

Despot: A Parimutuel Trading Protocol with Program-Derived Positions on Solana

Jotaro Yakazu

October 19, 2026

Abstract

Despot is a parimutuel trading protocol implemented as a single Solana program. It operates a fixed set of independently priced assets. Each asset is issued against a deterministic integer bonding curve that mints units on purchase and burns them on sale, and the lamports paid in are held in a reserve account owned by the program. That reserve backs the asset’s weight in contests. On a fixed slot schedule the program halts all trading, derives a 256-bit random value from a pre-committed seed chain combined with a Solana slot hash, and resolves a set of contests, each between one defending asset and one or more challenging assets selected by their holders’ weighted allocation votes. The losing asset’s units held in positions are transferred in full to the holders of the winning assets, so payouts on the winning side are funded, in units, by holders on the losing side, with no operator take on the transfer. The cycle repeats every 4 423 slots at the launch settings, about 16 minutes at the current slot time; both figures may differ at the time of reading (see Section 6).

The central design decision is that an asset unit is not an SPL token. It is a balance stored in a program-derived account, one per (owner, asset) pair, writable only by the program. This gives the unit the properties a market participant expects of a fungible token, continuous two-sided liquidity at a published price, while the program retains control over when the unit may trade, what fraction of a balance may be sold, and to whom the balance moves when a contest is lost. We are not aware of a prior design that uses program-derived accounts to give a freely traded, curve-priced asset an enforced trading calendar and an enforced partial lock.

This paper specifies the account model, the pricing curve and its exact integer arithmetic, the fee schedule, the round clock, the commitment lock, contest weight and target selection, the randomness pipeline, contest resolution with constant-time supply transfer and accumulator accounting, the uniform-price settlement auction, the invariants the program maintains, its liveness fallbacks, delegated signing and custody, and independent verification. All value arithmetic is unsigned integer arithmetic; there is no floating point anywhere a lamport is computed.

The deployment specified here is a beta, built to measure demand for the category before the design is generalised. Its constraints are deliberate, and the designs intended to replace them are set out at the end (see Section 16).

Contents

1 Terminology

2 Account Model

3 Units as Program-Derived Positions

3.1 Limits of an SPL mint · 3.2 The position account · 3.3 What the program controls · 3.4 Comparison

4 The Pricing Curve

4.1 Reserve function · 4.2 Settlement · 4.3 Reserve invariant · 4.4 Evaluating R in 128-bit arithmetic ·
4.5 Inverting the curve for a lamport budget · 4.6 Rounding · 4.7 Launch parameters

5 Fees

5.1 Base rate and terminal ramp · 5.2 Purchase · 5.3 Sale · 5.4 Fee-free operations · 5.5 Execution guards ·
5.6 Fee vaults

6 The Round Clock

7 The Commitment Lock

8 Contest Weight and Target Selection

8.1 Collateral as contest weight · 8.2 Occupied assets · 8.3 Why linear weight · 8.4 Allocation votes ·
8.5 Target selection

9 Randomness

9.1 Seed chain · 9.2 Capture and reveal · 9.3 Draw stream · 9.4 Withheld reveals

10 Contest Resolution and Capture

10.1 Resolving a contest · 10.2 Sequencing within a round · 10.3 Capture ·
10.4 Capture pools and constant-time accounting · 10.5 Claims

11 The Settlement Auction

12 Invariants

13 Liveness and Failure Modes

13.1 Permissionless crank · 13.2 Voided rounds · 13.3 Dead-man fallback · 13.4 Bounded compute

14 Delegated Signing and Custody

14.1 Program-controlled balance · 14.2 Session keys · 14.3 Derived destinations · 14.4 Emergency exit ·
14.5 Custody

15 Verification

15.1 Outcome proofs · 15.2 Differential testing · 15.3 Verifiable build

16 Scope of This Version

16.1 A deliberately constrained first deployment · 16.2 Known limitations and directions

1 Terminology

The product presents the protocol through a geopolitical theme. This paper uses general terms throughout; the table maps them to what a player sees. Program identifiers keep their original names where they are quoted as code.

Table 1: Terms used in this paper, with the product’s presentation and the program identifier.

Term in this paper	Product presentation	Program identifier
asset	country	Country
unit, token	share	Position.shares
contest	battle	resolve_battle
resolution window	war	war stage
contest weight	army	army
capture	conquest	conquer
captured supply, capture pool	loot, loot pool	Slot, pool
settlement auction	loot sale	queue_sell, close_window
commitment lock	half-lock	locked
allocation vote	vote for a target	set_vote
sealed buffer	sealed stage	lockdown stage
reserve	reserve	Reserve

2 Account Model

The protocol is one Anchor program. All state lives in accounts whose addresses are derived from fixed seeds and the program id (program-derived addresses, PDAs). A PDA has no corresponding private key: no external party can sign for it, so its contents change only through the program’s own instructions.

Table 2: Program-derived accounts. Seeds are hashed with the program id, so no account has a private key.

Account	Seeds	Count	Contents
Game	["game"]	1	round schedule, fee configuration, seed-chain head, void counter
Round	["round"]	1	captured slot hash, seed, Θ , contests, outcomes
History	["history"]	1	receipts of the last 64 rounds
Fee vault	["vault"], ["vault", i]	$1 + n$	trading fees only, never mixed with a reserve
Asset	["country", idx]	per asset, at most 16	minted supply S , allocation tally, capture pools, generation
Reserve	["reserve", idx]	per asset	the asset's lamports; balance equals $R(S)$
Holder	["player", owner]	per wallet	withdrawable lamport balance, delegated key and its limits
Position	["position", owner, idx]	per wallet and asset	shares = free + locked, vote, captured supply

The model has four global accounts (game, round, history and the first fee vault) plus up to 63 further fee vaults, two accounts per asset, one account per holder, and one position account per holder per asset held. The position account is the token: it records how many units of one asset a wallet holds and in what state. The reserve account holds the lamports backing those units, and the fee vaults hold trading fees, so asset collateral and protocol revenue never share an account.

The asset set is configuration within a compile-time ceiling of $N_{\text{MAX}} = 16$. Every fixed-size array (allocation tallies, capture-pool slots) is sized to that ceiling, and every loop is bounded by the active set.

3 Units as Program-Derived Positions

An asset unit presents to the holder as a fungible token and to the program as a ledger entry under its exclusive write authority. This section sets out why the unit is not an SPL token, how the position account reproduces token semantics, and what the program gains by owning the account.

3.1 Limits of an SPL mint

An SPL token resides in a token account controlled by its holder through the Token program. The holder can transfer it, or route it through any venue, at any slot, independently of the issuing program. The protocol requires four properties an SPL token cannot provide:

1. **An enforced trading calendar.** Contest outcomes are public on chain as soon as they resolve. Any venue open during the resolution window would let the fastest chain reader trade on known outcomes. Every market must therefore close and reopen at fixed slots, on every venue simulta-

neously. For an SPL token the only available control is a freeze authority, which acts on all holders at once, cannot be scoped to a phase, and constitutes a centralised kill switch.

2. **A partial lock.** A fixed fraction of each purchase must be unsellable until the end of the round in which it was bought. A token account carries a single balance and cannot withhold part of itself.
3. **No external liquidity.** If units could leave the program, an external pool could price and trade them during the resolution window, reopening the information leak the calendar closes.
4. **Constant-time supply transfer.** When an asset loses a contest, every unit of it held in positions moves to the winners within the same instruction. Moving SPL balances would require writing every holder's token account, an unbounded loop that no transaction can execute.

3.2 The position account

Each (owner, asset) pair has one account at the address derived from ["position", owner, idx]. Its principal fields:

Table 3: Fields of the position account.

Field	Width	Meaning
shares	u64	units held; invariant shares = free + locked
free	u64	units currently sellable
locked	u64	units under the commitment lock
locked_epoch	u32	round of the purchase that filled locked
generation	u32	the asset generation this balance belongs to
vote_target, vote_weight	u8, u64	allocation vote and its weight
z[16], pts[16], debt[16]	u128 each	recorded captured supply and accumulator bookmarks
payout_owed	u64	settled auction proceeds, in lamports

Circulating supply S is stored once, on the asset account, and all units of an asset are interchangeable. A unit is therefore fungible, publicly observable, priced by a published function of S , and redeemable against the reserve whenever trading is open. These are the economic properties of a token.

3.3 What the program controls

Timing. The phase of a round is a pure function of the slot, fixed when the round opens:

$$h = t_i + \tau_{trade}, \quad L = h + \tau_{sealed}, \quad W = L + \tau_{res}, \quad E = W + \tau_{auc} = t_i + \tau_{round} \quad (1)$$

where t_i is the opening slot of round i . `buy` and `sell` succeed only for $t_i \leq \text{slot} < h$; `queue_sell` succeeds only for $W \leq \text{slot} < E$ and only on an asset captured in that round. Every other slot rejects

every trade on every asset. Since no unit exists outside the program, there is no venue to route around the calendar.

Quantity. `sell` and `queue_sell` draw only from `free`. The lock is held as two counters rather than a list of purchases, so the check is constant cost.

Destination. The program exposes no transfer instruction. A unit changes owner in exactly two ways: it is burned into the curve by a sale, or it is transferred on capture. Sale proceeds are credited to the owner’s holder account, whose address is derived from the stored owner and never from a caller-supplied account.

Constant-time capture. Each asset account carries a generation counter g_X , and each position records the generation under which it was written. The effective balance of the position of owner o on X is

$$\text{bal}(o, X) = \begin{cases} \text{shares}_o & g_o = g_X \\ 0 & g_o \neq g_X \end{cases} \quad (2)$$

Capturing X increments g_X and credits the units of X held in positions to the winners’ capture pools. Every position on X evaluates as empty the next time it is touched, and no position account is visited during resolution. The cost of a capture is independent of the number of holders.

3.4 Comparison

Table 4: An SPL token compared with a program-derived position.

Property	SPL token	Program-derived position
Holder can transfer freely	yes	no transfer instruction exists
Trading halts on a schedule	no (freeze is global)	yes, by slot, on every asset
Part of a balance can be locked	no	yes, <code>free</code> and <code>locked</code>
Can trade on external venues	yes	no
Cost of transferring a losing asset’s supply	one write per holder	one counter increment
Price formation	whichever venue trades it	the curve, a function of S

4 The Pricing Curve

All assets share one curve, fixed for a season. Units are divisible: the program accounts in base units, 10^6 per displayed unit. Since a single base unit is worth a fraction of a lamport, the curve is specified by its integral, the reserve required behind a given supply, rather than by a per-unit price.

4.1 Reserve function

$$R(S) = \left\lfloor \frac{S(a + bS)}{2^{96}} \right\rfloor \quad (3)$$

S is circulating supply in base units and $S_{\max}$ its cap, and $a \geq 1$, $b \geq 0$ are integer coefficients scaled by 2^{96} . For an asset X , $\text{minted}(X)$ is its S . R is non-decreasing with $R(0) = 0$. Its continuous derivative is affine, so marginal price ramps linearly in supply:

$$p(S) = \frac{a + 2bS}{2^{96}} \quad \text{lampports per base unit}, \quad \rho = \frac{p(S_{\max})}{p(0)} = \frac{a + 2bS_{\max}}{a} \quad (4)$$

The quoted price of the next whole unit is $R(S + 10^6) - R(S)$, within one lampport of the exact integral of the price over that unit, $10^6 \cdot p(S + 5 \cdot 10^5)$.

4.2 Settlement

Purchase and sale of n base units at supply S are the two halves of one integral:

$$\text{cost}(S, n) = R(S + n) - R(S), \quad \text{payout}(S, n) = R(S) - R(S - n) \quad (5)$$

Both are exact integers. The sum telescopes, so any decomposition of an order into smaller orders has identical total cost, and a round trip of n at the same S returns exactly the amount paid, before fees. A single participant therefore faces no execution cost beyond the curve itself, and order splitting confers no advantage.

4.3 Reserve invariant

$$\text{reserve}(X) = R(\text{minted}(X)) \quad \text{after every instruction, for every asset} \quad (6)$$

A purchase adds exactly cost ; a sale subtracts exactly payout . The program applies these as increments and decrements and never recomputes the reserve from R , so a violation shows up as a mismatch and is never silently corrected. Capture moves units between holders and never moves reserve. Units are created only by purchase and destroyed only by sale, the settlement auction included, or by the season-end liquidation, which pays every remaining unit the asset's average backing at a snapshot; from the start of that liquidation the equality above no longer applies to the asset.

4.4 Evaluating R in 128-bit arithmetic

The product $S \cdot (a + bS)$ can exceed 128 bits, so it is evaluated in two 64-bit limbs:

$$\begin{aligned} T &= a + bS, & lo &= S(T \bmod 2^{64}), & hi &= S \lfloor T/2^{64} \rfloor, & mid &= hi + \lfloor lo/2^{64} \rfloor, \\ R &= \lfloor mid/2^{32} \rfloor \end{aligned} \quad (7)$$

This equals $\lfloor S \cdot T / 2^{96} \rfloor$ exactly (a floor of a floor), with all intermediates below 2^{128} . Configuration bounds enforce $T < 2^{127}$, supply below 2^{54} base units and $R(S_{\max}) < 2^{60}$ lamports.

4.5 Inverting the curve for a lamport budget

Orders are usually denominated in lamports. The program finds `unitsFor( $S$ , budget)`, the largest n whose cost fits the budget, by bisection over the monotone R ; the solution is unique and requires no square root:

```

if budget = 0 or S ≥ Smax: return 0
hi = Smax - S; if R(S+hi) - R(S) ≤ budget: return hi
lo = 0                                     // cost(lo) ≤ budget < cost(hi)
while hi - lo > 1:
    m = lo + ⌊(hi - lo)/2⌋
    if R(S+m) - R(S) ≤ budget: lo = m else: hi = m
return lo

```

On the launch curve this takes at most 41 evaluations of R , and at most 53 at the largest cap that curve can be raised to; the general bound, supply below 2^{54} , allows 57. The TypeScript reference library and the Rust program run the same algorithm and must agree to the base unit; a closed-form reference checks both.

4.6 Rounding

Every division rounds against the trader: curve and pro-rata divisions floor, so the reserve keeps the remainder, while fees, the lock and the occupied-asset pool loss round up. A base unit on a flat segment of the integer curve may cost zero lamports; such a unit is backed by less than one lamport, the reserve retains the remainder, and an order whose total cost is zero is rejected.

4.7 Launch parameters

The launch curve is fitted exactly from three inputs: a cap of 150 000 units per asset (1.5×10^{11} base units), a fully issued reserve of 30 SOL, and a price ramp $\rho = 15$. The first whole unit costs 25 001 lamports and the last about 375 000. The cap can only be raised, at a round boundary, along the same line, so a raise never changes the value of existing units. At m times the launch cap the top price is $1 + 14m$ times the first and the full reserve is $30 \cdot m(2 + 14m)/16$ SOL.

5 Fees

The protocol's revenue is its trading fees; the one other inflow, once at a season's end, is the lamports no holder is entitled to. Fees are charged on top of the curve amount, never deducted from a reserve, and are paid into dedicated fee vault accounts.

5.1 Base rate and terminal ramp

Purchases and sales pay a base rate of 50 basis points. Over the final 9% of the trading window, its last 273 slots at launch (about 59 seconds at the current slot time; see Section 6), a surcharge ramps linearly from zero to 50 basis points on both sides. With $\tau_{\text{late}} = \lfloor \tau_{\text{trade}} \cdot 9/100 \rfloor$ slots in the ramp and d slots remaining to the halt:

$$s(d) = \begin{cases} 0 & d \geq \tau_{\text{late}} \\ \left\lfloor 50 \cdot \frac{\tau_{\text{late}} - d}{\tau_{\text{late}}} \right\rfloor & d < \tau_{\text{late}} \end{cases} \quad \text{bps}(d) = 50 + s(d) \quad (8)$$

The last tradable slot precedes the halt by one, so an order reaches at most $\lfloor 50(\tau_{\text{late}} - 1)/\tau_{\text{late}} \rfloor$ of the surcharge, 49 basis points with the 273-slot ramp. The ramp prices terminal positioning, when the allocation tallies carry the most information, at up to twice the base rate.

5.2 Purchase

For a purchase of x lamports at total rate bps , the program removes the fee from the budget, fits units to the remainder, then charges the fee on the realised cost, rounded up:

$$B = \left\lfloor \frac{x \cdot 10^4}{10^4 + \text{bps}} \right\rfloor, \quad n = \text{unitsFor}(S, B), \quad \text{cost} = R(S + n) - R(S), \quad (9)$$

$$\text{fee} = \left\lceil \frac{\text{cost} \cdot \text{bps}}{10^4} \right\rceil$$

Then $\text{cost} + \text{fee} \leq x$ holds unconditionally: $\text{cost} \leq B$ implies $\text{cost} \cdot (10^4 + \text{bps}) \leq x \cdot 10^4$, hence $\text{cost} + \lceil \text{cost} \cdot \text{bps}/10^4 \rceil \leq x$. No search loop is needed, and n is the maximal affordable quantity. The amount entered is always an upper bound on what is debited.

5.3 Sale

A sale credits payout $-\lceil \text{payout} \cdot \text{bps}/10^4 \rceil$ to the seller's balance. In the settlement auction the base rate applies without surcharge.

5.4 Fee-free operations

The transfer of supply on capture carries no fee, and the validated configuration rejects any non-zero capture fee. Allocation votes, claims and syncs carry no fee. A voided round moves no fees.

5.5 Execution guards

Every trading instruction carries the expected round, the maximum accepted fee rate, and a maximum cost or minimum payout. If the round has advanced, the ramp has passed the accepted rate, or the price has moved beyond the bound, the instruction fails rather than executing at a worse price. A compromised read API or a sandwich attempt cannot alter the holder's execution.

5.6 Fee vaults

Fees accrue to one of up to 64 fee vault accounts. Each trade names its vault, so concurrent trades on different assets do not contend for a single writable account; protocol revenue is the sum across vaults. No instruction moves lamports from a fee vault to a reserve. The only movement the other way happens once, at a season’s end: retiring an asset sweeps its remaining reserve, orphaned captured supply and flooring remainders that no holder is entitled to, into the first fee vault.

6 The Round Clock

Time is measured in Solana slots, not wall-clock time, and this paper gives every duration in slots. The launch values were chosen in seconds and converted once, at initialisation, at the mainnet slot time measured then, 217 ms. Two things can change them after this paper was written. Solana’s slot time can change, which changes the wall-clock length of every duration while the slot counts stay fixed. The protocol’s settings can change the slot counts themselves, at a round boundary and after a public notice of six rounds (26 538 slots at launch). Every wall-clock figure in this paper is therefore an estimate at 217 ms per slot, and every slot count is a launch value; both may differ at the time of reading. Rounds are numbered from 1 and follow an absolute schedule, fixed at genesis and re-anchored only when a change to the slot counts applies:

$$t_i = t_a + (i - a) \cdot \tau_{round}, \quad \tau_{round} = \tau_{trade} + \tau_{sealed} + \tau_{res} + \tau_{auc} \quad (10)$$

The anchor (t_a, a) is $(t_{genesis}, 1)$ at launch; a change to the slot counts moves it to the boundary where the change applies, so no existing boundary moves. Every boundary of round i derives from t_i . No caller, the operator included, can move a boundary; a late call to open a round shortens that round and never shifts the randomness slot L . Launch parameters:

Table 5: Windows of the round clock at the launch settings. The About column is an estimate at 217 ms per slot (see Section 6).

Window	Slots	About	Permitted instructions
Trading, t_i to h	3 041	11 min	buy, buy_shares, sell, set_vote, sync
Sealed buffer, h to L	138	30 s	none that modify positions
Resolution, L to W	691	2 min 30 s	capture_hash, reveal_seed, resolve_battle
Settlement auction, W to E	553	2 min	queue_sell on captured assets, claim, sync

A round is 4 423 slots at launch: about 16 minutes, or 90 rounds a day, at 217 ms per slot.

Trading is the only window in which continuous trading is open, uniformly across assets. Allocation votes freeze and trading halts at the same slot h .

Sealed buffer. A fixed-length interval in which nothing trades, votes or syncs, so the slot whose hash seeds the round is separated from the last state change.

Resolution opens at L . The slot hash is captured, the seed is revealed and the contests resolve on chain early in the window; the remaining slots exist for presentation. No asset can be traded. Since every outcome is computable by anyone once the seed is revealed, an open market here would reward the fastest chain reader; closing every market until W makes that information worthless.

Settlement auction opens at W . Only assets captured in the round accept sell orders, and those settle jointly at E (see Section 11). Winning holders claim captured supply here. All other assets remain closed until the next trading window.

A round still unresolved at E cannot be closed. It can be voided once the stuck limit has passed (33 180 slots after L at launch, about 2 hours at the current slot time), and the following trading window is shortened by the delay rather than displaced.

7 The Commitment Lock

A fixed fraction of every purchase is locked until the round of purchase has ended. Without the lock, the dominant strategy would be to buy early, ride inflows while the allocation tallies are public, and exit before the halt if the asset is targeted: curve profit with zero exposure to resolution. Under the lock, every purchase carries collateral into at least one resolution.

The lock is two counters on the position, not a purchase list, so every check is constant cost. With $\lambda = 5\,000$ basis points and current round e :

$$\text{buy}(n) : \quad \ell = \left\lceil \frac{n \lambda}{10^4} \right\rceil, \quad \text{locked} += \ell, \quad \text{free} += n - \ell, \quad \text{lockedEpoch} = e \quad (11)$$

$$\text{release} : \quad \text{locked} > 0 \wedge e > \text{lockedEpoch} \Rightarrow \text{free} += \text{locked}, \quad \text{locked} = 0 \quad (12)$$

Release runs before every purchase and sale. The ceiling locks a single-base-unit purchase in full. Further purchases in the same round join the locked balance; a purchase in a later round first releases the earlier balance, so a new purchase never extends an older lock.

`sell` and `queue_sell` require $n \leq \text{free}$. No holder instruction sells, transfers or burns locked units. Two protocol paths ignore the lock: a capture takes every unit held in positions on the captured asset, locked units included, and the season-end liquidation pays them out with the rest. The invariant $\text{shares} = \text{free} + \text{locked}$ holds after every instruction.

The lock releases when the round advances, regardless of whether the round's resolution produced contests or was voided. A voided round had no resolution to protect, and tying release to resolution would freeze the most recent purchase indefinitely if resolutions ever stopped. Supply received by capture or claim is credited to `free`: it has already been through a resolution.

As a consequence, the committed collateral of a targeted asset includes its most recent buyers.

8 Contest Weight and Target Selection

8.1 Collateral as contest weight

An asset’s weight in a contest is denominated in lamports, not units. The collateral behind n units of asset X is their pro-rata share of its reserve, 0 when $\text{minted}(X) = 0$:

$$\text{backing}(X, n) = \left\lfloor \frac{n \cdot \text{reserve}(X)}{\text{minted}(X)} \right\rfloor \quad (13)$$

An asset’s contest weight is exactly the collateral that changes hands if it loses: its reserve, less the collateral behind its units held in other assets’ capture pools, plus the collateral behind every capture pool it holds. With $c_Y[X]$ the units of X in the pool held by Y , and $F(X)$ the units of X held in positions:

$$F(X) = \text{minted}(X) - \sum_{Y \neq X} c_Y[X] \quad (14)$$

$$w(X) = \text{backing}(X, F(X)) + \sum_{Z \neq X} \text{backing}(Z, c_X[Z]) \quad (15)$$

This is the weight at the launch pool strength of 100%; that setting, if lowered, scales each pool’s term down. For an asset with no pools on either side, $w = \text{reserve}$. Under this rule every lamport of weight is attributed to exactly one asset and each side’s weight is exactly what it would lose, which makes each contest an actuarially fair lottery (see Section 10). The occupied-asset setting described next changes the rule for partly occupied assets.

8.2 Occupied assets

The weight rule prices every unit of an asset at the same average backing, whether the unit sits in a position or in another asset’s capture pool. On a shared curve this matters for an asset that is partly occupied. A purchase of X raises $\text{reserve}(X)$, and with it the average backing of every unit of X , including the units held in its occupiers’ pools. Part of the buyer’s lamports therefore adds to the occupiers’ weight rather than to the weight of X : in one recorded case, with the setting below at 0, a purchase of 0.988 SOL after fees into a partly occupied asset added 0.716 SOL to its own weight and 0.272 SOL to its occupiers’. Nothing is lost, since the lamports are in the reserve of X and the occupiers’ captured supply is worth more, but an asset’s reserve, the lamports it holds, and its contest weight, the lamports it fights with, can differ. The client shows the difference: an occupied asset’s card gives, under its weight, what its own holders have in its reserve, and a purchase quote gives how much of the purchase counts in war for the asset.

How much of this effect a contest counts is a setting, `occupied_backup_bps`, written κ , from 0 to 10 000 basis points. It changes the weights of partly occupied assets and of the pools that hold their units, and through them the odds of each contest and the split of captured supply among the winners. It does not change which units a capture transfers, the price of any purchase or sale, or the season-end payout, and target selection’s exclusion test and the vote floor do not depend on it. At 0 the weights are w

above; at 10 000 a purchase of X counts its whole cost for X and nothing for its occupiers; a value between moves each weight $\kappa/10\,000$ of the way. Let F be short for $F(X)$, $P = \text{minted}(X) - F$ the units in capture pools, and $u = \min(R(P), \text{reserve}(X))$ the bottom of the curve, under the captured supply, with $u = 0$ when $P = 0$. With $(\cdot)^+$ truncating at zero, the asset's own weight is

$$\text{own}(X) = \text{backing}(X, F) + \left\lfloor \frac{\kappa \cdot (\text{reserve}(X) - u - \text{backing}(X, F))^+}{10^4} \right\rfloor \quad (16)$$

with $\text{own}(X) = 0$ when $F = 0$. A pool of c units of Z held by X contributes its average backing moved κ of the way down to its share of u_Z , its loss rounded up:

$$\text{pool}(X, Z) = \text{backing}(Z, c) - \left\lfloor \frac{\kappa \cdot (\text{backing}(Z, c) - \min(\lfloor c \cdot u_Z / P_Z \rfloor, u_Z))^+}{10^4} \right\rfloor \quad (17)$$

Here u_Z and P_Z are u and P computed for Z . Both formulas are at the launch pool strength of 100%; the weight of X is $\text{own}(X)$ plus the sum of its pools. Because the gain floors and the loss ceils, the parts of one reserve never sum past it. At $\kappa = 0$ both formulas reduce to $w(X)$ above. For the asset's own units the difference has a closed form on a curve of this shape, since the linear term of R cancels:

$$\text{reserve}(X) - u - \text{backing}(X, F) = \left\lfloor \frac{b F P}{2^{96}} \right\rfloor + \epsilon, \quad \epsilon \in \{0, 1, 2\} \quad (18)$$

The gap grows with the product $F \cdot P$: an asset with little supply barely moves, and one near its cap with its supply split between positions and pools moves most. The amount $\text{reserve}(X) - u$ is the top of the curve, above the captured supply. It is what the holders of X would receive, fees apart, by selling all F units before any captured unit of X is sold, and exactly what they paid when every unit in positions was bought after the asset's last capture and no captured unit has been sold since; it is the figure an occupied asset's card shows under its weight. The setting therefore chooses where a contest counts an occupied asset's own units between two valuations: at 0 their average backing, at 10 000 the top of the curve.

Neither end of the range is fair under both valuations. Measured at the average backing, which is what the season-end payout pays for every unit and how this paper counts the collateral a capture moves, a contest at 0 is an actuarially fair bet for every side, and above 0 it is not: with challenger weight C , defender weight D and stakes stake_C , stake_D , the average backing of what each side loses, the challengers' expected gain is

$$\mathbb{E}[\Delta_C] = \frac{\delta_C \text{stake}_D - \delta_D \text{stake}_C}{C + D}, \quad \delta = \text{weight} - \text{stake} \quad (19)$$

Above 0 the own weight of an occupied asset exceeds the backing of its units, and each pool holding its units counts for less than its backing. On this measure a third asset that attacks a partly occupied asset therefore loses in expectation when neither holds other pools, and one contest can move the weight

of an asset that takes no part in it. Measured instead at what the holders of X would receive by selling first, the stake of X is the top of the curve, its own weight at 10 000: on that measure the setting at 0 undercounts X in a contest with a third asset, and 10 000 counts it at its stake. Captured units are later sold in an order that no rule fixes, so which measure describes what a contest moves depends on who sells first. On the launch curve, take an asset X with a reserve of 9 SOL, of which its units in positions are backed by 5 SOL at the average and 6.74 SOL at the top of the curve and its occupier’s pool by the other 4 SOL at the average, and a third asset with a reserve of 5 SOL that attacks it. The weight of X is 5, 5.87 and 6.74 SOL at 0, 5 000 and 10 000, and the attacker’s chance is 50%, 46.0% and 42.6%.

The setting also changes what holding captured supply pays. Over 300 simulated rounds, a player who held captured supply finished ahead of an otherwise identical player who sold it by 2.26 ± 0.81 SOL with the setting at 0, 0.60 ± 0.59 SOL at 5 000 and 0.25 ± 0.60 SOL at 10 000. In the simulator’s baseline population on the launch curve, with every asset at its cap by round 100 and the setting applied from round 1, the expected value the setting shifts between sides, measured at the average backing, is about 29 SOL per 300-round season at 2 500, 85 SOL at 5 000 and 205 SOL at 10 000, most of it in contests between an asset and its own occupier.

The setting is game state, so changing it needs no program upgrade. `set_occupied_backup` stages a value, which replaces any value already pending and applies at the first round boundary after the settings notice has elapsed (six rounds, 26 538 slots at launch; see Section 6), but never at a boundary reached at or after the randomness slot L of the round it would open, so the value a round is played under can never be chosen after that round’s slot hash is public. The instruction is refused once a season’s end has been announced, and the value carries into the next season. The client shows the current value and marks any pending one before it applies.

8.3 Why linear weight

Each non-linear weight function admits a cheap attack. Per-account concavity, such as $(\sum \sqrt{\text{stake}})^2$, is defeated by Sybil splitting, since there is no identity layer. Per-asset concavity, such as $\sqrt{\text{reserve}}$, is defeated by coalitions, because capital split across allied assets gains combined weight. Convexity rewards concentration directly. Linear weight is the only choice that is neutral to splitting.

The early-entry advantage comes from the curve instead. A late buyer pays the marginal price but adds the average backing, so weight per lamport paid is

$$\frac{\text{average backing}}{\text{marginal price}} = \frac{\rho + 1}{2\rho} \quad (20)$$

for a linear ramp of ratio ρ from the bottom of the curve. At the top of a $15\times$ ramp this is 0.53: late capital converts to weight at about half its face value, and the remainder accrues as backing to earlier holders.

8.4 Allocation votes

Each position carries one target and a weight equal to its units. An asset’s tally is maintained incrementally by every purchase, sale, claim and vote change, so freezing the vote at the halt never iterates

over holders. Units in a capture pool do not vote; claiming them is how their holder acquires a vote in that asset. Votes persist across rounds until changed. An asset cannot target itself.

Two floors prevent dust from steering outcomes. A vote counts only if its position is backed by at least a minimum number of lamports, and selling below that floor removes the weight from the tally rather than leaving it standing. A purchase must also be at least a minimum amount, fee included. Both default to the price of 40 initial units, about 0.001 SOL at launch.

8.5 Target selection

At the halt, each asset with a non-empty tally selects one target by weighted lottery over its tally, using one draw from the round’s random stream (see Section 9). A single voting unit suffices to commit an asset; there is no abstention outcome. Targets with nothing to transfer are excluded and the tally re-weighted over the remainder. The transferable value is measured in lamports, against the minimum prize $P_{\min}$, the price of 40 initial units (1 000 040 lamports at launch):

$$\text{barren}(t) \iff \text{backing}(t, F(t)) + \sum_{Z \neq t} \text{backing}(Z, c_t[Z]) \leq P_{\min} \quad (21)$$

With admissible tally weights $v_1 \dots v_k$ and $V = \sum v_j$, target j is selected when

$$\sum_{m < j} v_m \leq (r \bmod V) < \sum_{m \leq j} v_m \quad (22)$$

so each target’s selection probability equals its share of vote weight. Assets that select the same target contest it jointly as one side. There is no adjacency constraint: any asset may target any other.

9 Randomness

All outcomes in a round derive from one 256-bit value Θ , built from two inputs that no single party controls at the moment it matters: a seed committed long before the round, and the hash Solana records for a slot after trading has stopped.

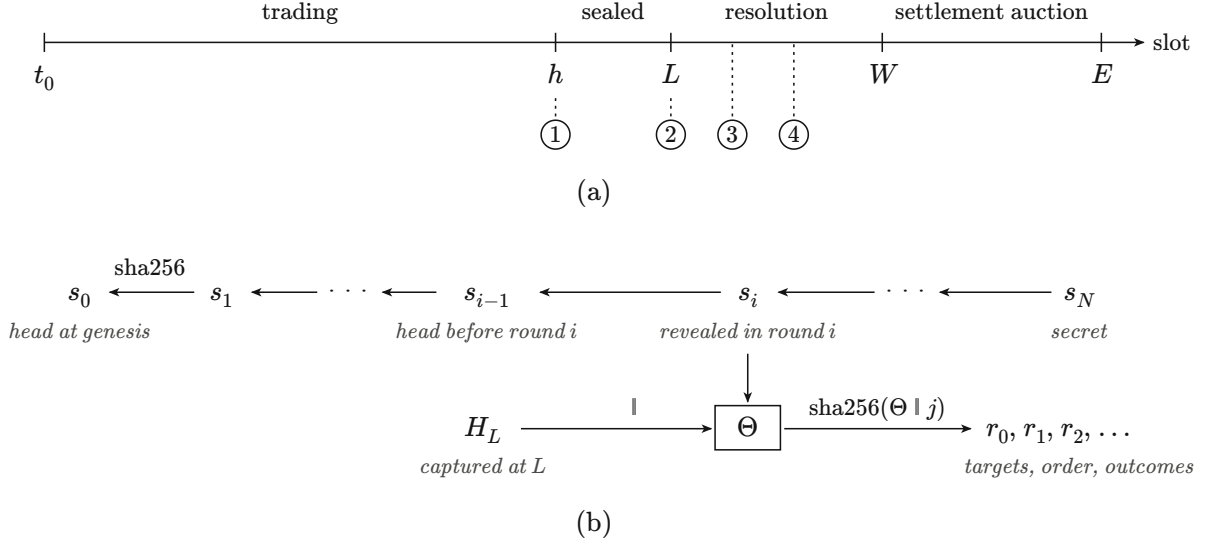


Figure 1: Randomness for round i . (a) Order of events on the slot axis, not to scale: (1) trading and allocation votes halt at h , so every tally is final; (2) from L , `capture_hash` stores H_L , the hash of the first block at or after L ; (3) the operator reveals s_i , accepted only if it hashes to the stored head; (4) the program computes Θ and resolves every contest from the draw stream. (b) The seed chain and the derivation of Θ and the draws. Each arrow in the chain is one application of sha256.

9.1 Seed chain

The operator generates a secret terminal seed s_N and hashes it backwards N times. The chain head is published on chain before the first round, committing every future seed at once:

$$s_N \text{ secret}, \quad s_{i-1} = \text{sha256}(s_i), \quad \text{head} = s_0 \text{ published at initialisation} \quad (23)$$

Each reveal discloses the next link; the program accepts it only if it hashes to the stored head, which it then replaces. The launch chain holds 1 000 000 links, one per round: about 30 years at the launch round length and the current slot time.

If a round voids after its slot hash was captured but before a seed was accepted, the required seed may already be public in a failed transaction. The program counts such skips and requires the next reveal to hash $k + 1$ times to the head, so a possibly exposed seed is never consumed:

$$\text{sha256}^{(k+1)}(s) = \text{head}, \quad 0 \leq k \leq 64 \quad (24)$$

A chain can be replaced only under both the admin and the operator signature, never in the round in which the replacement was staged, and never at a boundary late enough to observe the next round's slot hash. Every replacement is counted publicly.

9.2 Capture and reveal

1. Trading and voting halt at h , followed by the 138-slot sealed buffer (about 30 seconds at the current slot time).
2. From slot L , any account may call `capture_hash`. It reads from Solana’s SlotHashes sysvar the hash of the first slot at or after L that produced a block, within a 512-slot window (about 111 seconds at the current slot time). The 512 slots are the depth of the SlotHashes sysvar, so this bound is Solana’s, not a setting. Falling forward past a skipped slot gives a block producer nothing, because the seed is still secret.
3. The operator reveals s_i within a deadline of 512 slots after L at launch. The program refuses a reveal before capture.
4. The round value is

$$\Theta = \text{sha256}(s_i \parallel H_L) \quad (25)$$

The producer of slot L cannot know s_i , so it cannot steer Θ through block production. The operator knows s_i but cannot influence H_L . The remaining assumption is an operating rule: the seed holder never runs, controls or pays a Solana block producer. The captured slot is recorded, so an anomalous rate of skipped L slots would be publicly detectable.

9.3 Draw stream

Every random choice in the round is a draw from one stream, each interpreted as a big-endian 256-bit integer:

$$r_j = \text{sha256}(\Theta \parallel \text{u64le}(j)), \quad j = 0, 1, 2, \dots \quad (26)$$

Consumption order is fixed, so the TypeScript reference library and the Rust program reproduce it exactly:

1. **Targets.** One draw per active asset in index order, consumed even when the asset has an empty tally.
2. **Contest order.** Contests are grouped by defender and listed by defender index, challengers ascending. A Fisher-Yates shuffle runs over the 0-indexed list for $i = n - 1$ down to 1, swapping position i with position $r \bmod (i + 1)$, one draw per swap, $n - 1$ draws in all.
3. **Outcomes.** One draw per contest in shuffled order, consumed even when the contest is void: its defender, or every one of its challengers, was captured earlier in the round, or $D + C = 0$.

Every reduction takes the full 256-bit draw modulo a bound below 2^{66} , so every outcome’s probability is exact to a relative error below 2^{-190} , an absolute error below 2^{-192} .

9.4 Withheld reveals

The operator learns Θ before revealing and could withhold the reveal. If no reveal lands by the deadline, the round voids with zero transfers: no units move, fees stay where they are, and the next round proceeds. The void counter is on chain and shown on the verification page. Withholding gains the operator nothing, because neither the operator nor the provider holds positions, and it cannot select an outcome: at most it produces a void round, which is public and counted.

10 Contest Resolution and Capture

10.1 Resolving a contest

All challengers that selected the same defender contest it as one side. With C the sum of the challengers' weights, D the defender's weight, and r the contest's draw:

$$x = r \bmod (D + C), \quad \text{defender prevails} \iff x < D, \quad (27)$$

$$\Pr[\text{defender prevails}] = \frac{D}{D + C}$$

The reduction is taken over the full 256-bit draw, limb-wise on chain, with $D + C < 2^{66}$. If the defender prevails it captures every remaining challenger in index order; a failed challenge costs each challenger all of its transferable collateral, just as a successful one costs the defender all of its own.

With the occupied-asset setting at 0, each side's weight equals exactly the collateral it would lose, so every contest is actuarially fair before fees (for other values, see Section 8.2). For the defender:

$$\mathbb{E}[\Delta] = \frac{D}{D + C} \cdot C - \frac{C}{D + C} \cdot D = 0 \quad (28)$$

and symmetrically for the challengers. Expected value is therefore not created by contest selection; it arises from curve position, from anticipating which assets will be targeted, and from how holders manage captured supply. A contest is void, and still consumes its draw, if its defender or all of its challengers were captured earlier in the round; challengers captured earlier simply drop out of a contest that still has others.

10.2 Sequencing within a round

Contests resolve one at a time in shuffled order. An asset captured earlier in the round forfeits its own scheduled challenge, and an asset that prevails in one contest and loses a later one forfeits what it gained. Reciprocal targeting needs no special case: whichever contest is drawn first decides, and the second is void.

10.3 Capture

When the winning side captures asset X :

1. Each winner's weight is snapshotted before any transfer and reused for every loser in the same contest.
2. Every unit of X held in positions, locked units included, becomes captured supply. Units of X already held in some asset's capture pool remain there; they transfer only when their holding asset is captured.
3. X 's generation counter increments, which zeroes every position on X without visiting it (see Section 3).
4. The captured supply, M units, is apportioned among winners by weight, with the flooring remainder to the largest contributor, ties to the lowest index:

$$y_i = \left\lfloor \frac{M \cdot w_i}{\sum_k w_k} \right\rfloor, \quad \text{remainder} = M - \sum_i y_i \rightarrow \arg \max_i w_i \quad (29)$$

5. Every capture pool held by X transfers to the winners, apportioned the same way.
6. $\text{minted}(X)$ and $\text{reserve}(X)$ are unchanged. Units moved; nothing was burned and no lamport left a reserve.

10.4 Capture pools and constant-time accounting

A capture cannot write to every holder of the winning asset, so the program writes one record per source asset on the winner, $c_A[X]$, with an accumulator from which each entitled holder later derives their share. Each asset holds at most 16 pool slots, one per source.

Each slot keeps a monotone accumulator scaled by 2^{64} . When captured supply M of source X is credited to asset A , whose holders' total recorded stake is Σ_A :

$$\text{acc}_A[X] += \left\lfloor \frac{M \cdot 2^{64}}{\Sigma_A} \right\rfloor \quad (30)$$

If $\Sigma_A = 0$ the supply joins the pool with no entitlement and the accumulator does not move. A holder o with recorded stake σ_o and bookmark $\text{debt}_o[X]$ from their last interaction is entitled to

$$\Delta z_o[X] = \sigma_o \cdot (\text{acc}_A[X] - \text{debt}_o[X]) \quad (\times 2^{-64} \text{ units of } X) \quad (31)$$

Entitlements are computed in `settle`, which runs first in every instruction that touches a position and standalone as the permissionless `sync`. It iterates over at most 16 slots and their components, so its cost is bounded independently of the holder count; the bookmark then advances to the current accumulator.

Recorded captured supply also earns. A holder's stake is their own units plus stake points for recorded pool entitlements, each pooled unit converted into an equivalent number of the holding asset's units at prices snapshotted when the round's seed was revealed. The points for n units of source Y pooled on A are 0 when $\text{reserve}(A) = 0$, and otherwise

$$\text{pts}(Y \rightarrow A, n) = \left\lfloor \frac{\text{backing}(Y, n) \cdot \text{minted}(A)}{\text{reserve}(A)} \right\rfloor \quad (32)$$

A holder who leaves captured supply pooled therefore earns a larger share of the holding asset's next capture than one who claimed it, and loses it if the holding asset is captured. A slot that is retired and re-established clamps stale bookmarks out of the dead interval, so a holder can forfeit one component's accrual over one interval and nothing else.

10.5 Claims

`claim` moves whole units from a holder's recorded entitlement in $c_A[X]$ into their own position on X , unlocked. Claimed units are no longer exposed to A 's contests, vote in X , and can be sold. Claims are permitted only in the settlement auction of a resolved round, and they are irreversible: no instruction moves units into a capture pool. Outside contest resolution, no pool's unit count can increase and no slot can be created.

11 The Settlement Auction

After a capture, the receiving holders typically want to exit simultaneously. On a continuous curve the first seller after resolution would extract several times what the last does, converting the opening of trading into a priority-fee race that automated agents win. The settlement auction removes the race: every seller of a captured asset receives the same price.

From W to E , each asset captured in the round accepts `queue_sell(n)`. The units move from the seller's free balance into escrow immediately and cannot be withdrawn. The auction accepts no buys; purchasing resumes in the next trading window. At E , each captured asset's queue settles as a single block down its own curve:

$$\Delta S = \sum_i q_i, \quad G = R(S) - R(S - \Delta S) \quad (33)$$

$$g_i = \left\lfloor \frac{q_i \cdot G}{\Delta S} \right\rfloor, \quad f_i = \left\lfloor \frac{g_i \cdot b_s}{10^4} \right\rfloor, \quad \text{payout}_i = g_i - f_i \quad (34)$$

Here q_i is seller i 's queued units and b_s the sale rate, 50 basis points. The flooring remainder $G - \sum g_i$ goes to the last seller to queue, less its own fee $\lceil (G - \sum g_i) \cdot b_s / 10^4 \rceil$. Because the g_i and that remainder sum to G exactly and the reserve decreases by G , the invariant reserve $= R(S - \Delta S)$ holds after settlement and the protocol never pays from its own funds. Each captured asset clears at its own price; the base fee applies without surcharge.

Settlement is resumable. The clearing total, the new supply and the reserve change on the first call, so no later call can price against a different curve; each call, the first included, pays at most 20 sellers and advances a cursor. Any account may continue it. The next round cannot open until every captured asset with a queue has settled, so a large batch can delay the protocol but cannot halt it.

Proceeds are recorded on each seller’s position as `payout_owed` and collected by `withdraw_batch`, a permissionless instruction whose destination derives from the stored owner. The crank collects on every holder’s behalf; a third party can trigger a payout but cannot redirect it.

Participation is optional. Units not sold remain held into the next round, where continuous trading resumes.

12 Invariants

The program maintains the following equalities and bounds after every instruction. Each has an executable checker in the reference library and a property test over randomly generated states.

Table 6: Invariants maintained after every instruction.

Invariant	Statement
Reserve equality	$\text{reserve}(X) = R(\text{minted}(X))$ for every asset, until a season-end liquidation starts on it
Unit conservation	$\text{minted}(X)$ equals the units of X in live positions plus the units of X in every live capture pool plus the units escrowed in its unsettled settlement auction
Position split	$\text{shares} = \text{free} + \text{locked}$ for every position
Stake totals	each asset’s Σ_A equals its live holders’ units plus their recorded stake points
No over-distribution	for every live pool, claimed plus pending entitlements never exceed the supply credited
Retired slots carry no stake	a slot that is not live holds zero stake
Pools fill only by capture	outside contest resolution, no pool’s units increase and no slot is created
Fair sides	for each side of a contest, the sum of its weights equals the backing of exactly the units that side would lose, each counted once, when the occupied-asset setting is 0 (see Section 8.2)
Fee separation	no lamport moves from a fee vault to a reserve; a reserve pays a fee vault only when a retired asset’s residue is swept at a season’s end
Arithmetic bounds	every stored accumulator, entitlement and intermediate product is below 2^{128}

No floating point appears wherever a lamport, unit, vote weight or multiplier is computed. Multipliers are basis points over 10^4 . Every division rounds against the trader (see Section 4), and each formula is ordered so its intermediates fit in 128 bits; the two exceptions, the product inside R and the reduction of a 256-bit draw, are evaluated limb-wise.

The reference library is pure: identical inputs produce byte-identical state, with no clock and no randomness beyond the round’s committed inputs. That purity is what qualifies it as the oracle against which the on-chain program is tested (see Section 15).

13 Liveness and Failure Modes

The protocol is designed so that no single party going offline can strand holder funds.

13.1 Permissionless crank

Most of the round loop can be advanced by any account: `capture_hash`, `resolve_battle`, `void_round`, `sync` and `withdraw_batch` require no privileged signer. `resolve_battle` takes no arguments; it resolves the next contest in the drawn order, so a caller can choose neither which contest runs nor how. Independent keeper instances race to submit each step, and duplicates are rejected. Every crank instruction requires the complete set of asset accounts, each verified as the PDA for its index and in index order, so omitting an account to influence an outcome is impossible.

Three instructions are reserved to the operator: `reveal_seed`, which only the seed holder can produce, and `close_window` and `advance_round`, which close the settlement auction and open the next round.

13.2 Voided rounds

If the slot hash is not captured within its 512-slot window, or no seed is revealed by the deadline, any account may void the round. A void moves no units and no fees, increments a public counter, and lets the next round proceed. A round whose resolution never completes can always be voided once the stuck limit has passed: 33 180 slots after L at launch (about 2 hours at the current slot time), and never more than 216 000 slots (about 13 hours at the current slot time).

13.3 Dead-man fallback

If the operator stops opening rounds, the protocol continues without it. Once the operator has exceeded the fallback interval since its last on-time opening, any account may close and advance rounds:

$$\text{fallback} = \min(4 \cdot \tau_{\text{deadman}}, 2 \cdot \tau_{\text{round,max}}) \quad (35)$$

At launch the dead-man interval is 99 539 slots and the longest permitted round is 216 000 slots, so the fallback opens after the smaller of $4 \times 99\,539$ and $2 \times 216\,000$, that is 398 156 slots: about 24 hours at the current slot time (see Section 6). Until then the round remains where the operator left it; once the fallback opens, every round can be advanced by anyone. The pause instruction itself becomes permissionless after the dead-man interval (99 539 slots at launch, about 6 hours at the current slot time), and a paused protocol still accepts sales: purchases stop at once and contests from the round after next, while exits remain open.

13.4 Bounded compute

No instruction iterates over holders. Capture is a counter increment plus a bounded number of pool writes; claims and syncs are bounded by 16 slots per asset; weight computation reads at most 16 asset accounts. The worst single contest measured, pool-carrying challengers defeating a pool-carrying de-

fender, used 447 698 compute units with the occupied-asset setting at 0 and 625 011 above it, within a transaction with a raised compute budget. Holder-signed instructions request compute in proportion to the records they settle.

14 Delegated Signing and Custody

14.1 Program-controlled balance

A delegated key cannot spend from a wallet; Solana requires the wallet's own signature to move its lamports. A purchase without a wallet prompt therefore requires funds under program control. Each holder has one account at ["player", owner] holding a lamport balance. No keypair controls it, and a holder whose client state is lost reconnects the wallet and finds the balance intact. Sale proceeds, auction proceeds and returned account deposits are credited to this balance, so funds remain inside the protocol until the owner withdraws them.

14.2 Session keys

The holder account stores one delegated key and its limits:

Table 7: Fields of the delegated signing key on the holder account.

Field	Purpose
session_key	the delegated public key
session_epoch	incremented to revoke; a key is valid only at the current epoch
session_expires	hard expiry enforced by the program
session_scope	capability bits: buy, sell, vote, claim
session_spent, session_cap	cumulative cost plus fee under the session, and its ceiling
session_round_ops, session_max_ops	operations this round, and the per-round ceiling
session_positions_opened, session_max_positions	positions created under the session, and the ceiling

Revocation is a single counter increment, independent of how many keys were ever issued; the program never stores a key list, because revoking a list would be an unbounded loop. Every delegated instruction is validated against live account state at execution time, never against data carried in the instruction. The vote and claim capabilities are off by default: claims are irreversible, and a sell-only key could otherwise zero a vote at the last tradable slot.

14.3 Derived destinations

Every instruction that moves lamports derives its destination from stored state; none accepts a destination as an argument. Under this rule the caller's identity does not matter for an instruction that

only pays its rightful owner, which is why collection and sync can be permissionless.

Table 8: Callers permitted for each lamport-moving instruction.

Instruction	Owner	Delegated (capability)	Any account
withdraw_war_balance, register_session, revoke_all, revoke_all_and_withdraw	yes	no	no
close_position, close_player_account	yes	no	no
buy, buy_shares	yes	buy	no
sell, queue_sell	yes	sell	no
set_vote	yes	vote, off by default	no
claim	yes	claim, off by default	no
withdraw_batch, sync	yes	n/a	yes

14.4 Emergency exit

`revoke_all_and_withdraw` is one owner-signed instruction that increments the session epoch and transfers the entire balance to the wallet in the same transaction, leaving a compromised key no window in which to act. It empties the balance; it does not liquidate positions, and the locked portion of the latest purchase remains locked until the round ends.

14.5 Custody

No instruction moves lamports out of a holder balance or a position to anyone but its owner, and spending them requires the owner’s signature or the owner’s scoped delegate. Fee vaults are separate accounts, and protocol-withdrawable revenue is each vault’s balance above its rent floor. The operator holds no key that can move a reserve, a holder balance or a position.

15 Verification

15.1 Outcome proofs

Any participant can open any round’s proof and verify its randomness locally, from values already public on chain. The client and the backend run identical checks using the reference library’s hashing and draw code:

1. s_i , hashed $k + 1$ times, equals the chain head before the reveal, with k the recorded skip count (0 unless a round was voided after its hash was captured), so the seed was committed in advance.
2. The slot hash originates from slot L or later and was captured in a transaction that landed no later than the reveal.
3. $\text{sha256}(s_i \| H_L)$ equals the recorded Θ .

4. Each asset’s target is the weighted selection over its admissible tally at $r_j \bmod V$, one draw per active asset in index order.
5. The contest order is the Fisher-Yates permutation of the defender-ordered contest list, one draw per swap.
6. Contest k in that order consumes the next draw: $x = r \bmod (D + C)$, and the defender prevails if and only if $x < D$, including draws consumed by voided contests.

The round account holds the live round’s slot hash, seed and Θ , and a history account retains the last 64 rounds’ seeds and hashes, so verification requires no trusted server. When the backend records a round, it runs the same verification and raises an alert on any mismatch, so a defect reaches an operator before a participant observes it.

15.2 Differential testing

The TypeScript reference library implements every rule in this paper. It is pure and deterministic, and each invariant above is a property test over randomly generated states. The Rust program is tested against it by replaying 1 905 recorded cases through both implementations: an accepted operation must leave byte-identical state, and a refused one must be refused with the same error. Scenario suites then run the program on a local validator with real slots, slot hashes and lamports, from a two-round, six-asset configuration to a sixteen-asset joint contest over four rounds, together with delegated signing, the keepers, season termination and the operator fallback.

Adversarial cases are included: a crank instruction given the asset set minus one account, with a duplicate account, or out of order is rejected before any state changes.

15.3 Verifiable build

The program interface definition is published, the build is reproducible, and the client is open source, so anyone can confirm that the deployed program is the one specified here.

16 Scope of This Version

16.1 A deliberately constrained first deployment

The protocol described in this paper is the beta of Despot. Its purpose is empirical: to measure whether demand exists for parimutuel trading as a product category before the design is generalised. The constraints of this version follow from that purpose. Each was chosen because it is simple to specify, to test differentially against the reference library and to audit, not because it is the best mechanism available. The deployment should be read as a prototype of the models that will follow it, not as their final form.

16.2 Known limitations and directions

1. **A fixed clock.** Every round has four stages of fixed length, so a round with few contests waits through a resolution window sized for many, and a round with many contests compresses their presentation. An elastic clock, in which the resolution window lasts as long as its contests require, a round without contests skips it, and the settlement auction is folded into the next round's opening, preserves every fairness property in this paper: a randomness slot no one can move, one price per captured asset, and no open market while results are public. It requires a program upgrade, not new infrastructure.
2. **A bounded asset set.** $N_{\text{MAX}} = 16$ is compiled into the account layouts as one slot per asset. Keying those slots by asset identifier, bounding them by what an asset actually holds and making per-round state lazy removes the bound, so the asset set and the coalitions within it can grow on demand, with costs proportional to the contests actually fought rather than to the size of the map. The new layout requires a new season.
3. **A synchronised opening.** Every curve reopens and every lock releases at the same slot, which rewards the fastest transaction at the boundary. A uniform-price batch at the round's opening, analogous to the settlement auction, removes that race.
4. **An operator-held seed chain.** The randomness relies on the seed holder neither steering nor withholding outcomes. Withholding is public and costs holders nothing (see Section 9), but it is still possible. A randomness source with no single seed holder, such as a threshold signature scheme or a verifiable random function, would remove that residual trust.
5. **A single global calendar.** Every asset trades and resolves on one schedule. Asynchronous fronts, compressed state for idle assets, contests resolved in an ephemeral rollup or proven off chain, and an application-specific chain were each evaluated. At the scale of this beta each is worse on at least one of cost, trust or latency, and they are kept as options for scales far beyond it.

The invariants, the integer arithmetic and the verification procedure in this paper are intended to carry over unchanged to those later models, which change the clock, the size of the asset set and the source of randomness.