeper's `LiquidationAuth` fields and computes the margin condition in-circuit against the live oracle price. Liquidation is performed by a **1-of-N keeper set** that races for a deterministic reward, removing any single trusted liquidator.

🔒

Position Privacy

Size, entry price, leverage, and PnL encrypted in Aleo private records. No one except the trader can read position data.

🛡️

Commit-Verify Termination

BHP256 position commitment stored at open time. Both close and liquidate recompute the hash from supplied params — no trusted figure accepted on either path.

📡

On-Chain 2-of-3 Oracle

Three independent relayers submit prices to zkperp_oracle_v4.aleo. 2-of-3 quorum enforced by the Leo contract — no coordinator.

🛂

ZK Compliance

KYC enforced at the circuit level via BHP256 Merkle allowlist. Identity never published — only the Merkle root is on-chain.

🌑

ZK Dark Pool

Batch auction settlement at uniform clearing price. Order size and limit price remain private until settlement.

💧

Concentrated AMM

Standalone spot subproject: Uniswap v3-style CL AMM for USDCx/ALEO, 0.3% fee, LP positions as private records.

§1 The Privacy Problem in DeFi Trading

Every perpetual DEX deployed on an EVM chain exposes all trading activity in full. Position opens, closes, liquidations, and LP deposits are written to public storage and indexed by dozens of data providers within seconds. This is not a minor inconvenience — it is a structural attack surface that fundamentally limits who can trade and on what terms.

Concrete Attack Vectors

- **Front-Running:** A whale opens a $50M long on ETH. Bots see the pending tx in the mempool. MEV bots buy ETH before the tx lands, then sell immediately after — pocketing the price impact at the whale's expense.
- **Liquidation Hunting:** Every open position's liquidation price is calculable from public data: entry price, collateral, and size are all on-chain. Coordinated traders push price toward known liquidation clusters to trigger cascading liquidations.
- **Strategy Copying:** Every trade by a skilled trader is visible the moment it settles. Competitors can replicate positions in real-time. Institutions cannot trade meaningful size without telegraphing their thesis to the entire market.

The Aleo Difference

Aleo's programming model distinguishes between **public state** (mappings: readable by anyone, stored on-chain) and **private records** (UTXO-style encrypted objects, readable only by the owner's viewing key). ZKPerp stores all position data in private records — privacy is the default, not an option.

> FUNDAMENTAL PROPERTY
>
> In Aleo, a smart contract transition can verify that an operation is mathematically correct — that a position was opened at a valid price with valid collateral — entirely via ZK proof, without the contract ever learning the actual numbers. The verifier learns only that the computation was correct, not what it computed over.

What Remains Public in ZKPerp

- Oracle prices — required for correct settlement, unavoidably public
- Pool state — total liquidity, aggregate long OI, aggregate short OI
- Commitment hashes — `position_commits[id]` is a BHP256 hash, reveals nothing about position parameters
- Transaction counts — number of positions opened/closed per block, not their contents

Everything else — position size, entry price, collateral, leverage, direction, PnL, and liquidation level — is encrypted in private records.

§2 System Architecture

ZKPerp consists of six integrated layers. Each layer has a clearly defined responsibility and trust boundary.

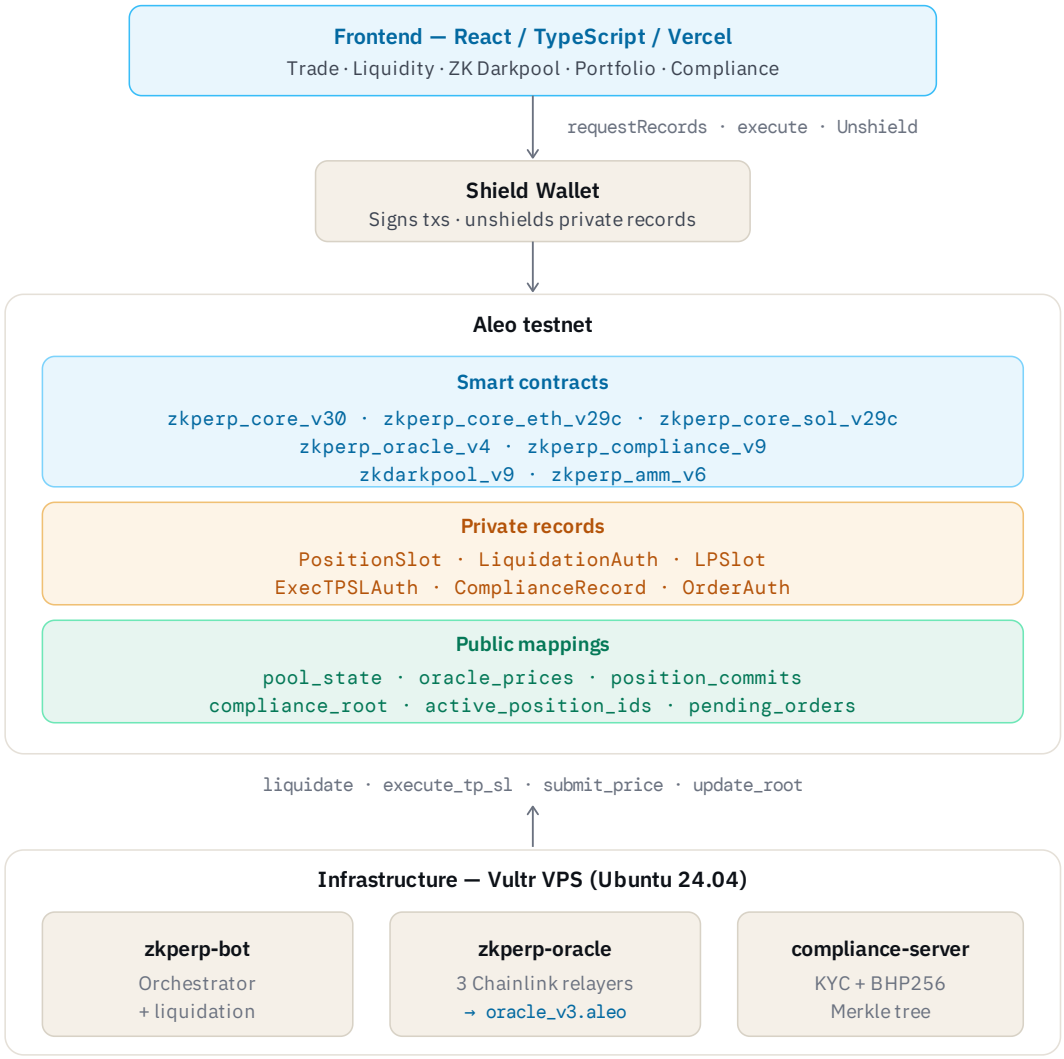

Figure 1. ZKPerp's layered architecture. Signed transactions flow down from the React frontend through the Shield Wallet, which unshields the user's private records before each call. The Aleo programs hold all position state — private records readable only by their owner, and public mappings carrying only commitments and aggregates. The Vultr VPS services run the privileged paths the chain cannot trigger itself: keeper liquidation, TP/SL execution, the 2-of-3 oracle submission, and compliance-root rotation.

Deployed Contracts

CONTRACT	MARKET / PURPOSE	STATUS
`zkperp_core_v30.aleo`	BTC/USDC perpetuals — commit-verified close + keeper-race liquidate	Live
`zkperp_core_eth_v29c.aleo`	ETH/USDC perpetuals	Live
`zkperp_core_sol_v29c.aleo`	SOL/USDC perpetuals	Live
`zkperp_oracle_v4.aleo`	2-of-3 on-chain quorum price oracle	Live
`zkperp_compliance_v9.aleo`	KYC compliance gating	Live
`zkdarkpool_v9.aleo`	ZK dark pool batch auction	Live
`zkperp_amm_v6.aleo`	Concentrated liquidity AMM (USDCx/ALEO) — standalone spot subproject; demo-grade, pending on-chain validation	Prototype
`test_usdcx_stablecoin.aleo`	Official Aleo testnet stablecoin	Official

Program Imports

Each core market contract declares the following imports, giving it access to the stablecoin transfer interface, compliance gating, and on-chain oracle prices:

```
import test_usdcx_stablecoin.aleo;
import zkperp_compliance_v9.aleo;
import zkperp_oracle_v4.aleo;
```

Protocol Parameters

Each parameter is enforced either *in-circuit* — proven in the transition body, which can read the trader's private inputs but not on-chain mappings — or in *finalize*, executed on-chain by validators against public mapping state. Values derived from private inputs are bound into a BHP256 commitment in-circuit and re-checked on-chain on the close and liquidation paths; values that depend on live pool or oracle state are necessarily computed and asserted in finalize.

PARAMETER	VALUE	WHERE ENFORCED
Max leverage	20× — `(size × 100) / collateral ≤ 2000`	In-circuit assertion in `open_position` (bare literal `2000u64`, no named constant)
Minimum position size	`size ≥ 100` (smallest `size_usdc`)	In-circuit assertion in `open_position`
Opening fee	0.10% of position size (`size × OPENING_FEE_BPS / 1_000_000`, `OPENING_FEE_BPS = 1_000`)	Computed in-circuit; the net `collateral_after_fee` is bound into the position commitment, then commit-checked on-chain on close/liquidate
Maintenance margin	`MAINTENANCE_MARGIN_BPS = 50_000` (5% of notional, per-million divisor)	Equity below this is liquidatable. Equity computed in-circuit; `maint_margin` and the `equity < maint_margin` assertion run on-chain in `finalize_liquidate`, over commit-verified params and a fresh oracle price
Liquidation reward	`LIQ_PENALTY_BPS = 100_000` (10% of collateral, per-million divisor)	Deterministic, paid to the winning keeper in `liquidate()`. Computed in-circuit from `collateral_usdc`; the amount is pinned by the commit check, and pool solvency (`total_liquidity ≥ reward`) is asserted on-chain
LP withdrawal buffer	10% of total liquidity (`WITHDRAWAL_BUFFER_BPS = 100_000`, per-million divisor)	Computed and asserted on-chain in `finalize_remove_liquidity` (derived from live `pool_state`)
Oracle staleness limit	150 blocks (≈ 5 min at ~2 s/block, `MAX_PRICE_AGE_BLOCKS = 150`)	Asserted on-chain in every price-reading finalize: open, close, liquidate, take-profit, and stop-loss
Compliance validity	≤ 7,776,000 blocks (≈ 180 days at ~2 s/block)	Maximum set at issuance in `zkperp_compliance_v9::issue_compliance`; core asserts `block.height ≤ expires_at` and not-revoked in every trader-function finalize
Price decimals	8 (oracle encoding; $69,000 = 6,900,000,000,000)	Decimal scale set by `zkperp_oracle_v4`, not by a core constant. On-chain PnL uses integer arithmetic with divisor `PRICE_PRECISION = 100_000_000_000` (1e11)

Leo 4.0 Constraints

- **Ternary both-branch evaluation:** Leo evaluates both branches before selecting. All subtraction uses the safe-cap pattern: `let safe_b = b <= a ? b : a; let result = a - safe_b;`
- **finalize arguments always public:** Any value passed to a `finalize` block appears in plaintext on-chain. Position parameters must never be passed to finalize — only commitment hashes and nonces.
- **32-call-per-transaction limit:** Caps bulk operations. Batching is handled at the orchestrator layer.
- **`Mapping::get` in ternary panics:** Always use `Mapping::get_or_use` inside ternary expressions.

- **2M variable limit:** The single-contract design for BTC/ETH/SOL + USDCx MerkleProof verification exceeded the 2,097,152 variable limit. Resolved by splitting into separate per-market contracts (`_core` for market orders, `_orders` for limit orders).

§3 Slot Record Model

ZKPerp uses two distinct slot patterns, both built on Aleo's consume-and-reissue mechanic for refilling records in place. The patterns share an implementation technique but serve different purposes — one is a natural fit for accumulating balances, the other is a deliberate tradeoff that solves a specific UX problem.

LPSlot: Persistent Balance Accumulator

Liquidity provision is fundamentally a single accumulating balance: deposits add, withdrawals subtract, and order doesn't matter. The LPSlot is the natural representation of this — one slot per trader, refilled in place on every state change.

TRANSITION	INPUT (CONSUMED)	OUTPUT (REISSUED)
`add_liquidity()`	LPSlot (existing amount)	LPSlot (existing + new_lp_tokens)
`remove_liquidity()`	LPSlot (existing amount)	LPSlot (existing – burn_amount)

The pattern composes perfectly with the underlying semantics: balance arithmetic is commutative, associative, and order-independent. The LPSlot is essentially a database row with mutable balance, expressed in Aleo's UTXO model. The trader holds exactly one LPSlot for the lifetime of their account.

PositionSlot: Post-Liquidation UX Guarantee

Positions are different. Each position is a distinct economic object with its own entry price, size, direction, and liquidation threshold — they don't compose into a single balance. The slot pattern is applied here for a different reason: to solve the dead-record problem that arises from private-position liquidation.

When a keeper liquidates a position, it consumes its own LiquidationAuth record (which it owns) but cannot consume the trader's PositionSlot (which only the trader can consume). Without mitigation, every liquidation would leave a permanent dead record in the trader's wallet, with no

user-side recovery path. The PositionSlot pattern ensures the trader always has a fresh, usable slot available, refilled either by themselves on voluntary close or by the winning keeper on liquidation.

TRANSITION	INPUT (CONSUMED)	OUTPUT (REISSUED)
`initialize_slots()`	— no inputs —	PositionSlot(0) · PositionSlot(1) · LPSlot — all zeroed
`open_position()`	PositionSlot (is_open: false)	PositionSlot (is_open: true) + LiquidationAuth × 3 → keeper set
`close_position()`	PositionSlot (is_open: true)	PositionSlot (is_open: false, zeroed)
`liquidate()`	LiquidationAuth (keeper-owned, 1-of-N)	PositionSlot (is_open: false, zeroed) → trader

Slot Direction Binding (PositionSlot only)

Within a single market program, each trader holds **two `PositionSlot` records**, distinguished by a `slot_id` field: the `slot_id = 0` record holds longs, the `slot_id = 1` record holds shorts. `open_position` asserts the record's `slot_id` matches the trade direction (`is_long ? 0u8 : 1u8`). A trader can therefore hold at most one long and one short position **per market** concurrently — they cannot, for example, open two BTC longs at once, since both would require that program's single `slot_id = 0` record. Because BTC, ETH, and SOL are deployed as separate programs, each with its own slot records and LP pool, this is *not* a cross-market limit: a BTC long and an ETH long coexist freely. This is a deliberate v1 tradeoff: bounded record count and post-liquidation UX in exchange for one-position-per-direction-per-market. Future versions with orderbook matching will use per-position records to remove even this per-market constraint.

Liquidation UX

On liquidation, the winning keeper's `liquidate()` transition consumes its LiquidationAuth and mints a fresh empty PositionSlot directly to the trader's address. The trader's *old* filled PositionSlot remains in their wallet as harmless dust — any attempt to use it reverts at finalize because `active_position_ids` no longer contains the position ID — but a clean empty slot is immediately available for the next trade. No user action is required after liquidation.

UX RESULT

For the trader: the active slot set is always three records (two PositionSlots, one LPSlot), regardless of trade history. Pending TP/SL orders add one OrderReceipt record each until executed or cancelled. Dead records from past liquidations accumulate as wallet dust but do not require user intervention. For the keeper set: three LiquidationAuth records exist per open position (one per keeper) until the position closes or is liquidated — this scales with protocol activity, not with any individual user. Losing keepers reclaim their stale auths via `burn_liquidation_auth`.

ORPHANED RECORDS — HARMLESS BY DESIGN

Two record types can become orphaned wallet dust as a side effect of asymmetric ownership:

- **Keepers' LiquidationAuths after voluntary close:** when a trader calls `close_position`, only their `PositionSlot` is consumed — the three keepers' `LiquidationAuth` records for that position remain in the keepers' wallets. Any attempt to use one in `liquidate()` reverts at finalize because `active_position_ids[position_id]` has been cleared; keepers clear them with `burn_liquidation_auth`.
- **Trader's OrderReceipt after TP/SL execution:** `execute_take_profit` / `execute_stop_loss` only consume the orchestrator's `ExecTPSLAuth` — the trader's `OrderReceipt` remains as dust. It can be reclaimed via the standalone `burn_order_receipt` transition, which asserts `!pending_orders[order_id]` to guarantee the order really has been settled.

Spent Record Detection

The on-chain `active_position_ids: field => field` mapping is the source of truth for which positions are alive. The frontend queries this mapping and filters wallet records before requesting Shield Wallet to Unshield, ensuring popups are scoped to genuinely active records and dead-slot dust is hidden from the UI.

§4 LP Pool Design & Withdrawal Guard

The ZKPerp liquidity pool is a single-sided USDCx pool. LPs deposit USDCx and act as the direct counterparty to all traders. There is no impermanent loss from token price divergence because the pool holds only one asset.

Withdrawal Guard Formula

Available withdrawable liquidity must account for four components, all enforced on-chain in `finalize_remove_liquidity`:

```
available = total_liquidity
      - long_open_interest      // full notional locked, not netted
      - short_open_interest     // full notional locked, not netted
      - pnl_liability           // max(net_unrealised_pnl, 0)
      - safety_buffer           // 10% × total_liquidity
```

Asymmetric PnL Treatment

TRADER STATE	POOL TREATMENT	LP WITHDRAWAL CAPACITY
Traders up +10%	Real liability — reserved as pnl_liability	Decreases
Traders flat	Zero liability	Unchanged
Traders down −10%	Floored at zero — not a real asset yet	Unchanged — not increased

Why Long OI and Short OI Are Not Netted

Both sides are locked independently and summed — they do not net off. Each market is its own program with its own LP pool, so within a pool the long and short open interest are always the *same* asset; the reason they are not netted is the independence of the positions, not cross-market exposure. Positions settle asynchronously and at the trader's choosing — a winning position can close and be paid before the opposing side moves, is liquidated, or round-trips back to breakeven — so the pool cannot assume one side's losses bankroll the other side's gains. Leverage compounds this asymmetry: a losing position's contribution to the pool is capped at its collateral (it is liquidated at the maintenance margin), while the pool's payout to a winning position is bounded only by available liquidity. A $5M-long / $5M-short book therefore looks balanced but is not a hedge from the pool's perspective; netting would assume an offset the protocol does not enforce and would understate the true capital requirement.

DOCUMENTED TRADEOFF — OI TRUST ASSUMPTION

Because position parameters are private witnesses inside the ZK proof, the per-position sizes needed to compute open interest and net PnL cannot reach `finalize` without becoming public on-chain. The orchestrator (`roles[1u8]`) computes these aggregates off-chain and writes them via `update_pool_state` and `update_net_pnl` . The trust assumption is broader than OI alone: `update_pool_state` overwrites the *entire* `PoolState` from unverified inputs — `total_liquidity` , `total_lp_tokens` , and `accumulated_fees` as well as the two OI figures. A malicious orchestrator can therefore distort LP withdrawal limits, and by inflating `total_liquidity` or skewing `total_lp_tokens` let one LP withdraw a disproportionate share of the pool — harming *other LPs* — or deny service by deflating liquidity so closes and withdrawals fail their solvency check. What it **cannot** do is inflate an individual trader's payout — close, take-profit, stop-loss, and liquidation each recompute the position from its on-chain commitment (§7) — or move more than the pool's real USDCx balance, which the token contract enforces independently. The honest summary: the orchestrator is trusted for *all* pool accounting, not merely OI; trader position payouts and the pool's real balance are not at its mercy. Because `add_liquidity` and `remove_liquidity` already maintain `total_liquidity` and `total_lp_tokens` on-chain, two hardenings are planned (§15): first, restrict `update_pool_state` to the genuinely off-chain aggregates (OI and net PnL); second, place those remaining writes behind a *k*-of-*n* orchestrator quorum — the same on-chain pattern `zkperp_oracle_v4.aleo` uses for prices — reducing trust from a single key to a threshold. This is defense-in-depth, not full verification: the aggregates are still computed off-chain. See §13 for the full architectural analysis.

§5 Liquidation Architecture Updated v28

Liquidation is the hardest unsolved problem in any privacy-preserving perpetual DEX. Positions are private records readable only by their owner — yet a liquidator must be able to act on them without the trader being online. ZKPerp solves this with a **keeper-auth pattern**: every `open_position()` mints **three** `LiquidationAuth` **records** atomically — one to each keeper in the admin-managed `liquidator_set` — and any single keeper can liquidate using its own record without ever touching the trader's slot. The N-keeper set buys **liveness and censorship-resistance, not trust for correctness**: because any single keeper can act, up to N−1 keepers may be offline or decline without halting liquidations, and a keeper that withholds merely cedes the reward to another. No keeper is trusted to act honestly — correctness is enforced on-chain regardless of which keeper submits, since the finalize gates (commitment match, a fresh 2-of-3 oracle price, and a genuine below-maintenance-margin check) make wrongful liquidation impossible. This contrasts with the

orchestrator quorum of §4, which distributes trust for aggregates the chain cannot verify; here the chain already verifies everything, so the keeper set need only provide liveness.

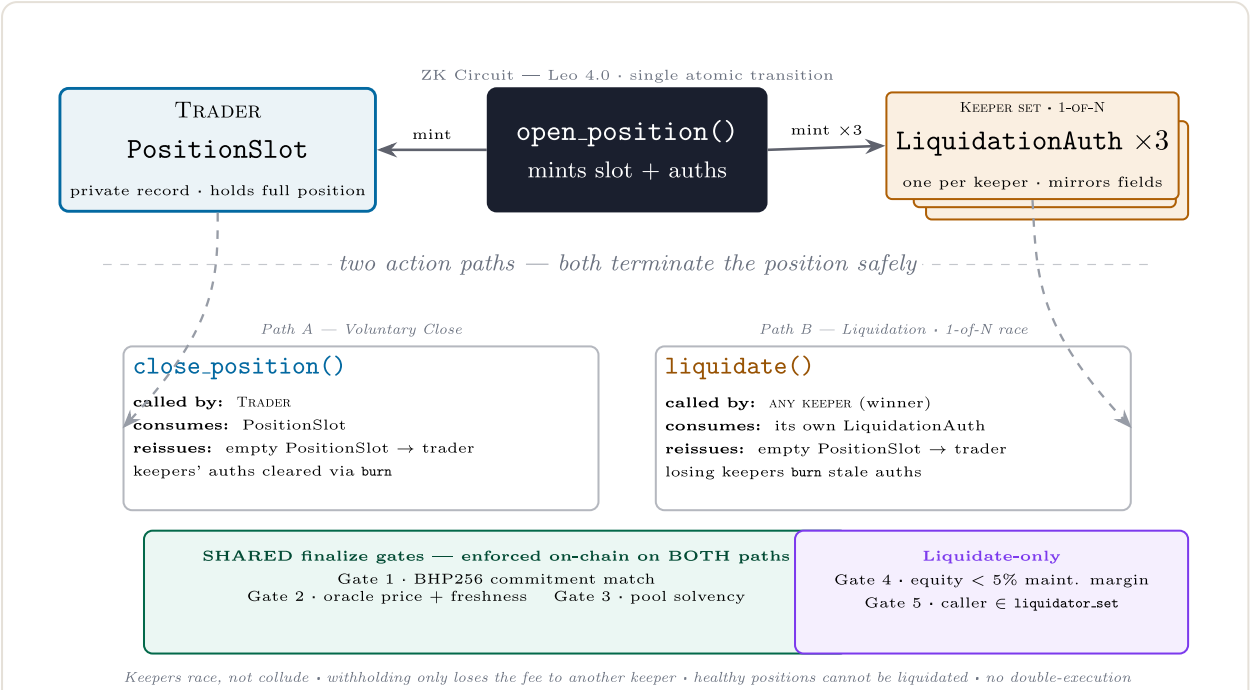

Figure 2. Keeper-auth liquidation. `open_position()` mints the trader's `PositionSlot` and three `LiquidationAuth` records (one per keeper) in one atomic transition. Either the trader closes voluntarily or any single keeper liquidates with its own auth; both paths share the same on-chain finalize gates, with two additional gates on the liquidation path.

Keeper-Auth Architecture (1-of-N)

Every `open_position()` call creates, in a single atomic transaction, the trader's updated `PositionSlot` plus **three `LiquidationAuth` records**:

- **PositionSlot** — owned by the trader. Contains full position data. Only the trader can close or modify it.
- **LiquidationAuth × 3** — one minted to each keeper in `liquidator_set[0u8..2u8]`. Each mirrors the same position fields (`entry_price`, `size_usdc`, `collateral_usdc`, `is_long`, `position_id`). Any one keeper passes its record into `liquidate()`; the trader's slot is never read. Keeper addresses are pinned to `liquidator_set` and verified in `open_position`'s finalize, so a trader cannot substitute colluding addresses.

Liquidation is **1-of-N**: up to N−1 keepers may be offline. The reward goes to whichever keeper settles first, so keepers **race** rather than collude — a keeper that withholds a liquidation simply loses the fee to another. After one keeper wins, the losing keepers clear their now-stale records via `burn_liquidation_auth`.

ACTION	WHO CALLS	OWNS THE RECORD?	WORKS?
`close_position(slot)`	Trader	Yes — it's their slot	✅ Yes
`liquidate(slot)` view key only	Keeper	No — trader owns slot	❌ No
`liquidate(liq_auth)` keeper-auth	Keeper	Yes — keeper owns a LiquidationAuth	✅ Yes

On-Chain Verification (zkperp_core_v30)

Each keeper owns a `LiquidationAuth`, but its fields are *not* trusted by the chain. The strengthened `finalize_liquidate` verifies them against the on-chain commitment, fetches the live oracle price, computes the margin condition itself, and asserts the caller is a current member of `liquidator_set`. A keeper cannot liquidate a healthy position regardless of what it puts in the auth record.

```
// In the transition body (private witnesses)
// entry_price / size_usdc / collateral_usdc / is_long are private inputs;
// position_id is taken from the keeper's auth record.
let position_id: field = auth.position_id;
let recomputed_commit: field = BHP256::hash_to_field(PositionCommit {
    entry_price:     entry_price,
    size_usdc:       size_usdc,
    collateral_usdc: collateral_usdc,
    is_long:         is_long,
    position_id:     position_id,
});

let raw_pnl: i64 = is_long
    ? (exit_price as i64 - entry_price as i64) * size_usdc as i64 / PRICE_PRECISION
as i64
    : (entry_price as i64 - exit_price as i64) * size_usdc as i64 / PRICE_PRECISION
as i64;

let equity_i64: i64 = collateral_usdc as i64 + raw_pnl;     // equity = collateral
+ PnL

// Reward is deterministic, derived from collateral (not caller-supplied).
// collateral_usdc is pinned by the commit check below, so it cannot be inflated.
let liquidator_reward: u64 = collateral_usdc * LIQ_PENALTY_BPS / 1_000_000u64; //
100_000 -> 10%

// In finalize{} — gates enforced on-chain
let stored_id: field      = Mapping::get(active_position_ids, position_id);
assert_eq(stored_id, position_id);                              // replay
protection
let stored_commit: field  = Mapping::get(position_commits, position_id);
assert_eq(recomputed_commit, stored_commit);                    // Gate 1:
commitment
let price_data            = Mapping::get(zkperp_oracle_v4.aleo::oracle_prices,
asset_id);
assert_eq(exit_price, price_data.price);                        // Gate 2a: oracle
match
assert(block.height - price_data.timestamp <= MAX_PRICE_AGE_BLOCKS); // Gate 2b:
freshness
let maint_margin: i64     = (size_usdc * MAINTENANCE_MARGIN_BPS / 1_000_000u64) as
i64;
assert(equity_i64 < maint_margin);                             // Gate 4: underwater
(MARGIN_BPS = 50_000 -> 5%)
let s0: address = Mapping::get(liquidator_set, 0u8);
let s1: address = Mapping::get(liquidator_set, 1u8);
let s2: address = Mapping::get(liquidator_set, 2u8);
assert((caller == s0) || (caller == s1) || (caller == s2));     // Gate 5: caller
in liquidator_set
let pool: PoolState       = Mapping::get(pool_state, 0field);
assert(pool.total_liquidity >= liquidator_reward);              // Gate 3: solvency
```

Verification Gates Across Both Paths

GATE	CLOSE_POSITION	LIQUIDATE	WHAT IT PREVENTS
Gate 1 · BHP256 commitment match	✅	✅	Forged position parameters by either party
Gate 2 · Oracle price + freshness	✅	✅	Trader/keeper supplying a stale or fabricated price
Gate 3 · Pool solvency	✅	✅	Payout / reward exceeding available liquidity
Gate 4 · Equity < 5% maintenance margin	—	✅	Healthy position being liquidated
Gate 5 · Caller ∈ `liquidator_set`	—	✅	Non-keeper calling liquidate
Replay protection · `active_position_ids` atomic clear	✅	✅	Double-close, close+liquidate race, double-liquidation

Liquidation Formula (computed in-circuit, asserted on-chain)

```
pnl = is_long
  ? size × (exit_price − entry_price) / PRICE_PRECISION
  : size × (entry_price − exit_price) / PRICE_PRECISION

equity            = collateral + pnl                        // pnl may be negative
maintenance_margin = size × MAINTENANCE_MARGIN_BPS / 1_000_000   // 50_000 -> 5%
assert(equity < maintenance_margin)                         // reverts if healthy

liquidator_reward  = collateral × LIQ_PENALTY_BPS / 1_000_000    // 100_000 -> 10% of collateral
```

Replay and Double-Execution Prevention

The `active_position_ids` mapping is checked and cleared atomically in both `close_position` and `liquidate`. This prevents double-liquidation, close + liquidate races, and stale TP/SL execution for already-closed positions.

SECURITY RESULT

A keeper can attempt to liquidate any position whose ID is alive, but the chain rejects the call unless (a) the LiquidationAuth fields match the BHP256 commitment stored at open time, (b) the supplied oracle price matches the live `oracle_prices` mapping and is fresh, (c) equity computed from those verified inputs is below the 5% maintenance margin, and (d) the caller is a current member of `liquidator_set`. A malicious keeper can grief by submitting failed liquidations (wasted gas, no state change) but cannot extract value from a healthy position or over-pay itself — the reward is derived deterministically from the verified collateral.

Decentralization: An Honest Open Problem

A real concern is the centralization of liquidators — that ZKPerp alone decides who may liquidate. We do not claim to have solved this. At launch the liquidator set is permissioned, and the protocol can change it. Our defense is not that this is decentralized, but that it is non-custodial and rule-bound: liquidators cannot set prices, invent positions, or redirect funds. The powers that remain centralized are therefore limited to withholding liquidations or competing for rewards — not seizing positions that are still solvent.

Opening the set up further runs into a problem we consider genuinely unsolved, not just unbuilt: detection under privacy. For liquidation to be permissionless, any party must be able to spot a liquidatable position. But ZKPerp positions are private. A keeper can only check whether a position is underwater if the liquidation price is visible to them — and once it is, so are the position's direction, leverage, and approximate entry, since entry can be inferred from the public oracle price at the block where the position opened. Other perpetual exchanges allow permissionless liquidation only because their positions are public in the first place. ZKPerp chooses privacy instead, and treats decentralizing detection as open research.

§6 Advanced Order Types

ZKPerp supports three advanced order types beyond spot market open/close. All follow the keeper-executed pattern: the trader submits order parameters, the orchestrator receives a private auth record, monitors the oracle, and submits execution when the trigger condition is met on-chain.

ORDER TYPE	PURPOSE	LONG TRIGGER	SHORT TRIGGER	FILL PRICE
Limit Order	Open at a better price than market	oracle ≤ trigger	oracle ≥ trigger	Trigger price (limit guarantee)
Take Profit	Close profitable position at target	oracle ≥ trigger	oracle ≤ trigger	Trigger price (limit guarantee)
Stop Loss	Cap downside loss at threshold	oracle ≤ trigger	oracle ≥ trigger	Oracle at execution time (market)

Privacy Model for Orders

Order trigger prices, sizes, and directions are never published on-chain before execution. The only on-chain footprint of a pending order is a boolean in `pending_orders` keyed by a BHP256 commitment hash:

```
order_id = BHP256::hash_to_field(OrderIdInput { trader, nonce, trigger_price })
// On-chain: only this is stored
pending_orders: field => bool   // true = active; null = executed/cancelled
```

An observer can see that some order exists, but cannot determine the trigger price, direction, or size from the commitment hash.

Security: Keeper Cannot Fabricate Execution

The oracle price condition is verified inside `finalize_execute_*` using the live on-chain `oracle_prices` mapping — not a value supplied by the keeper. The keeper cannot claim a trigger was met when it wasn't; the finalize layer will revert if the oracle price does not satisfy the condition at the time the transaction is processed.

§7 Position Commitment Scheme Extended v28

INFLATED-PAYOUT ATTACK — RESOLVED

An earlier version of `close_position` accepted `expected_payout` as an unverified private input — a malicious trader could claim an inflated payout and drain the pool. v26 introduced the BHP256 commitment scheme on the close path. the liquidation path extends it the same way: `finalize_liquidate` recomputes the same commitment from a keeper's `LiquidationAuth` and asserts equality with the stored hash. The commitment is the on-chain source of truth for position parameters on every termination path.

The Attack Vector (Pre-v26)

In the original design, a trader closing a position would supply their own `expected_payout` as a private input. The contract had no way to verify whether this figure was honestly computed — position parameters (entry price, size, direction) were stored only in the private `PositionSlot` record, which the `finalize` block cannot read. A malicious trader could pass an inflated payout and, as long as the pool had sufficient liquidity, drain funds from LPs.

The Fix: BHP256 Position Commitment

At `open_position` time, the ZK circuit computes a single commitment hash from all sensitive position parameters and stores only this hash in the `position_commits` mapping:

```
// Computed privately inside the ZK transition at open_position
commit = BHP256::hash_to_field(PositionCommit {
    entry_price:     entry_price,      // private witness
    size_usdc:       size,             // private witness
    collateral_usdc: collateral_after_fee,
    is_long:         is_long,
    position_id:     position_id,
})

// Stored on-chain — only this single field is visible
Mapping::set(position_commits, position_id, commit)
```

None of the sensitive position parameters — entry price, size, direction, collateral — ever appear in public state. Only their cryptographic hash is visible on the explorer.

Verification at Termination Time

Both termination paths re-supply position parameters as private inputs, recompute the commitment inside the ZK circuit, and assert equality with the stored hash in `finalize`. The source of those private inputs differs by path:

- **close_position**: trader supplies the parameters directly. Three additional gates: oracle price match, oracle freshness, pool solvency. Payout is computed entirely from verified inputs — no user-supplied payout figure is trusted at any point.
- **liquidate**: a keeper supplies the parameters via its `LiquidationAuth` record's fields, which were set at `open_position` time and bound inside the auth record. The recomputed commitment must still match. Additional gates: oracle price match, oracle freshness, equity below maintenance margin (5%), and caller ∈ `liquidator_set`.
- **execute_take_profit / execute_stop_loss**: orchestrator supplies the parameters (with `is_long` and `position_id` taken from the `ExecTPSLAuth` record set at order placement). Same commitment recomputation, plus oracle price match, oracle freshness, an order-pending check, execution-nonce replay protection, and pool solvency. The trigger *direction* is validated on-chain at placement (`place_take_profit` / `place_stop_loss` assert the trigger sits on the correct side of entry); execution does *not* re-check that the oracle price has crossed the trigger — the orchestrator is trusted to fire only when it has.

```
// finalize_close_position — three gates over verified inputs
let stored_commit: field = Mapping::get(position_commits, position_id);
assert_eq(recomputed_commit, stored_commit);          // Gate 1: params verified

let price_data = Mapping::get(zkperp_oracle_v4.aleo::oracle_prices, asset_id);
assert_eq(exit_price, price_data.price);              // Gate 2: price verified
assert(block.height - price_data.timestamp <= MAX_PRICE_AGE_BLOCKS);

let pool = Mapping::get(pool_state, 0field);
assert(pool.total_liquidity >= payout);               // Gate 3: solvency
```

Limit orders (`execute_limit_order`) open a new position rather than close one, so they store a commitment at execution time via `Mapping::set(position_commits, position_id, commit)` — enabling the resulting position to be closed via `close_position` with full commitment verification, and liquidated via `liquidate` with the same protection.

SECURITY RESULT

No party — trader, keeper, or orchestrator — can cause a position to terminate with parameters different from those committed at open time. A malicious trader cannot inflate their payout. A malicious keeper or orchestrator cannot inflate a liquidator reward, liquidate a position at the wrong size, or claim a position is underwater when it isn't. The only on-chain artifacts of a trade are commitment hashes — a block explorer sees the entry commitment hash and the exit oracle price, never the entry price, size, or direction.

§8 Oracle Infrastructure — On-Chain 2-of-3 Quorum

ZKPerp requires tamper-resistant price feeds for BTC, ETH, and SOL. An earlier design used a coordinator process to aggregate 2-of-3 relayer signatures off-chain before submitting to Aleo. The current design eliminates the coordinator entirely: quorum is enforced by **the Leo contract itself** on-chain.

Architecture: zkperp_oracle_v4.aleo

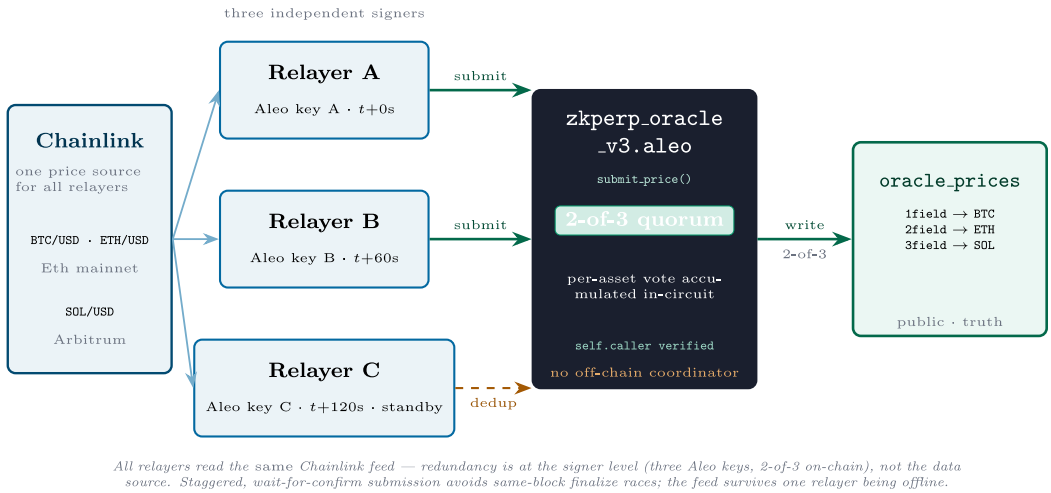

Figure 3. On-chain 2-of-3 oracle quorum. All three relayers read the same Chainlink feeds and submit under their own Aleo keys; redundancy is at the signer level, not the data source. Staggered, wait-for-confirm submission avoids same-block finalize races, and the feed survives one relayer being offline. `zkperp_oracle_v4.aleo` writes `oracle_prices` only when 2-of-3 agree per asset.

Each relayer independently reads **all three** Chainlink feeds — BTC/USD and ETH/USD on Ethereum mainnet, SOL/USD on Arbitrum — and submits under its own Aleo private key. Relayers are staggered and wait for each transaction to confirm before the next, eliminating same-block finalize races; once 2-of-3 have confirmed, the third relayer deduplicates and skips. `self.caller` inside the program is cryptographically verified at the protocol level — it is not a trust assertion. When 2-of-3 relayers submit the same price for the same asset, the Leo program atomically writes it to `oracle_prices`.

Oracle Mappings

MAPPING	KEY	VALUE	PURPOSE
`roles`	u8 (0–3)	address	Registered oracle node addresses
`oracle_prices`	field (asset_id)	PriceData	Committed prices read by core contracts
`price_proposals`	field (asset_id)	PriceProposal	Pending votes accumulating toward quorum

Cross-Program Price Reads

`zkperp_core_v30` reads prices directly from the oracle mapping inside `finalize` — no additional trust assumption. Freshness is enforced as `price_age = block.height - timestamp ≤ 150` (≈ 5 min); age is measured in *blocks* rather than seconds because `finalize` has no trusted wall-clock, only `block.height` — so the relayer writes the current Aleo block height (fetched from the Provable API) into `PriceData.timestamp`, not a Unix time:

```
// In finalize_open_position — cross-program mapping read
let price_data: zkperp_oracle_v4.aleo::PriceData =
    Mapping::get(zkperp_oracle_v4.aleo::oracle_prices, asset_id);
let price_age: u32 = block.height - price_data.timestamp;
assert(price_age <= MAX_PRICE_AGE_BLOCKS);  // 150 blocks ≈ 5 min
let slippage: u64 = entry_price > price_data.price
    ? entry_price - price_data.price
    : price_data.price - entry_price;
assert(slippage <= max_slippage);
```

> SLIPPAGE IS USER-CONTROLLED
>
> `max_slippage` is a **private trader input**, not a protocol constant. It protects the trader from front-run-induced bad fills at the user's chosen tolerance. The protocol does not impose a system-wide slippage cap — a trader who passes `u64::MAX` effectively disables their own slippage check. Position safety against bad fills is the trader's responsibility; the protocol guarantees only that whatever bound the trader sets is enforced on-chain.

Chainlink Feed Configuration

MARKET	ASSET KEY	CHAIN	PROXY ADDRESS	HEARTBEAT
BTC/USD	`1field`	ETH Mainnet	`0xF4030086522a5bEEa4988F8cA5B36dbC97BeE88c`	1h / 0.5%
ETH/USD	`2field`	ETH Mainnet	`0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419`	1h / 0.5%
SOL/USD	`3field`	Arbitrum	`0x24ceA4b8ce57cdA5058b924B9B9987992450590c`	1h / 0.5%

Security Model

- **Single compromised key:** Can delay a round by submitting a divergent price, causing the proposal to reset. Cannot commit a false price alone.
- **Two compromised keys:** Can write an arbitrary price to `oracle_prices`. This is the 2-of-3 trust assumption.
- **All 3 offline:** No price updates; existing on-chain price used; staleness check blocks new position opens after 150 blocks.

> PATH TO STRONGER GUARANTEES
>
> Today, quorum is a *role-check across transactions*: each relayer submits separately, the program authenticates `self.caller` against the registered oracle addresses, and votes accumulate in `price_proposals` until 2-of-3 match. The stronger design is a *single* transaction carrying a co-signed price report whose signatures are verified inside `finalize` — so "2-of-3 agreed" becomes a fact proven by the ZK proof itself rather than reconstructed from a sequence of on-chain writes, removing the multi-transaction accumulation surface (and the same-block races the staggered submission currently works around). Leo already verifies *native Aleo-account* Schnorr signatures (`signature.verify`); what it does not yet expose is the general / secp256k1 signature verification needed here (see §15). When Aleo ships those lower-level primitives, the oracle can move to a single-transaction threshold scheme — e.g. FROST threshold Schnorr, where a *k*-of-*n* set jointly produces one signature against one group key — verified in-circuit, with **no changes to consuming contracts** (`zkperp_core` reads only the `oracle_prices` mapping, not the submission path). One caveat: this hardens how relayers *agree*, not the data itself — all relayers still read the same Chainlink feed, so the feed remains the price-trust root either way. The same primitive would, however, also allow verifying Chainlink's own signed reports directly in-circuit, which would remove the relayer layer entirely — a stronger end state than threshold-signing among relayers.

§9 KYC Compliance Layer

Regulated DeFi needs KYC. Traditional approaches publish user identity on-chain — destroying privacy. ZKPerp's compliance layer enforces KYC at the circuit level: prove you are compliant without revealing who you are.

Architecture

```
REGISTRATION (once per user)
  1. User completes KYC — address added to allowlist
  2. Backend rebuilds depth-10 BHP256 Merkle tree
  3. Admin calls update_root on-chain (only root published — never addresses)
  4. User fetches Merkle proof from backend API
  5. User calls issue_compliance(proof) on zkperp_compliance_v9.aleo
     → ZK proof: "I am a leaf in the tree whose root matches compliance_root"
     → Receives private ZKPerpComplianceRecord in wallet (valid up to 7,776,000
blocks ≈ 180 days)

TRADING (every action)
  6. User passes ZKPerpComplianceRecord as input to open_position / add_liquidity /
etc.
  7. finalize asserts (the typed record IS the membership proof — its root was
     already verified at issuance, so the root is NOT re-checked per trade):
       !Mapping::get_or_use(revoked, caller, false)          // not revoked
       block.height <= cr_expires_at                         // not expired
```

The ZKPerpComplianceRecord

```
record ZKPerpComplianceRecord {
    owner:        address,  // private — only holder can spend
    issued_under: field,    // Merkle root active at issuance
    expires_at:   u32,      // expiry block height (max +7,776,000 blocks ≈ 180
days @ ~2s)
}
```

The record lives in the user's wallet and is re-used on every trade for up to 7,776,000 blocks (≈ 180 days at ~2 s/block) — the maximum value of `expires_at` − `block.height` enforced in `issue_compliance`'s finalize. It is returned unchanged as a transition output, so trading never consumes it. Because `zkperp_core` checks only `revoked` and `expires_at` at trade time — never re-asserting `issued_under` against the live root — an existing record stays valid even after the admin rotates the Merkle root. This is deliberate: re-checking the root per trade would invalidate

every active trader the moment a new user registers. Re-issuance is therefore needed only when the record **expires** or after the holder is **revoked and later reinstated**. The `issued_under` field is retained purely as audit provenance (which allowlist epoch the record was minted under).

Note: the `verify_compliance` transition exposed by `zkperp_compliance_v9.aleo` does perform a full root match, and is available for external programs that want a single-call gate — but `zkperp_core` does not call it, reading `revoked` / `expires_at` directly in its own finalize instead.

Merkle Tree Construction

The allowlist is a depth-10 BHP256 Merkle tree supporting 1,024 users. Only the root — a single field element — is published on-chain. Individual addresses are never revealed.

- **Leaf hash:** `BHP256::hash_to_field(address)`
- **Node hash:** `BHP256::hash_to_field(FieldPair { left, right })`
- **Empty slots:** pre-computed zero hashes (hardcoded, never recomputed)
- **Tree cache:** full tree layers persisted to `tree-cache.json` — server restarts are instant

> IMPLEMENTATION NOTE — LEO HASHING REQUIRED
>
> BHP256 hashes must be computed by the Leo CLI to guarantee byte-identical output to the on-chain snarkVM circuit. The Provable SDK's JavaScript BHP256 implementation does not match Leo 4.0's snarkVM output. The compliance server shells out to `leo run` via a subprocess. This requires a machine with Leo installed — the compliance server runs on the Vultr VPS alongside the oracle bot.

Double-Issuance Protection

To prevent a user from minting multiple records under the same root (which would allow re-use across multiple identities or transfer to non-KYC'd addresses), `issue_compliance` computes a per-issuance nullifier and rejects repeats:

```
nullifier = Poseidon2::hash_to_field(FieldPair { left: leaf, right: computed_root
})
assert(!issued_nullifiers[nullifier])
issued_nullifiers[nullifier] = true
```

The nullifier binds the user's leaf hash to the active Merkle root. A user can issue exactly one record per (user, root) pair. When the admin rotates the root, nullifiers from previous roots no longer match new issuances, so the same user can issue a fresh record post-rotation. Poseidon2 is used here (rather than BHP256) because the nullifier needs only collision resistance, not in-circuit verification — Poseidon2 is faster.

Revocation

Revocation is instant via `revoke_user(address)` — sets a boolean in the on-chain `revoked` mapping. No Merkle tree rebuild is required. The next trade from that address fails immediately. Reinstatement is via `unrevoke_user(address)`.

Regulatory Context

- **KYC enforcement:** Only approved wallets can trade — enforced at the circuit level, not the application layer
- **Sanctions screening:** Instant revocation via `revoke_user` — no tree rotation needed
- **Audit trail:** On-chain root epoch + `/audit` endpoint proves enforcement without exposing user data
- **MiCA/FATF positioning:** The off-chain allowlist maps wallet → identity under legal order, never published on-chain

§10 ZK Dark Pool

ZKDarkPool (`zkdarkpool_v9.aleo`) is a companion protocol deployed on Aleo testnet. It implements institutional-grade batch auction settlement where all bids and asks are sealed until a uniform clearing price is computed and proven on-chain.

The Problem with Continuous Order Books

In a continuous limit order book (CLOB), order placement and cancellation are public events. Large institutional orders must be broken up into small slices to avoid telegraphing intent. Even sliced orders are visible in the order flow data after the fact. MEV bots can detect large buy-side pressure building up and front-run each slice.

Batch Auction Mechanics

```
Phase 1: Order Submission    → All bids/asks submitted as encrypted OrderAuth
records
Phase 2: Window Closes       → No new orders accepted; state frozen
Phase 3: Off-chain Matching  → Operator computes uniform clearing price P*
                               P* = argmax Σ buy_qty(≥P) = Σ sell_qty(≤P)
Phase 4: On-chain Settlement → settle_match() ZK proof enforces price constraints
```

The batching itself — the submission window, its close, and the choice of which orders enter a batch — is coordinated *off-chain* by the operator. The contract holds no batch or window state: it settles individual matched pairs, and the `batch_root` passed into `settle_match` is operator-supplied metadata recorded in each `FillReceipt`, not validated on-chain. What the chain guarantees is per-settlement (price bounds, escrow, nonce, expiry), not the integrity of the batch as a whole.

The Operator-Auth Record Pattern

Previous dark pool designs required off-chain record transfer: the operator needed to receive the user's order record via an encrypted API call. v5 eliminates this using the same pattern as ZKPerp's liquidation architecture:

- `submit_order()` issues an `OrderAuth` record directly to the operator's on-chain address
- `deposit_asset()` issues a `DepositAuth` record to the operator
- The operator receives everything on-chain at order placement time — no JSON files, no ECIES, no API calls

On-Chain Settlement Constraints

Every `settle_match()` execution is a ZK proof that enforces:

- `clearing_price >= sell_order.limit_price` — seller gets at least their minimum
- `clearing_price <= buy_order.limit_price` — buyer pays at most their maximum
- Order nonces not previously consumed — no double-fill
- `deposit_auth.amount >= fill_size` — sufficient escrowed collateral
- Both order expiries respected — `block.height <= buy_expiry` and `block.height <= sell_expiry` at finalize time

A companion transition, `partial_fill()`, settles a match for less than the full order size using the same operator-auth records and the identical finalize gates (`consume_nonces` for double-fill protection and `accrue_fee` for the fee counter), so the same constraints above apply to partial settlements.

Fee Model

Protocol fee is **0.10% (10 bps)** of gross trade value, paid **buyer-side only**. The seller receives the full clearing-price-multiplied amount without deduction. The fee is computed in the transition body as `(fill_size × clearing_price / 1_000_000) × 10 / 10_000` and is reflected in the buyer's `FillReceipt.fee_paid` field; the seller's receipt always shows `fee_paid: 0`.

> ⚠️ CURRENT BEHAVIOR — FEE COMPUTED BUT NOT COLLECTED
>
> As of `zkdarkpool_v9.aleo`, `settle_match` does **not** actually charge the fee on-chain. The buyer is required to hold only `gross_cost` (the constraint is `buyer_token.amount >= gross_cost`, not `gross_cost + fee`), the seller receives exactly `gross_cost`, and the computed `fee` is only written to the buyer's `FillReceipt.fee_paid` and added to the `fee_vault[0u8]` counter. No USDCx is moved to the program, so `fee_vault` is an unbacked accounting counter and `withdraw_fees` decrements it without a corresponding balance. Collecting the fee for real — asserting `buyer_token.amount >= gross_cost + fee` and routing `fee` to the program — is a pending contract revision.

Privacy Model

The dark pool's privacy guarantee is against the *public and peer* surface — the surface front-running needs — not against the operator. At order time, `submit_order` issues an `OrderAuth` record *owned by the operator*, and in Aleo a record's owner decrypts it; the operator therefore sees the full order (size, limit price, direction, asset, and trader identity), which it must in order to compute the uniform clearing price off-chain. What no one — operator or public — can do is settle outside the signed limit prices or move funds, both of which are ZK-enforced on-chain.

WHAT	VISIBLE TO	NOTES
Order size	Operator	Carried in `OrderAuth` (operator-owned → operator decrypts); needed for off-chain matching
Limit price	Operator	Carried in `OrderAuth`; needed to compute the clearing price
Trader address	Operator	`user` field in `OrderAuth` / `OperatorOrderRef`
Order direction	Operator	`OrderAuth` / `OperatorOrderRef`
Asset ID	Operator	`OrderAuth` / `OperatorOrderRef`
Clearing price	Operator-chosen; both parties via `FillReceipt`	Private transition input — never public on-chain (and bounded by both limit prices)
Fill size	Operator-chosen; both parties via `FillReceipt`	Private transition input — never public on-chain
Anything about pending orders, to other traders / MEV bots	Nobody	No order is public before or after settlement — this is the anti-front-running property
Fee-vault total	Everyone	On-chain `fee_vault` mapping
Consumed order nonces	Everyone	Finalize — double-fill prevention
That a settlement occurred	Everyone	On-chain transaction

> OPERATOR TRUST — ANALOGOUS TO THE §13 OI ORCHESTRATOR
>
> The operator is trusted with full order contents (it must decrypt them to match orders), exactly as the perp orchestrator is trusted with open interest. The trust is bounded the same way: the operator cannot settle outside the limit prices, cannot move more than the escrowed amount, and cannot fabricate a fill against a consumed or expired order — all ZK-enforced in `settle_match`. Its residual power is censorship (declining to match an order), mitigated by order expiry and user-initiated `cancel_order` / `cancel_and_refund_asset`. Sealing order contents from the operator as well would require in-circuit matching over committed orders — open research, not implemented as of v9.

Live Testnet Results

SETTLEMENT	CLEARING PRICE	VOLUME	STATUS
Batch #1 (BTC)	$49,500	1.0 BTC	✅ On-chain confirmed

FURTHER READING

The dark pool is maintained as a standalone subproject alongside ZKPerp. For full contract internals — batch auction proof construction, OrderAuth and OperatorOrderRef record layouts, settlement transition flow, and operator runbooks — see zkperp-darkpool/README.md on GitHub.

§11 Concentrated Liquidity AMM

Alongside the perpetuals protocol, the project maintains a **standalone spot DEX**: `zkperp_amm`, a Uniswap v3-style Concentrated Liquidity AMM for the USDCx/ALEO pair with a 0.3% fee tier. It is a separate subproject and is *not* the perpetuals liquidity engine — perps liquidity is the slot-record LP pool with the withdrawal guard (see LP Pool Design). The AMM exists to bootstrap spot USDCx/ALEO depth and to exercise the `credits.aleo` integration end-to-end. LP positions are private records; pool state (price, tick, liquidity) is public — full AMM privacy is architecturally impossible because price discovery requires public state.

Both legs of every transition move real assets. USDCx flows through `test_usdcx_stablecoin.aleo`'s private token interface; native ALEO flows through `credits.aleo`'s `transfer_private_to_public` and `transfer_public_to_private` functions. `mint_position` and `swap_sell` consume a private `credits.aleo::credits` record from the caller; `burn_position` and `swap_buy` mint one in return, so both legs settle as real on-chain transfers rather than a mapping counter.

Key Parameters

- Tick spacing: 60 (matching Uniswap v3 0.3% pools)
- sqrt price stored as Q64 fixed-point: `sqrt(price) × 2^64` (u128)
- Up to 4 tick crossings per swap (unrolled in Leo — no loops in finalize)
- Fee accrual via `fee_growth_global` pattern

Swap Math

```
// swap_buy (USDCx → ALEO, price decreases)
sqrtAfter = L × Q64 × sqrtBefore / (L × Q64 + amountIn_net × sqrtBefore)
amountOut  = L × (sqrtBefore − sqrtAfter) / Q64

// swap_sell (ALEO → USDCx, price increases)
sqrtAfter  = sqrtBefore + amountIn_net × Q64 / L
amountOut  = L × (sqrtAfter − sqrtBefore) × Q64 / (sqrtAfter × sqrtBefore)
```

These closed forms are evaluated *client-side by the frontend*, and the resulting amounts, final price, and step data are submitted by the **trader**, who signs the transaction with their own wallet. There is no backend or privileged operator — the caller is the trader. Since the trader chooses and signs these inputs (and nothing compels them to use the frontend's honest values), the contract treats them as untrusted and **verifies the result against the curve** in `finalize`: output paid out is capped at the curve value for the supplied price move, and input collected is floored at the curve minimum, both computed from the on-chain liquidity. A trader therefore cannot extract more (or pay less) than the price move justifies, regardless of what they submit.

Leo 4.0 Constraints in the AMM

CONSTRAINT	SOLUTION
No loops in `final{}`	4-step unrolled tick crossing
Both ternary branches evaluated	max-min pattern for all subtractions
`Mapping::get` in ternary panics	`Mapping::get_or_use` everywhere
Q64 values overflow JS Number	Divide by 2^32 via BigInt before converting to float
Nested ternaries in final{} rejected	Flatten all conditionals to intermediate variables

SWAP VERIFICATION — TRADER-SIGNED, CONTRACT-VERIFIED V6

The AMM has **no orchestrator or backend**: the trader's own wallet builds and signs every swap, so all swap parameters (amounts, final price, per-step data) are trader-supplied and untrusted. The contract is therefore the *sole* arbiter and verifies every swap against the curve in `finalize`. Up to four unrolled tick-crossing steps plus a terminal segment are each checked the same way — output capped at the curve value and required input floored against it, both derived from the on-chain liquidity (outputs floored, inputs ceiled, no tolerance band). The final tick is pinned to the final price via an in-circuit `sqrt_ratio_at_tick`, and per-step fees accrue through the `fee_growth_global` pattern. A trader cannot extract more (or pay less) than the price move justifies, whatever they submit. LP `mint` / `burn` and swap amounts are public `finalize` arguments. The build is demo-grade and not yet validated on-chain (see Implementation Status below).

IMPLEMENTATION STATUS — HARDENED, PENDING ON-CHAIN VALIDATION V6

`zkperp_amm_v6.aleo` verifies every swap and LP action against the curve: the terminal segment and each tick-crossing step cap output and floor required input against on-chain liquidity, the final tick is pinned to the final price via an in-circuit `sqrt_ratio_at_tick`, `mint_position` / `burn_position` deposits and payouts are derived from liquidity, and per-step fees accrue through `fee_growth_global`. The threat model is client-side proving: the trader/LP supplies every public input, so the contract treats them as untrusted and is the sole arbiter.

Two limitations gate production. Amount math runs in inline `u128`, so the contract is provably safe only inside a bounded envelope — position ticks within **±88,440** and per-position liquidity **≤ 2^{55}**; outside it, a multiply aborts the proof or reverts the transaction (funds are never lost — only liveness is limited). Lifting the envelope to the full ±443,636 tick range requires a verified 256-bit `mul_div` (the equivalent of Uniswap's `FullMath.mulDiv`). The build is demo-grade and has not been compiler-verified or run through its full test suite in this environment; the multi-tick path and rounding-tolerance calibration await devnet validation and a third-party audit. Until then the AMM is a research prototype, not a usable venue.

PRIVACY NOTE

Swap amounts on the AMM are public — they appear as finalize arguments. For large private trades, use `zkdarkpool_v9.aleo` (batch auctions) instead. The AMM and dark pool serve complementary use cases: the AMM for continuous liquidity provision, the dark pool for institutional sealed-bid settlement.

FURTHER READING

The AMM is maintained as a standalone subproject alongside ZKPerp. For full contract internals — Q64 fixed-point math, tick crossing, the credits.aleo integration, and how the frontend pre-computes swap steps — see zkperp-amm/README.md on GitHub.

§12 USDCx & Cross-Chain Bridge

ZKPerp uses USDCx (`test_usdcx_stablecoin.aleo`) — the official Aleo testnet stablecoin issued by Provable — as its collateral and settlement token. All collateral flows use real private token transfers.

Bridge Flow: ETH → Aleo

```
Step 1 — Sepolia:  executeBridgeTransfer() on HumanityLinkVault.sol
Step 2 — xReserve: Circle bridge (0x008888878f94...) — 5-15 min settlement
Step 3 — Aleo:    USDCx appears in Shield Wallet, ready to trade

Live test: ~81 USDC bridged successfully in under 15 minutes
```

USDCx in Transactions

All ZKPerp transitions that move USDCx use `transfer_private_to_public` with a `MerkleProof[2]` (2-element array) for the USDCx freeze list. This MerkleProof verification is the primary reason per-market contracts approach Aleo's 2M variable limit — it expands the circuit significantly and is the root cause of the core/orders contract split.

ATOMICITY GUARANTEE

If the token transfer fails for any reason, the entire position operation reverts. Collateral is transferred to `self.address` (the Leo program address) — not to the orchestrator — ensuring the program controls the funds at all times.

§13 OI Tradeoff: Privacy vs On-Chain State

A natural question is whether `update_pool_state` should derive open interest from on-chain position data rather than trusting orchestrator inputs. This section addresses that question with full architectural transparency.

Why OI Cannot Be Derived On-Chain Without Leaking Sizes

The chain stores only:

- `active_position_ids` — which position IDs exist (BHP256 hashes, no sizes)
- `position_commits` — hashes of position params (opaque by design)
- `pool_state` — whatever was last written there

There is no way to sum `size_usdc` values from `active_position_ids` because sizes were never stored on-chain — they are encrypted inside `PositionSlot` records. To update `long_open_interest` atomically inside `open_position`'s finalize, those values would need to be declared as public finalize inputs, making them visible in plaintext on the explorer. We verified this empirically — when we included OI updates in finalize, trade sizes and directions appeared as `u64.public` and `boolean.public` in every transaction.

The Three Options

OPTION	OI ON-CHAIN	PRIVACY	TRUST
Store OI in finalize	✅ Atomic	❌ size/is_long leak publicly	None needed
Orchestrator calls update_pool_state	✅ Visible	✅ Fully private	Orchestrator honest
Compute off-chain via view key, show on frontend	❌ Not on-chain	✅ Fully private	None needed

There is no fourth option that provides private trades, on-chain OI, and no trust assumption simultaneously. This is a fundamental constraint of Aleo's execution model — and of any ZK execution model where private witnesses cannot flow into public state.

Comparison to Other Protocols

PROTOCOL	POSITION PRIVACY	OI SOURCE	FUND SAFETY
GMX	❌ Fully public	Atomic, on-chain	Smart contract
dYdX v4	❌ Fully public	Off-chain engine	Smart contract
ZKPerp	✅ Fully private	Orchestrator (view key)	Commit hash

BOUNDED TRUST ASSUMPTION

The orchestrator writes `long_open_interest` and `short_open_interest`, which affect LP withdrawal limits via the safety-buffer check in `remove_liquidity`. As detailed in §4, `update_pool_state` currently also lets it overwrite `total_liquidity`, `total_lp_tokens`, and `accumulated_fees`, so the practical write surface is *all* pool accounting plus net PnL — but **not** oracle prices, which are governed separately by the 2-of-3 quorum in `zkperp_oracle_v4.aleo`. What the orchestrator cannot do is inflate an individual trader's payout (close, TP, SL, and liquidation each recompute the position from its on-chain commitment, §7) or move more than the pool's real USDCx balance, which the token contract enforces. Only OI is genuinely architecture-bound (sizes are private witnesses, per the table above); `total_liquidity` and `total_lp_tokens` are already tracked on-chain and need not be orchestrator-writable, so scoping `update_pool_state` down to OI + net PnL and placing those behind a quorum is a planned hardening (§15). No existing privacy-preserving perpetual DEX on any chain derives OI atomically from private position data, because doing so requires making that data public.

§14 Known Limitations

ZKPerp is testnet software. The following limitations are documented in full.

ITEM	STATUS	NOTES
USDCx Token Integration	✅ Resolved	All collateral flows use real private token transfers via test_usdcx_stablecoin.aleo
Record Accumulation	✅ Resolved	Slot model: wallet holds exactly 3 records per market program (2 PositionSlots + 1 LPSlot), forever
Dual-Record Liquidation	✅ Resolved	LiquidationAuth pattern fully implemented and live
Single Oracle Point of Failure	✅ Resolved	On-chain 2-of-3 quorum via zkperp_oracle_v4.aleo — no coordinator
Inflated Payout Attack (close)	✅ Resolved	BHP256 position commitment scheme in v26 — payout fully verified in finalize
Unverified Liquidation Parameters	✅ Resolved	v28 — LiquidationAuth fields commitment-checked; margin asserted on-chain in finalize from commit-verified params + a fresh oracle price
KYC / Compliance	✅ Resolved	zkperp_compliance_v9.aleo — BHP256 Merkle allowlist, circuit-level enforcement
ZK Dark Pool	✅ Resolved	zkdarkpool_v9.aleo — batch auction, live on testnet
Concentrated Liquidity AMM	⚠ Prototype	zkperp_amm — Uniswap v3-style USDCx/ALEO, a standalone spot subproject. `zkperp_amm_v6.aleo` verifies every swap against the curve (output capped, required input floored, final tick pinned); demo-grade, pending on-chain validation and a third-party audit (see §11).
2M Variable Limit (single contract)	✅ Resolved	Split into _core (market orders) + _orders (limit orders) contracts
Trusted pool-state aggregation	⚠ Documented Tradeoff	OI itself is architecture-bound (private sizes cannot enter public state — §13); the broader `update_pool_state` write surface (liquidity, LP tokens) is fixable — hardening planned in §15
Single Orchestrator	⚠ Open	One wallet (`roles[1u8]`) controls TP/SL order execution and pool-state aggregation. Liquidation is *not* orchestrator-run — it is 1-of-N keepers from the admin-managed `liquidator_set` (§5). Multi-orchestrator rotation and the §15 pool-state hardening are planned.
Non-Transferable Compliance Admin	⚠ Open	The compliance admin is fixed at deploy time — `zkperp_compliance_v9.aleo` has no `set_admin` (the admin is written once in the constructor). A future version will add multisig / governance-gated admin rotation.

AMM Swap-Invariant Verification	⚠ Pending validation	`zkperp_amm_v6.aleo` binds every swap to the curve on both legs — output capped, required input floored, final tick pinned via in-circuit `sqrt_ratio_at_tick`, deposits/payouts derived from liquidity — across up to four tick-crossing steps plus a terminal segment. Not yet validated on-chain: inline u128 math caps the safe envelope (ticks ±88,440, liquidity ≤ 2^{55}), and a verified 256-bit `mul_div`, rounding calibration, and audit remain pending.
Dark Pool Deposit-to-Order Binding	⚠ Documented Tradeoff	Sell-side deposits bound to seller by address + asset_id, not by order nonce. Operator picks which deposit settles which order. Stronger nonce-binding planned.
No Funding Rate	⚠ Open	Flat borrow rate model. Dynamic funding rates planned for mainnet.
No Partial Close	⚠ Open	Positions must be closed in full. Partial close is architecturally supported by the slot model.
Leo Ternary Both-Branch Evaluation	Language	Both branches evaluate before selection — safe-cap pattern used throughout
32-Call-Per-Transaction Limit	Language	Caps bulk operations. Batching handled at orchestrator layer.

§15 Roadmap

✅ Completed

- Core perpetuals: open / close / liquidate with USDCx token integration
- Slot model — 3 records per market program (2 PositionSlots + 1 LPSlot)
- Dual-record liquidation architecture
- 2-of-3 Chainlink oracle relay — on-chain quorum (`zkperp_oracle_v4.aleo`)
- Limit orders / TP / SL — keeper executed, ZK trigger verification
- LP pool with on-chain withdrawal guard
- BTC, ETH, SOL markets — separate programs per market (privacy model constraint)
- Position commitment scheme — closes the inflated-payout vector (v26)
- Commitment verification on the liquidation path — margin asserted on-chain in finalize (v28)
- KYC compliance — BHP256 Merkle allowlist, circuit-level enforcement

- ZK Dark Pool — batch auction, uniform clearing price
- Concentrated Liquidity AMM — Uniswap v3-style USDCx/ALEO (`zkperp_amm_v6.aleo`), demo-grade
- Portfolio, compliance, and status pages

📋 Next

- Verifiable performance proofs (ZK) — copy trading without revealing trades
- Privacy-preserving leaderboard (opt-in)
- Improved website UX — trading interface, onboarding, wallet/Shield connection
- Multi-orchestrator rotation — reduce single-orchestrator trust
- Pool-state write hardening — restrict `update_pool_state` to OI / net-PnL, then move those behind a *k*-of-*n* quorum
- Admin controls — pause, fee update, role rotation
- On-chain secp256k1 oracle signature verification
- Shared core library (`zkperp_lib`) — commit hashing, PnL, margin, compliance gate, oracle read; markets become thin wrappers
- Standardize fixed-point conventions across core / dark pool / AMM
- AMM hardening — devnet validation, 256-bit `mul_div` , rounding calibration

📋 Phase 2 — Testnet Mature

- Trustless ZK PnL aggregation
- Dynamic funding rates — longs pay shorts when long OI > short OI
- Partial close + order updates
- LP withdrawal cooldowns
- Privacy copy trading
- Explore per-position records — removes one-position-per-direction-per-market; orderbook prerequisite
- Compliance admin rotation — multisig / governance-gated `set_admin`

🔬 Phase 3 — ZK Orderbook

- ZK CLOB on the dark-pool primitives (sealed orders, operator-auth, `settle_match` / `partial_fill`)
- In-circuit order matching — seal order contents from the operator (open research)
- Frequent batch auctions — MEV / front-running resistance
- Price-time priority via ZK proofs

💸 Phase 4 — Money Markets & Vaults

- Private borrowing / lending — perp commitment + 1-of-N keeper pattern for loan liquidation, index-snapshot interest accrual
- ERC-4626-style tokenized vaults — standard yield-bearing shares over the LP pool / AMM (Aleo tokenized-vault ARC)
- Verifiable vault solvency — provable assets / shares without exposing per-depositor flows (FROST + Pedersen)
- LP-collateralized leverage — vault shares → lending → perps (ZKPrime direction)

🔐 Phase 5 — Security & Audit

- Formal Leo verification — circuit soundness + in-circuit ↔ finalize boundaries
- Verify key properties — commitment-binding, compliance nullifier non-reuse, oracle quorum
- Third-party ZK circuit audit — soundness + completeness, covering lending and vault paths
- AMM swap-invariant audit
- Bug bounty program
- Extended oracle quorum (5-of-7)

🚀 Phase 6 — Mainnet & Go-to-Market

- Guarded mainnet — position / notional caps, gated on audit sign-off
- Full Aleo mainnet deployment
- Liquidity bootstrapping — market-maker partnership + LP incentives
- Institutional lane — KYC-gated dark-pool settlement; MiCA / FATF positioning
- Distribution — frontend / venue partner integration
- Native Chainlink / Pyth integration (if Aleo-supported)
- Cross-margin / portfolio margin
- Mobile application
- DAO governance

§16 Security Properties Updated v28

On-Chain Guarantees

PROPERTY	MECHANISM	WHERE ENFORCED
No inflated close payout	BHP256 commitment recomputed + asserted against stored hash	`finalize_close_position`
No forged liquidation parameters	BHP256 commitment recomputed from `LiquidationAuth` fields + asserted against stored hash	`finalize_liquidate`
No healthy position liquidated	Equity computed in-circuit from verified params and verified oracle price; `equity < maintenance_margin` asserted (maintenance margin = 5% of notional)	`finalize_liquidate`
No double-close or double-liquidation	`active_position_ids[id]` checked and cleared atomically	`finalize_close_position`, `finalize_liquidate`
Exit / liquidation / TP / SL price must match oracle exactly	`assert_eq(supplied_price, oracle_prices[asset_id].price)` + freshness assertion	`finalize_close_position`, `finalize_liquidate`, `finalize_execute_take_profit`, `finalize_execute_stop_loss`
Entry price bounded by trader-set slippage	`assert(|entry_price - oracle_price| <= max_slippage)` with `max_slippage` as private trader input	`finalize_open_position`
No unauthorized oracle update	Role check: only registered nodes can submit; 2-of-3 required	`finalize_submit_price`
No LP over-withdrawal	Full guard formula checked; tx reverts if breached	`finalize_remove_liquidity`
No stale price trade	`block.height - timestamp <= 150`	`finalize_open_position`, `finalize_close_position`, `finalize_liquidate`, `finalize_execute_take_profit`, `finalize_execute_stop_loss`
Compliance gating	`revoked` and `expires_at` asserted (`issued_under` retained as provenance, not re-checked at trade time)	Every trader finalize function

Dark pool price constraints	buy_limit ≥ clearing ≥ sell_limit, ZK enforced	`finalize_settle_match`

What the Keeper / Orchestrator Cannot Do

- Steal trader funds — payouts and rewards are computed from the verified commitment hash, not a bot-supplied figure
- Liquidate a healthy position — equity computed in-circuit from BHP256-verified params; the maintenance-margin threshold and the `equity < maint_margin` assertion run on-chain in finalize, against a fresh oracle price
- Forge LiquidationAuth fields — recomputed commitment must equal the hash stored at `open_position`
- Execute a TP/SL at a fabricated or stale price — the execution price must equal the fresh oracle price (`finalize_execute_take_profit` / `finalize_execute_stop_loss`). Note: the trigger *direction* is asserted at placement, but execution does not re-check that the price crossed the trigger — the orchestrator is trusted to fire only when it has
- Execute an order for a closed position — `active_position_ids` check
- Read the trader's `PositionSlot` — the orchestrator does not own this record

Privacy Guarantees

- Position size, entry price, leverage, PnL, liquidation level — encrypted in private records; readable only by the position owner's viewing key
- Order trigger prices — sealed behind BHP256 commitment hashes; never published before execution
- LP balances — stored in private `LPSlot` records; only the LP can see their own position
- KYC identity — off-chain allowlist; only the Merkle root is on-chain; individual addresses are never revealed
- Dark pool order size and limit price — encrypted in `OrderAuth` records; never published before settlement

> AUDIT STATUS
>
> ZKPerp has not been formally audited. This is testnet software intended for development and demonstration purposes. Do not use it with real funds. A third-party security audit is planned before any mainnet deployment. Responsible disclosure: contact the team via GitHub issues.

PROTOCOL

ZKPerp — Privacy-First Perpetuals on Aleo

zkperp_core_v30.aleo

zkperp_core_eth_v29c.aleo

zkperp_core_sol_v29c.aleo

zkperp_oracle_v4.aleo

zkperp_compliance_v9.aleo

zkperp_amm_v6.aleo

zkdarkpool_v9.aleo

test_usdcx_stablecoin.aleo

LINKS

zk-perp.vercel.app

github.com/hwdeboer1977/ZKPerp

Provable Explorer

Shield Wallet

AUTHOR

Henk-Wim de Boer

Discord: @lupo1977

aleo1d9es6d8kuzg65dlfdpx9zxchcs...

ZKPerp · MIT License · Aleo Testnet · v30 · May 2026