Solana Development Patterns

Mark Virchenko
RedDuck

Version 1.0 — August 28, 2026
https://redduck.io/research/solana-development-patterns

Abstract

This paper catalogues eighteen recurring design patterns for Solana programs, drawn from production protocols and from post-mortems of exploits that the patterns would have prevented. Solana's execution model differs from the EVM in ways that reshape program architecture: a program does not own the accounts it receives, transactions are bounded in size and compute, there is no scheduler and no sender nonce, and every write rewrites the account in full. The patterns are grouped by the constraint they answer. Trust and inputs: Root of Trust, Identity Rejection, Defensive Computation, Fail-Fast, Boundary of Consent. Execution and incentives: Untrusted Executor, Commit-Execution Split, Action as Data, Virtual Execution, Pay-As-You-Act. State and data layout: Lazy Loading, One-Way State, Divide and Conquer, Complexity Asymmetry, Internal Oracle. Roles and dependencies: Role Inversion, Execution Layer Shift, Dependency Independency. Each pattern is stated as a design rule and illustrated with examples from protocols such as Wormhole, Helium, Drift, Squads, Monaco, Pyth and Jupiter, and, where relevant, with an anti-example from a real incident, including the Wormhole and Cashio exploits and the Meteora rate-limiter bypass. The catalogue is intended as a working vocabulary for engineers who design and audit Solana programs.

1. Root of Trust

The root of trust is the point at which a check stops being dependent on the caller. Everything that is verified before this point is peripheral.

On Solana your program does not own its data. Data (including accounts) that comes from the users within the transaction should prove nothing to you, because the senders of those accounts are also their creators.

If your checks rely solely on that data (which is a peripheral security practice), they could substitute the data in a way that the checks would pass just fine. This is true for accounts, prices, signatures, and anything else the caller gives you.

Instead

The place for the core check has to be something the caller could not create, something that only depends on the program. On Solana it would be a PDA of your program, a hardcoded address, or an account your program owns. The right data layout of an account that a user creates is never enough, because the data layout doesn't guarantee the data itself. The same source of truth serves every instruction. The attackers will find the one instruction where it doesn't.

A useful test: every piece of data the program receives gets the question: "if this one is fake, what happens to me?", and the answer points to the next piece. The chain either stops at something that cannot be faked, or the program has no root of trust.

Examples

- Wormhole lost 120,000 wETH (Wrapped Ether on Solana) in this way. To confirm that signatures were verified, the program read the list of instructions from an account. That account came from the sender too, and nobody checked its address. So the attacker passed his own.
- Cashio lost the same way. To mint, the program checked that the collateral account and the token account agreed with each other. Nothing checked that either of them belonged to the protocol's real mint. The attacker created his own accounts that agreed with each other and minted two billion CASH.
- Switchboard On-Demand does the Wormhole trick right. The oracle's signature is checked by a built-in program that returns nothing, so the client puts that check next to the oracle instruction in the same transaction. The program reads the list of instructions, but first it checks that the account it reads from has the known address. Then it checks that the neighbor is the built-in program and that the data it verified sits inside that same instruction.

2. Untrusted Executor (via Game Theory)

Nothing happens on-chain by itself. So every piece of background work in your protocol, like paying rewards or settling trades, needs someone to send a transaction and pay for it, as programs cannot be invoked by themselves. The simplest approach is to deploy your own bot, so that the protocol lives exactly as long as your server does.

However, that would be contradictory to the idea of decentralization. The better option instead is game theory.

If you make the job profitable to execute, users would find it out and implement their own bots to trigger the transactions. It’s reasonable to assume users would want to make extra money. In that case, it’s also reasonable to assume that many different users would run their bots and decentralize the execution of the jobs.

This approach can be applied not only to job execution but also to general protocol mechanics. If you design your protocol in a way that it makes certain actions more profitable than others, you would naturally shift the users’ behaviour in the desired way.

Examples

- Helium rewards. The oracle signs the total reward for a device, and the device’s account stores how much was already paid out. Anyone can call “claim” for any device, because the money goes to the owner no matter who called.
- Monaco does the same for trades: a trade only marks the order as filled, and anyone can later call the instruction that pays the makers. Governance works the same way: after the vote, anyone can execute the proposal.
- Symmetry V3 rebalances its index funds with a Dutch auction. When the weights drift, the fund offers the rebalance trade at a price that starts good for the fund and gets better for the taker every slot. Whoever takes it first keeps the difference. Bots watch every fund, and the rebalance is done by whoever agreed to the smallest reward.
- Drift, Marginfi and Kamino pay liquidators a slice of the collateral. Nobody runs a liquidation service; bots watch every position because each liquidation pays.
- Helium’s tuktuk makes the reward explicit. A scheduled task is an account with a prepaid reward and a start condition. Anyone runs it, takes the reward and closes the account, and the task queues the next one, so the schedule maintains itself.

3. Identity Rejection

The ultimate, most important goal of a program is to stay integral and produce the intended results — whether it’s a DEX that produces swaps and needs to keep its liquidity and pools in full integrity, or any other example.

What isn’t the ultimate goal is who, when, or why sends a transaction. This is a very peripheral question that shouldn’t make an impact on anything most of the time.

We call this pattern ‘Identity Rejection’. That is, it is the practice of rejecting any dependency on identity, circumstances and reasons, and rather focusing on the ‘what’, i.e. the action. “Is this transaction good for this protocol and its integrity?” is the question that should be asked, not “Is this a trustworthy address?”

Examples

- Protocol architecture: the untrusted executor (#2) is a subset of this pattern.
- Protocol security, the bad way: blocking wallet addresses to stop sniper bots. This wouldn’t last, as one address can be replaced with another one fairly simply, and you would still face the same problem.
- Protocol security, the good way: doing something that doesn’t depend on the identity but rather prevents the bad action itself. Meteora and Heaven use a launch tax instead. It is a fee that starts high and drops over the first few seconds, or

one that grows with the size of a single purchase. The time since launch and the trade size stay true no matter which address is doing it. This is the ‘what’, i.e. the focus on the action, not the identity.

Anti-examples

- Meteora’s Dynamic Bonding Curve limited snipers to one swap per transaction. But the rule only checked whether one specific instruction was called, not how many swaps actually happened (the real invariant). The second instruction, doing the identical job, was shipped later, but was never added to that check — the rate limiter missed it. Audits found up to sixteen swaps running inside a single transaction: the exact outcome the limiter was built to prevent.

4. Role Inversion

Many roles in a design are inherited, not chosen. The oracle writes prices, so it pays for every update. The bridge delivers a message, so it calls the receiver. The program keeps a sorted list, so it does the searching.

On Solana these roles could be the expensive ones. Role inversion is a way to bypass the wall, instead of attempting to break through it. The responsibility of actions shifts to the party that actually needs them, and the program only keeps the verification of the actions.

Examples

- Pyth oracles. Pyth used to write (push approach) every price to the data feed program at its own cost. Now the consumer who needs the price brings (pull approach) the signed update in the same transaction as his trade, and the program only verifies the signature. The oracle turned from a submitter into a verifier, nobody waits for anybody, and it is now easier and cheaper for the oracle.
- Wormhole delivery. The bridge does not call the receiving program. It only stores the verified message. The receiver comes for it in its own transaction, with its own accounts, calls the bridge and proves it is the correct address. The bridge turns from a caller into a place to read from, and it never needs to know which accounts the receiver wants.
- LayerZero moves the knowledge instead of the call. The receiving program has one extra instruction whose only job is to return the list of accounts it needs. The executor simulates it, builds the real transaction from the answer, and sends it. The one who knows the accounts names them; the one who delivers only carries them.
- Monaco’s order book gives away the search. The client finds where the new order goes and passes “after this one”. The program only checks that the neighbors are in the right order.

Anti-examples

- Not every flip works. Symmetry V2 turned its index fund from a taker into a maker: instead of trading to rebalance, it quotes to aggregators and lets other people’s trades move the weights. It sounded smart, but with this approach the rebalance happened only when someone else decided to trade. So that introduced a dependency on the aggregator, and the protocol had to wait for the aggregator to allow routing to you.

Note

The difference between Symmetry and Pyth is whether, after the flip, the one who needed the action was still in control. For Pyth — yes, for Symmetry — no.

5. Lazy Loading (Compute on Touch)

In traditional programming, lazy loading means you don't compute or fetch something until someone asks for it. On Solana it's usually not an optimization, but rather it's the only option. Programs cannot iterate over all accounts ever created. They can only write to the accounts passed into the transaction. And there is no scheduler, so every piece of work needs someone to send a transaction and pay for it.

So whenever your design says "for all holders", "every period", or "when the signers change, update everything", somebody has to send those transactions.

The lazy way (in a good sense): the result is not pushed to every account. Each account stores only what is needed to compute it later, and computes it when someone touches that account. The one who touches it pays for it, and that's the one who wanted the result. Usually this takes two fields: a running total, and a marker of how much of it this account has already seen. The result is the difference.

Examples

- Squads stale approvals. When signers change, the multisig doesn't walk through all open proposals. It sets one counter to the number of the latest proposal. When a proposal is read, its number is compared to the counter, and a lower number means stale. One comparison, no matter how many proposals there are.
- Fragmetric. A transfer hook updates the holder's reward balance at the moment of the transfer. The program runs on that account anyway, so the accounting comes for free.
- Helium rewards. The oracle signs the total a device has earned since the beginning, and the device's account stores how much was already paid. Nothing is pushed to a million devices every day. When someone claims, the program pays the difference and moves the marker up to the signed total.
- Drift funding. Perp markets charge funding every hour, but the program never walks through the positions. The market stores a running total of funding per unit of position, and each position stores the total it last saw. When a position is touched (a trade, a close, a liquidation), the difference times the size is settled first, and the marker moves to the current total.

Note

Important to remember: lazy means untouched accounts never update. If an action really needs to happen to update something explicitly (liquidations, expiry etc.), you would need a paid executor (#2) or to rework your design.

6. Complexity Asymmetry

To decide which algorithm is best to use, engineers consider the complexities in memory and computations, and based on the trade-offs between the two and the specific usage of operations (search, insert, etc.), they decide which algorithm is the most applicable for a specific problem.

Web3 and on-chain programs are not different, but they introduce one more variable for engineers to consider: on-chain complexity.

On-chain complexity can be very different from the overall complexity. You can have O(1) search and insert operations on-chain, while the general O(n) complexity is shifted fully onto the off-chain layer.

Examples

- Sorted linked list. That's a surprisingly frequently used data structure in web3, and the reason behind this is that even though the algorithm for finding and inserting an element into it is generally inefficient due to O(n) for both operations, it is possible to turn both operations into O(1) on-chain, while shifting the O(n) off-chain, making it extremely efficient for on-chain purposes. Here's how: let the client side find the position of the element and send it in, and have your program only check it. That way, the client side would spend O(n) computations to find the position, but your program would benefit from it by optimizing itself into O(1) complexity by just checking a few positions in a constant time.
- Monaco Protocol's order book is this list. The book is a fixed set of cells in one account, and each order points to its price neighbors by cell number. The client computes where the new order goes and sends "after this order". The program checks three things: the left neighbor is not cheaper, the right neighbor is cheaper, both are alive. That check costs the same at any depth, and the capacity is fixed, so nobody can inflate the book past what fits.
- Compressed NFTs (Bubblegum) do it for memory optimization. A million NFTs are one account with one merkle root hash. The full merkle tree lives off-chain with an indexer. To change one NFT, the client brings the leaf and about twenty hashes along its path, and the program recomputes the root and compares. On-chain memory is O(1), the on-chain check is O(log n), and the O(n) part lives with the indexer.

Note

Merkle trees such as in Bubblegum are the worst data structure you can imagine using for a normal traditional program. With web3 programs however, it is one of the most popular and frequently used ones.

Important to remember: the reason why we don't want any high complexity (higher than O(log n)) on-chain is because it makes it easy to just spam the program with data and make it no longer fit the compute limit, after which nobody could use it. This is also frequently referred to as a "DOS attack".

7. Execution Layer Shift

Solana is already extremely fast, so block speed is not a reason for choosing L2 over L1. Unless, of course, the idea is to build a real-time game like Counter-Strike — then it could be justified.

Instead, the choice of L2 over L1 is justified in case the goal is to build a protocol with less technical constraints than on L1 (for example, the program may need to reference more accounts, or users may need to execute transactions with more instructions than allowed on L1), and at the same time the block security and finalization trade-off doesn't matter to the project much. If it does, the alternative is to think of possible ways to optimize the program and keep it on L1.

Examples

- Zeebit is an on-chain casino. A spin has to feel instant, and a two-step random number on L1 does not. Speed was the only thing missing, so Zeebit did not leave L1. The player deposits into Zeebit's program on Solana, and the money stays there. For one round, only the state of that round is handed to an ephemeral rollup (MagicBlock), where moves are instant. When the round ends, the result is written back to Solana and the payout happens there.
- Zeta went to an L2. Its cross-margin check touches every market a trader holds, and the limit on accounts per transaction capped how many that could be. That is the constraint case: it moved trading to its own L2 (Bullet), kept settlement and data on Solana, and accepted that a withdrawal to L1 takes longer than a trade.
- Helium tried both ways. It had its own chain and moved onto Solana. A million devices fit as compressed NFTs, and rewards fit as per-device claims instead of daily payouts. What did not fit was reshaped, not moved.

8. Divide and Conquer

Traditionally, in algorithms, "divide and conquer" means splitting a task into independent parts, solving each part on its own, and merging the results. On Solana, the same pattern is used but for the purposes of architecting programs around the chain's constraints.

Context: on Solana, transactions are limited in size, programs are limited in compute (amount of operations), and every single change to a dynamic structure in an account rewrites the account fully — even if you change one byte out of a thousand, you write a thousand bytes again.

The pattern: divide data into partitions that can be stored separately, while retaining a central storage for the result of those partitions. Users interact only with the very partitions they need in order to execute their actions, making the transactions as minimal as they can be.

Depending on the program specifics, Pay-As-You-Act (#17) can be utilized in order to update the central storage within the same transaction where the partitioned state is accessed. If it's not necessary or not optimal, the Pay-As-You-Act pattern may not be used.

Examples

- Orca DEX. All ticks are stored virtually, with only the active ones referenced in the programs directly. The ones that are inactive are not referenced in the program instructions, thus significantly optimizing the transactions.
- Compressed NFTs (Bubblegum). A million NFTs live in one account as a Merkle tree. Leaves are hashed in pairs, level by level, up to a single root, and that root stands for all million. To prove or change one NFT, the client brings the leaf and

the hashes along its path, about twenty for a million leaves. The program recomputes the root and compares it with the stored one. The top levels of the tree are stored in the account (the canopy), so the client sends only the bottom of the path.

- ZK Compression does the same for any account and adds one more merge on top: proofs for several accounts fold into one proof of fixed size, however many the transaction should touch.

9. Commit-Execution Split

Some actions that users commit cannot be executed in the same instant, whether for security or architectural reasons. For those cases, the best practical approach is to split the actions into commit and execute phases.

The best example is your typical program with a random outcome. You can’t implement a secure program that lets a user get the random outcome in the same transaction as they’re submitting it. Because if you did, the user could simply simulate transactions many times and only send them those few times when they actually succeed to produce the winning outcome for them.

In fact, because of it, you also can’t let them have the authority of being the sender of that ‘execute’ part of the transaction at all. So naturally, you want to split it into ‘commit’ and ‘execute’ types of transactions, because you can’t let them have the ‘execute’ part.

Examples

- Switchboard VRF (coinflip). In the first transaction players pick a side, pay, and the program creates a request for a random number. Nobody knows the number yet, not the player, not the operator. In the second transaction the oracle (not the player!) writes the number into that request, and the program settles the bet, the resolution of which is already locked. The player cannot start a new round until this one is settled, so there is no way to abandon a losing bet.
- Parcl. Orders collect over an interval and clear at one price, so nobody gains from being first.
- Drift. Before the pool fills an order, makers get a short window to bid for it.

10. Action as Data

Some actions need a signer who is offline, or a condition that isn’t true yet. This is where the ‘Action as Data’ pattern comes in.

On Solana, a transaction signature is valid only for a short period of about a minute. So the common EVM flow does not work: sign an order once, let a server submit it later. No signed orders waiting for a taker, no approvals collected over a week. A durable nonce keeps one prebuilt transaction alive longer, but the signature still cannot be reused, and the accounts cannot change.

One workaround is to keep the user’s request on a server and let it act with its own key. But then the server signs transactions that move the user’s money. It can crash, censor, or be taken over.

‘Action as Data’ is the pattern of storing the intended user’s transaction on-chain as transaction metadata — target program address, account list, instruction payload

— instead of relying on the user to submit the transaction when they're ready. A Program-Derived Address (PDA) holds that record, and the action is not executed at the moment it is stored. Later, when the conditions are met, a keeper (such as an untrusted executor, #2) submits the transaction, and the PDA approves it with the invoke_signed instruction. The action that the user put on-chain before its execution was always public: anyone could read it and execute it once the condition was met.

Examples

- Squads creates an account for every proposed transaction, storing the full list of instructions inside it. Members approve the proposal one by one. Once enough members vote yes, anyone can trigger the execution (even if they aren't a member), and the program named 'vault' will execute the stored instructions.
- Realms does the same, by keeping governance instructions inside the proposal account itself. Once a vote passes, any user could trigger the stored instructions to execute.
- A Jupiter limit order is an account with a price, DCA is an account with a schedule, and when the price or the time comes, anyone can fill it.

Note

The only trade-off in this pattern is that the rent for every pending action has to be paid, and the funds are locked until it runs. So every action needs a way to cancel or expire.

11. One-Way State

Sometimes the program logic requires one transaction to happen strictly before another, otherwise the state becomes invalid. On Solana there is no sender nonce, so two transactions can land in either order if sent simultaneously.

EVM contracts rely on transaction nonces to enforce sequential order, and track message ids natively. Trying to replicate this on Solana would be an antipattern. You would allocate a separate account to store the number and pay rent for it, and it still wouldn't be bullet-proof: used ids and nonces give uniqueness, but they do not guarantee commutativity (A + B = B + A).

The One-Way State pattern flips the design: instead of attempting to restore message delivery order, you design in a way that order stops mattering at all. The message carries the destination, not the step: the total earned so far, not today's reward. Deltas break this even when they only move forward: +5 applied twice is not +5 applied once. A total applied twice is just the total.

The program only accepts actions that move state progress forward or restrict capabilities, strictly rejecting attempts to roll back the value or broaden access. When the state can only move in one direction (never down, never wider), a delayed, re-submitted transaction is no longer a security risk. The program never compares two messages with each other, it only checks the message against the current state. You don't have to ask the question "which one came later" when you can ask the question "has this already been taken into account". When you're moving in one direction, you can't suddenly find yourself at point 80 if you've already passed point 100.

Examples

- Helium rewards. The oracle signs the total a device has earned since the beginning. The device’s account stores how much was already paid. A claim pays the difference and moves the paid amount up to the signed total. Send it twice, and the second claim pays zero. Bring an older signature with a smaller total, and it pays nothing, because the paid amount is already above it. The counter never goes down, so nobody can take back what was paid.
- Squads does the same with rights: when signers change, one counter moves to the latest proposal, and every older proposal can be cancelled but not approved.
- Drift’s binary markets turn reduce-only near expiry: you can close a position but not open or grow one.

Note

The One-Way State survives a simple test: any two transactions, swapped, land on the same final state.

12. Dependency Independency

System resilience requires minimal dependencies, and for those that are difficult to avoid, explicit decoupling paths. With this approach, the protocol can achieve what we call ‘Dependency Independency’.

Examples

- Zeebit on MagicBlock. The casino needs instant moves, so it runs rounds on an ephemeral rollup (a temporary, fast layer-2 network built just for short-lived tasks). But the player deposits into Zeebit’s program on Solana, and the money stays there. Only the state of the current round is handed to the rollup. When the round ends, the result is written back to Solana and the payout happens there. The dependency lasts one round and never touches the funds, so the exit is the normal flow, not an emergency plan.
- Drift keeps the oracle of every market as a setting: moving a market to another feed is an admin instruction, not a deploy.

Anti-examples

- UXD kept its hedge on Mango Markets with no way to move it, and when Mango was hacked there was nothing left to do but close the product.
- Serum’s upgrade key was held by FTX, so when FTX collapsed the only exit left was a fork.

Note

The test is simple: would the product make sense without this dependency? If yes, the dependency is just a part of it, and parts can get replaced. If not, there is nothing to plan. Sometimes, protocols exist just because of their dependency, and this is okay. Aura exists only because Balancer does; if Balancer disappears, Aura loses nothing, it simply wouldn’t be there in the first place without Balancer. It’s an EVM example, but the concept is the same for any chain and frankly, any product in any field.

13. Defensive Computation

Due to the nature of Web3, programs cannot trust anyone and have to consider the worst-case scenario at any point of time. Because of that, the programs have to do what we call ‘Defensive Computation’: choose the lowest possible boundary to work with. It can be the lowest evaluation of an asset’s price when accepting it as collateral, or the highest evaluation of existing debt.

That is, in order to avoid the risk of taking inaccurate data, the data has to be taken into account at the worst possible version.

Examples

- Marginfi. The oracle gives a price and a confidence band. To value a user’s collateral, the program takes the bottom of the band. To value the debt, the top. When the band widens, the position looks worse from both sides at once: the user can borrow less, and the liquidation line moves closer. The protocol never lends against the optimistic reading.
- Drift refuses to fill when the band is too wide relative to the price, and rejects prices that are too old.
- Kamino rejects a USDC price far from $1, no matter what the oracle says.
- Helium prices Data Credits at the bottom of the band.

14. Internal Oracle

Not all programs really require an external oracle in order to receive the necessary state to work with. In some cases, programs can utilize their internal state if properly structured. Structuring the internal state to function as an ‘oracle’ that provides an outside-chain view on the data is what we call ‘Internal Oracle’. Programs derive state instead of attempting to calculate it or receive it from the outside.

Examples

- Raydium TWAP (and the protocols that use it). Raydium already has the data on liquidity, positions, swaps and prices. And even though pricing data is commonly thought of as a reason to include an oracle, in some cases a program can simply structure its internal state to produce the indexed data that can later on act as an oracle. TWAP is that indexed state, and if a program only works with prices of tokens that exist on Solana, then it may not even need any oracles, and could just utilize Solana’s DEXs and their TWAPs.
- Realms utilizes the same pattern for the “already voted” state. Each vote is an account at an address computed from the proposal and the voter (a PDA). Nobody separately stores who voted and who didn’t. But if the address is taken, it logically means that the voter has already voted. This is a definition of deriving a state from another state, instead of relying on computing it or obtaining it from the outside.

15. Fail-Fast

Some inputs are not good enough to act on: the oracle doesn’t give a high confidence score about the price, the price jumped too far since the last slot, or the random number has not arrived yet.

On Solana, just as on any other blockchain where transactions are atomic, your program cannot wait for a better input. It has to decide now, with the data it was given. If acting is its only output, it will act on bad data too. So a failure to proceed, alongside its reason, must be an output as well.

Examples

- Drift. The program grades the oracle before every action. The band around the price (Pyth's confidence interval) is too wide: uncertain. The update is too old: stale. The price is too far from its own five-minute average: too volatile. Each action has its own rule. The pool fills only on a price that passed every check.
- Kamino keeps a plausible range per token. A USDC price far from $1 is refused, even if the oracle is confident.
- A coin flip on Switchboard VRF stores its refusal. While a round is open, the player cannot start a new one. Only the oracle's answer closes it.

Note

Sometimes a trade on a live market is refused because the oracle is not sure. A user can lose some trades. But at least the protocol is not lost when the oracle is wrong.

16. Virtual Execution

Sometimes you can't predict or know the exact execution details of an action. To solve this problem, protocols apply what we call 'Virtual Execution'. That is, the execution is made virtually, with the real execution performed later on when the details of that execution become known.

Examples

- Monaco Protocol's order book. When a match happens between a maker and a taker, the protocol doesn't yet know how to send the tokens to the maker, because in order to do that it would have to know the accounts of the maker, for which it would require storing additional data and having the taker compute the account addresses. So it marks them as eligible for sending, and later on, when someone is ready to perform the execution, they turn the virtually executed send into a real one.
- ZK Compression drops each spent record into a queue on the spot. Anyone running a node later moves the queue into the tree, for a reward.

17. Pay-As-You-Act

Protocols very frequently use the pattern of distributing computations across users and having them pay only for their part. We call this 'Pay-As-You-Act', as with this pattern everyone pays for their own piece of the work, at the moment of their own action, instead of paying for the whole protocol at once.

Take an election as an example. The bad design stores the votes: when the voting period closes, someone walks through all of them to find the winner. That iterating transaction pays for every vote, and nobody would volunteer to execute that transaction, as it's a common effort shifted onto a single person.

Instead, the good design gives each action its piece of the final step. When a vote comes in, the program immediately increases the candidate's count and compares it with the leader's count, without waiting for the 'final resolution' window. If the count

is higher, this candidate becomes the new leader. Each voter executes one addition and one comparison, triggered by that voter. When the voting period is finished, the protocol already knows the winner, as the leader is already recorded in the state, and the closing transaction reads one field.

Examples

- Hxro's parimutuel (pool betting) markets. A bet on "up" or "down" adds its amount to that side's total. When the round ends, the oracle names the winning side, and the losing pool goes to the winners. Each winner claims on their own: their share is their bet divided by the winning total. Bettors paid for the totals when they bet, and each winner pays for his own payout when he claims.

Note

Most of the "at the end" steps can be embedded into intermediate in-process transactions: an election winner, a payout, an average number. The key to the pattern is to store the answer instead of inputs, and let every action update it. The answer has to be something that one action can update in one step: a count, a total, a maximum, an average, a leader.

18. Boundary of Consent

On-chain state can change between signing and execution. An exact expected result may already be stale when the transaction arrives.

The boundaries can be signed together with the intended execution data: maximum input, minimum output, limit price, maximum oracle age or deadline. Execution inside the accepted boundaries is valid, but execution outside of them fails. For example, in swaps, this boundary is slippage. It defines acceptable price movement rather than predicting it. The same pattern applies to orders, oracle reads, liquidations etc.

Examples

- Jupiter: an expected output can be calculated right on the spot, but the user in fact signs the minimum acceptable output they can agree to. The swap executes while the result stays above that minimum, but completely fails when it is below the acceptable minimum.

References

[1] CertiK. Wormhole Bridge Exploit Incident Analysis. 2022. — https://www.certik.com/blog/wormhole-bridge-exploit-incident-analysis

[2] Halborn. Explained: The Wormhole Hack (February 2022). — https://www.halborn.com/blog/post/explained-the-wormhole-hack-february-2022

[3] CertiK. Cashio App Incident Analysis. 2022. — https://www.certik.com/resources/blog/cashio-app-incident-analysis

[4] Halborn. Explained: The Cashio Hack (March 2022). — https://www.halborn.com/blog/post/explained-the-cashio-hack-march-2022

[5] Code4rena. Meteora — Dynamic Bonding Curve. Audit Report. 2025. — https://code4rena.com/reports/2025-08-meteora-dynamic-bonding-curve

[6] Solana Foundation. Precompiled Programs (Ed25519, Secp256k1). — https://solana.com/docs/core/programs/precompiles

[7] Switchboard. switchboard-on-demand-client (Rust crate documentation). — https://docs.rs/switchboard-on-demand-client

[8] Helium. helium-program-library (Lazy Distributor, Rewards Oracle). — https://github.com/helium/helium-program-library

[9] Helium. tuktuk: Permissionless crank turner on Solana. — https://github.com/helium/tuktuk

[10] Solana Foundation. Case Study: A Technical Deep Dive on Helium. — https://solana.com/news/case-study-helium-technical-guide

[11] LayerZero Labs. LayerZero V2 Solana OApp Reference. — https://docs.layerzero.network/v2/developers/solana/oapp/overview

[12] Squads Protocol. Accounts Reference (stale_transaction_index). — https://docs.squads.so/main/development/reference/accounts

[13] Neodyme. Security Audit — Squads Multisig v4. — https://neodyme.io/reports/Squads-Multisig-v4.pdf

[14] Monaco Protocol. Monaco Protocol SDK. — https://github.com/MonacoProtocol/sdk