Fair Outcomes for Solana: Bounded Atomic Fair DeFi and Native Fair Validator Execution

100% On-Chain Atomic Composable Trading With the Rest of the State Machine

Draft prepared for the Fair Outcomes / SolanaCDN working group

Complete protocol revision with account-set closure, bounded atomicity modes, Fair Proof Bundle finality, committee-failure, and launch-parameter rules - May 8, 2026

Contents

Abstract

High-throughput blockchains increasingly route economically sensitive transactions through specialized transaction supply chains: RPC peering agreements, private relay paths, bundle systems, priority-fee markets, aggregators, RFQ venues, proprietary automated market makers, and off-chain solver networks. These systems improve landing probability, liquidity discovery, and sometimes atomicity, but they also create fairness failures. A user may be harmed by privileged ordering, hidden private orderflow, selective inclusion, normal-lane insertion around fair traffic, or a quote/fill boundary in which one economic outcome is displayed and another economic outcome is settled.

This paper specifies Fair Outcomes as a bounded, validator-enforced Fair DeFi protocol for Solana-style high-throughput execution. The core framing is:

```
Fair sequencing fixes who can order transactions.
On-chain atomic composable trading fixes what outcome is allowed to settle.
A complete fair DeFi system needs both.
```

The phrase “100% on-chain atomic composable trading” is used in a precise sense. It does not require every non-binding price model, routing search, or market-maker inventory computation to run on-chain. Off-chain systems may discover prices, compute candidate routes, optimize fills, and propose solutions. The requirement is that every binding authorization, candidate universe, executable offer, selection or clearing rule, settlement transfer, fee, rebate, refund, bond consequence, execution report, and downstream DeFi composition used to settle a fair trade is verified and committed by the Solana state machine. Off-chain systems may propose. The chain decides, verifies, settles, accounts, reports, and applies consequences.

This revision fixes the remaining production blockers in earlier drafts. It adds an account-set closure rule for Solana account-lock compatibility, three explicit atomicity modes, batch-auction size classes, Fair Proof Bundle vote/finality rules, a committee-failure ladder, hard launch parameters for fair/normal fences and composability, a bounded MVP scope, stronger replay invalidity rules, and additional formal invariants for Fair DeFi settlement. It preserves the native fair validator-client design: fair encrypted pre-transaction envelopes, deterministic admission, post-admission randomness, threshold reveal, post-reveal validation, fair/normal account fencing, dependency-graph scheduling, forced inclusion, Fair Proof Bundles, data availability, and fair-domain replay verification.

The result is not an unbounded claim that arbitrary DeFi workflows or arbitrary batch auctions can be atomically settled on mainnet. The result is a buildable protocol with explicit bounds. A trade, batch, or sequence can be labeled `FairDeFiSettled` only if its account set is closed before fair ordering, its compute and CPI plan fit the selected atomicity mode, its Fair Proof Bundle data is reconstructable before voting/finality, its committee reveal state is deterministic, and its on-chain settlement program enforces the user’s signed constraints. Under broad validator-client adoption, invalid fair-domain claims are rejected during replay rather than merely punished by reputation.

Keywords: Solana, Fair Outcomes, Fair DeFi, on-chain trading, atomic composability, state machine, validator client, MEV, encrypted mempool, threshold decryption, account-set closure, Fair Proof Bundle availability, batch auction, RFQ, PropAMM, quote commitment, solver competition, deterministic settlement.

1 Introduction

Transaction ordering is a market-structure primitive. In DeFi, execution position determines whether a user receives the price they approved, is sandwiched, is routed through a worse venue, or is silently excluded. The original MEV literature showed that priority gas auctions and block-producer ordering discretion can create both user-level losses and consensus-layer instability [1]. Later work on order-fair consensus formalized fairness constraints for Byzantine replicated systems [2,3]. Encrypted mempools and threshold decryption provide a complementary technique: hide transaction contents until ordering is fixed [4].

Solana adds a distinct context. It targets high throughput, low latency, parallel execution, and a transaction pipeline in which priority fees, stake-weighted quality of service, account locks, compute budgets, and application-layer routing interact. Solana transactions contain one or more instructions, authorizing signatures, and a recent blockhash; the network processes all instructions in a transaction together, and if any instruction fails the whole transaction fails with state changes reverted [7]. Solana cross-program invocation enables composability because one program can call another program during execution, with caller privileges and compute budget flowing through the runtime [19]. These properties make Solana uniquely suitable for atomic composable trading, but they also make ordering and account-state interaction economically important.

A pure fair-sequencing protocol is necessary but incomplete. It can make content-based reordering hard or impossible inside a certified fair path. It can prevent normal-lane insertion around protected accounts. It can prove admission, ordering, reveal, scheduling, and execution boundaries. But if the final trade outcome is still determined by an off-chain quote, a hidden route, a private solver decision, or a venue that changes terms after user approval, the user can still be harmed.

This paper therefore upgrades Fair Outcomes from a fair transaction-supply-chain protocol into a **Fair DeFi state-machine protocol**.

The core design objective is:

```
Every economically binding part of a fair trade must be on-chain,
atomic where the selected mode requires atomicity, deterministic, and
composable with the rest of the state machine within declared runtime bounds.
```

This does not mean every non-binding price-discovery computation must happen on-chain. PropAMMs may still compute model prices off-chain. Aggregators may still search routes off-chain. Solvers may still compute optimizations off-chain. Market makers may still manage inventory off-chain. The rule is narrower and more important:

```
Off-chain systems may propose.
The chain verifies the candidate universe.
The chain selects or verifies the selected result.
The chain settles, accounts, reports, refunds, and applies bond consequences.
```

The protocol therefore replaces the ambiguous phrase “off-chain quote that later settles” with a stricter primitive: a state-machine-verified settlement object. A fair trade is not fair merely because it was routed through a fair validator. It is fair only if the validator-ordering proof and the on-chain settlement proof both validate.

1.1 From fair order to fair outcome

A fair DeFi outcome requires two layers.

First, a native fair validator client enforces the transaction-supply-chain rules:

```
witnessed ingress
fair admission
post-admission randomness
encrypted reveal
fair/normal account fences
dependency-graph scheduling
forced inclusion
Fair Proof Bundle publication and verification
```

Second, on-chain Fair DeFi programs enforce the economic-output rules:

```
user intent constraints
binding maker or solver offers
state-bound quote validity
deterministic fill selection
batch clearing
atomic token settlement
fee and rebate accounting
downstream DeFi composition
execution reports
bond and refund settlement
```

A validator can no longer sell transaction position inside the fair domain. A route provider can no longer show one route and submit another without proof. A venue can no longer advertise a firm quote and settle worse terms without a violation. A solver can no longer insert a private bundle around the user. The final trade output becomes a deterministic property of an on-chain state transition.

1.2 Central claim

The central claim of this revision is:

```
A Solana-style high-throughput blockchain can make economically sensitive
DeFi execution fairer by combining validator-enforced fair sequencing with
on-chain atomic composable trading, so that both order and outcome are
commitment-bound, state-machine-verified, and externally auditable.
```

The protocol does not eliminate arbitrage, inventory risk, market-maker spread, cross-market latency, or informed trading. It does not force every market maker to provide firm quotes. It does not guarantee stale transactions execute after blockhash or quote expiry. It does eliminate or make replay-invalid specific supply-chain failures inside certified fair domains: content-based ordering before reveal, paid intra-batch order, private fair-domain bundles, unjustified exclusion, normal-lane insertion around fair batches, hidden route changes, quote/fill degradation beyond committed terms, and off-chain settlement ambiguity.

1.3 Adoption assumption

This paper assumes the validator-client adoption problem is solved for certified fair domains:

```
A0: A supermajority of block-producing stake, enough RPC/gateway ingress,
and major fair-domain wallets and DeFi applications run or route to a
validator client that verifies Fair Outcomes proof objects during replay.
```

A0 does not mean every Solana transaction becomes fair-domain traffic. It means that when a validator, application, route, or venue claims `FairSequenced` or `FairDeFiSettled`, that claim is enforceable by replay rules, Fair Proof Bundles, and on-chain program validity.

1.4 Contributions

This paper makes ten contributions.

1. **Fair DeFi thesis.** Fair sequencing is combined with 100% on-chain atomic composable trading so the chain, not an off-chain supply chain, computes the final trade outcome.
2. **Native validator-client specification.** The paper specifies fair ingress, envelope queues, deterministic admission, post-admission randomness, threshold decryption, account fences, dependency scheduling, forced inclusion, and Fair Proof Bundles.
3. **On-chain intent model.** User authorization is expressed as an on-chain intent with constraints, nullifiers, expiry, allowed venues, quote mode, settlement mode, and fail-closed policy.
4. **Binding offer model.** Makers, solvers, PropAMMs, AMMs, and RFQ venues submit fill offers or quote commitments that are verifiable on-chain and backed by bonds where needed.
5. **Atomic settlement model.** Settlement updates token balances, fees, rebates, venue inventory, user output, execution reports, and downstream DeFi state in a single atomic transition.
6. **Batch clearing model.** The system supports single-intent best-fill settlement and multi-intent batch auctions with deterministic clearing, surplus sharing, and state-bound execution.
7. **Composable state-machine integration.** The protocol specifies how fair swaps compose atomically with lending, margin, liquidations, oracles, vaults, governance, and bridges.
8. **Formal invariants.** The paper states safety and liveness invariants for fair order, settlement correctness, quote integrity, fence isolation, Fair Proof Bundle availability, and on-chain atomicity.
9. **Economic design.** The paper specifies fee separation, validator revenue replacement, quote-risk pricing, user revert insurance, solver migration, fair-lane congestion controls, and cross-domain MEV mitigation.
10. **Deployment plan.** The paper defines an implementation path from single-intent on-chain settlement to batch auctions and then ecosystem-wide Fair DeFi standards.

2 Background and Motivation

2.1 The transaction-supply-chain problem

Modern DeFi execution is not a simple path from wallet to block. A transaction may pass through a wallet, swap app, aggregator, RPC provider, private relay, market maker, RFQ venue, bundle engine, validator, and one or more on-chain programs. Each handoff can introduce a fairness failure:

- The user may see a quote that is not actually binding.
- The aggregator may change route after user approval.
- The market maker may reserve last look or internalize orderflow.
- The RPC or relay may leak information.
- The validator may reorder, exclude, or insert conflicting transactions.
- A normal-lane transaction may touch the same pool around a fair batch.
- A private bundle may create atomicity for searchers but not for users.
- A downstream program may observe different state from the user-approved quote context.

Fair sequencing constrains validator discretion. It does not alone make a quote binding or force the final state transition to match user intent. Fair DeFi therefore moves the binding economic logic on-chain.

2.2 Solana transaction and composability constraints

Solana's native execution model is relevant for three reasons.

First, a Solana transaction is already atomic at the instruction level: all instructions succeed or all state changes revert [7]. That is the foundation for on-chain atomic settlement.

Second, Solana composability is enabled by cross-program invocation. One program can invoke another during execution, and privileges and compute budget are governed by runtime rules [19]. That makes it possible for a Fair Settlement program to call token programs, AMMs, lending programs, vaults, margin engines, oracle-checking programs, and execution-report programs within one transition.

Third, Solana validators need visible transaction structure for ordinary banking: signatures, account metas, recent blockhash or nonce state, compute budget, and account locks. Therefore a fully encrypted fair object cannot be treated as a normal transaction before reveal. It must be a pre-transaction object accepted by a dedicated fair path, then decrypted into ordinary signed transactions after order is committed.

2.3 Private atomicity and the user-alignment problem

Private bundle infrastructure provides useful services: low-latency landing, all-or-nothing execution, and multi-transaction coordination. Jito's documentation describes bundles as lists of up to five transactions that execute sequentially and atomically, with all-or-nothing behavior inside a slot [12].

The problem is not atomicity. Atomicity is good. The problem is who controls it and what it is used for.

Private bundles often give atomicity to searchers, not to users. They can allow backruns, arbitrage, liquidations, or multi-leg strategies to execute safely. But the user whose trade creates the opportunity may not receive protection, surplus sharing, or route transparency.

Fair DeFi replaces private bundle atomicity with user-authored on-chain atomicity:

```
The user signs the intent.
The solver competes to improve that intent.
The state machine selects and settles a valid outcome.
No unrelated third party can insert itself into the atomic object.
```

2.4 Quote/fill ambiguity

Current aggregator flows often begin with a route quote. Jupiter's documentation, for example, describes a quote endpoint that returns route plans from a routing engine based on input mint, output mint, amount, and slippage parameters [15]. This is useful, but a route quote is not the same as a state-machine-enforced settlement guarantee.

A quote/fill fairness failure occurs when:

```
displayed expected output != binding commitment != final output
```

Fair DeFi makes the user-approved constraints binding on-chain. The user does not merely approve a quote display; the user authorizes a settlement predicate. If the predicate is not satisfied, settlement fails or produces a reason-coded violation.

2.5 PropAMMs and active liquidity

PropAMMs and RFQ venues may improve liquidity by using off-chain predictive models, inventory management, and active quote updates. This is legitimate market making. The protocol should not force every price model on-chain or require market makers to accept unbounded stale-price risk.

The Fair DeFi distinction is:

```
Models can be off-chain.
Executable commitments must be on-chain.
```

A PropAMM may compute a quote off-chain, but if it wants the wallet and validator client to label the transaction `FairDeFiSettled`, the enforceable quote class, expiry, state bounds, and settlement predicate must be represented on-chain.

3 Design Thesis: The Chain Computes the Outcome

The new design is built around a state-machine principle.

Let `S_t` be Solana state at the start of a fair execution segment. Let `B` be a fair batch of revealed intents, offers, and transactions. Let `C` be the deterministic clearing and settlement logic implemented by Fair DeFi programs. Let `Delta` be the state transition.

```
Delta = ApplyFairDeFi(S_t, B, C)
S_{t+1} = Commit(S_t, Delta)
```

A fair trade is valid only if `Delta` satisfies:

```
user intent constraints
binding offer constraints
quote class constraints
account fence constraints
state-bound freshness constraints
fee and rebate constraints
solver surplus rules
downstream program preconditions
execution report emission
```

No off-chain actor can later reinterpret the fill. No hidden route can be substituted. No venue can settle a worse firm quote and call it slippage. No private searcher can attach a third-party bundle around the user unless the user's intent delegates that role and the delegation is enforced on-chain.

3.1 What "100% on-chain" means

This paper uses "100% on-chain atomic composable trading" in the following precise sense:

1. **Authorization is on-chain.** User intents, constraints, nullifiers, expiry, allowed venues, max fees, and settlement mode are represented on-chain or in a signed transaction verified by an on-chain program.
2. **Executable offers are on-chain.** A maker, solver, RFQ venue, AMM, or PropAMM offer that can settle a fair trade must be posted or referenced in an on-chain-verifiable form.
3. **Selection is on-chain.** The winning offer, clearing price, or route set is selected by deterministic program logic from valid committed inputs.
4. **Settlement is on-chain.** Token transfers, inventory updates, fees, rebates, refunds, and reports are committed by the runtime.
5. **Composition is on-chain.** Lending, margin, liquidation, vault, oracle, and governance interactions occur through instructions and CPIs in the same state transition when required.
6. **Failure is on-chain.** If constraints are not satisfied, the program returns a deterministic error or emits a reason-coded failure report.

The protocol does **not** require every non-binding quote computation or route search to happen on-chain. It requires every binding input to the final fair outcome to be verifiable on-chain.

3.2 Why this changes the product

The original Fair Outcomes framing could be described as:

```
anti-MEV validator client + quote commitments
```

The new framing is:

```
fair execution layer for Solana DeFi
```

The validator client protects admission and order. Fair DeFi programs protect outcome and composability. Together they can serve as a replacement for both private fair-domain orderflow and ambiguous off-chain quote/fill systems.

3.3 Boundedness principle

The protocol is intentionally bounded. A trade may be labeled `FairDeFiSettled` only if it satisfies the selected execution mode's bounds before fair ordering and replay validation. The bounds are not implementation details; they are part of the safety property.

```
FairDeFiSettled(x) iff
    FairSequenced(x)
    AND AccountSetClosed(x)
    AND ComputePlanBounded(x)
    AND AtomicityModeValid(x)
    AND SettlementProgramValid(x)
    AND FairProofBundleFinalitySatisfied(x)
    AND ExecutionReportComplete(x)
```

A workflow that cannot satisfy these predicates may still be a useful ordinary DeFi workflow, but it must not be labeled `FairDeFiSettled`. This prevents the protocol from overclaiming beyond Solana's transaction, account, compute, and CPI limits.

3.4 What "100% on-chain" does not mean

The phrase does not mean that every market-maker model update, every route search, every solver optimization, every price forecast, or every candidate auction solution is executed on-chain. That would be unnecessarily expensive and would reduce market-maker quality.

The phrase means that the economically binding part of the transaction is on-chain:

```
user authorization;
intent constraints;
```

```
candidate account universe;
offer validity;
selection or clearing certificate;
settlement transfers;
downstream CPI plan;
fee, rebate, surplus, refund, and bond consequences;
execution report;
replay-verifiable proof objects.
```

This is the distinction that makes the design viable. The protocol uses off-chain computation for discovery and optimization, and on-chain verification for settlement truth.

3.5 Solana execution constraints as protocol constraints

Solana transactions are atomic across their instructions: if any instruction fails, the transaction fails and state changes are reverted. Solana transactions also have explicit transaction-size, account-count, blockhash-expiry, instruction-count, compute, and CPI limits. CPIs enable program-to-program composability, but they share compute budget and cannot exceed runtime privilege, stack-depth, account-info, instruction-data, and return-data constraints.

Fair Outcomes treats those facts as protocol constraints:

```
No settlement may select or invoke accounts outside the predeclared closure.
No settlement may exceed its declared compute and CPI plan.
No settlement may claim NATIVE_SINGLE_TX if it requires multiple txs.
No settlement may claim FAIR_ATOMIC_SEQUENCE without sequence replay rules.
No large auction may claim one-transition atomicity if staged escrow is required.
```

The result is a stronger and safer claim: the protocol works for bounded Fair DeFi objects whose accounts, compute, CPI plan, Fair Proof Bundle data, and settlement mode are known and verifiable before the validator accepts the fair-domain claim.

4 System Architecture

The architecture has four layers.

```
Layer 1: Fair Ingress and Witnessing
Layer 2: Native Fair Validator Client
Layer 3: Fair DeFi On-Chain Programs
Layer 4: Indexers, Scorecards, Disputes, and Governance
```

4.1 Layer 1: Fair ingress and witnessing

Users and apps submit `FairTxEnvelope` objects to certified gateways. Witnesses issue signed ingress receipts. Gateways issue propagation receipts. Receipt quorums prove that a commitment was observed before a cutoff. This layer supports censorship evidence, forced inclusion, and fail-closed routing.

4.2 Layer 2: Native fair validator client

The validator client implements:

```
fair envelope queue
pre-reveal admission
admission and exclusion roots
commit-before-randomness ordering
threshold reveal
post-reveal transaction validation
```

```
fair/normal account fence
dependency-graph scheduler
forced inclusion
Fair Proof Bundle generation
Fair Proof Bundle availability checks
fair-domain replay validity
```

This layer prevents content-based ordering and normal-lane insertion around protected fair accounts.

4.3 Layer 3: Fair DeFi programs

The on-chain program suite includes:

```
FairIntent Program
FairOffer Program
FairAuction Program
FairSettlement Program
FairExecutionReport Program
FairBond and Insurance Program
```

These programs define the economic state transition. They ensure that user intent constraints, offer commitments, quote classes, state bounds, and downstream DeFi composition are enforced by the runtime.

4.4 Layer 4: Indexers and governance

Indexers verify Fair Proof Bundles, publish fair execution reports, compute venue and aggregator scores, monitor residual MEV, and support disputes. Governance manages domain manifests, fee parameters, quote-bond rules, admission quotas, committee rosters, Fair Proof Bundle retention, and solver-surplus parameters.

4.5 End-to-end flow

```
1. User creates a FairIntent.
2. App or wallet submits a FairTxEnvelope.
3. Witnesses issue ingress receipts.
4. Validator admits the envelope into a fair batch.
5. Validator commits admitted and excluded roots.
6. Post-admission randomness fixes fair order.
7. Committee releases threshold decryption shares.
8. Validator validates revealed transactions and public headers.
9. Validator installs fair/normal fences for protected accounts.
10. Fair DeFi programs select fill or clear batch on-chain.
11. FairSettlement atomically commits token, fee, inventory, and downstream state.
12. Execution report and Fair Proof Bundle publication and verification are emitted.
13. Indexers publish scorecards and disputes if needed.
```

5 Participants and Domains

5.1 Participants

Participant	Role
User / wallet	Creates intent, signs constraints, chooses execution mode

Participant	Role
App / aggregator	Helps discover routes, submits envelopes, may sponsor fair fees
Solver	Competes to improve user outcome under intent constraints
RFQ / PropAMM venue	Provides binding or conditional fill offers
AMM / DEX program	Supplies on-chain liquidity or route leg
Fair gateway	Accepts envelopes, enforces fallback policy, routes to validators
Witness PoP	Signs ingress receipts and time/slot buckets
Fair validator	Admits, orders, reveals, fences, schedules, executes, proves
Committee	Provides threshold randomness and threshold decryption
Indexer	Verifies Fair Proof Bundles and publishes execution-quality data
Registry	Maintains certification and domain manifests
Dispute resolver	Processes reason-coded claims and bond/insurance disputes

5.2 Fairness domains

The system supports explicit domains:

- `FAIR_SWAP`: retail and app-sponsored swaps.
- `FAIR_RFQ`: binding RFQ or reserved inventory trades.
- `FAIR_PROPAMM`: PropAMM offers with state/oracle bounds.
- `FAIR_AUCTION`: multi-intent batch clearing.
- `FAIR_LIQUIDATION`: liquidation-sensitive flows.
- `FAIR_ORACLE`: oracle and risk-state updates.
- `FAIR_BRIDGE`: bridge mint/burn events that affect tradeable state.
- `FAIR_GOVERNANCE`: governance and control-plane actions.
- `NORMAL`: non-fair traffic.

Each domain has a manifest:

```
FairDomainManifest {
    domain_id,
    admission_policy_hash,
    account_fence_policy,
    encryption_policy,
    quote_policy,
    settlement_program_ids,
    fair_proof_bundle_requirements,
    capacity_split,
    forced_inclusion_policy,
    domain_interaction_edges,
    fee_policy,
    version,
    registry_signature
}
```

5.3 Domain interaction graph

Fair DeFi must handle cross-domain MEV. A swap may depend on an oracle. A liquidation may depend on a swap or price update. A governance instruction may alter a pool. A bridge mint may

affect liquidity. The validator therefore maintains a domain interaction graph:

```
G_D = (Domains, Edges)
```

An edge `A -> B` means state changes in domain `A` can materially affect domain `B`. If two domains interact through protected accounts or declared dependency edges, the validator applies one of three policies:

Policy	Meaning
`COMPOUND_BATCH`	Domains share admission, ordering, and fence boundary
`PRECEDENCE_EDGE`	Domain A must complete before Domain B
`ISOLATED`	No material cross-domain conflict; execute independently

Default examples:

Interaction	Default policy
Oracle affects swap pool	`FAIR_ORACLE` before affected `FAIR_SWAP`
Swap affects liquidation health	Compound or account-set precedence
Liquidation conflicts with oracle	`FAIR_ORACLE` before `FAIR_LIQUIDATION`
Governance changes pool control state	Governance fence before affected domain
Bridge mint/burn affects asset pool	Compound if same protected asset set

6 Core Data Objects

6.1 FairTxEnvelope

`FairTxEnvelope` is a pre-transaction object, not a normal Solana transaction.

```
FairTxEnvelope {
    version,
    domain_id,
    tx_commitment,
    public_header,
    encrypted_payload,
    payload_cipher_suite,
    receipt_quorum_hash?,
    intent_commitment_hash?,
    route_commitment_hash?,
    quote_commitment_root?,
    fair_fee_ticket?,
    reveal_bond?,
    fallback_policy,
    user_envelope_signature,
    app_or_aggregator_signature?
}
```

The public header contains fields needed before reveal:

```
FairPublicHeader {
    domain_id,
    fee_payer,
    account_metas,
    writable_accounts_root,
```

```
    readonly_accounts_root,
    protected_account_group_ids?,
    declared_compute_unit_limit,
    declared_compute_unit_price,
    max_execution_slot,
    recent_blockhash_expiry_slot?,
    intent_id,
    intent_nullifier,
    replacement_sequence,
    fair_policy_hash,
    fail_closed,
    payload_size_bytes
}
```

The encrypted payload contains signed transactions, intent openings, offer openings, and private route data:

```
EncryptedFairPayload {
    serialized_signed_solana_transaction,
    fair_intent_opening?,
    solver_offer_opening?,
    route_commitment_opening?,
    quote_commitment_openings?,
    auction_metadata_opening?,
    report_commitment_openings?
}
```

After reveal, the validator verifies that the decrypted transaction matches the public header. Mismatch produces an invalid reveal reason code and may burn the reveal bond.

6.2 FairIntent

The user intent is the core authorization object.

```
FairIntent {
    intent_id,
    user,
    input_mint,
    output_mint,
    amount_mode,              // exact_in, exact_out, range
    amount_in_max?,
    amount_in_exact?,
    amount_out_min,
    allowed_venues_root,
    blocked_venues_root?,
    allowed_programs_root,
    max_fee_bps,
    max_slippage_bps?,        // only for best-effort mode
    quote_mode,               // firm, reserved, conditional, best_effort
    settlement_mode,          // single_fill, split_fill, batch_auction
    composability_plan_hash?,
    state_bounds_root?,
    oracle_bounds_root?,
    expiry_slot,
    fair_domain,
    fail_closed,
    no_private_bundle,
    solver_policy_hash?,
```

```
    surplus_split_policy,
    refund_policy,
    nullifier,
    user_signature
}
```

The intent says:

```
I authorize a trade only if the on-chain state machine can settle it
under these constraints before this expiry.
```

The user does not authorize hidden off-chain route substitution.

6.3 FairOffer

A solver, market maker, RFQ venue, PropAMM, or AMM adapter submits an offer.

```
FairOffer {
    offer_id,
    intent_id_or_batch_id,
    maker_or_solver_id,
    offer_type,               // direct, rfq, propamm, amm_route, solver_route
    amount_in_required?,
    amount_out_promised,
    fee_bps,
    rebate_bps?,
    route_plan_hash?,
    inventory_commitment?,
    state_bound_hash?,
    oracle_bound_hash?,
    valid_from_slot,
    expires_at_slot,
    quote_class,
    bond_id?,
    settlement_accounts_root,
    offer_signature
}
```

An offer is valid only if:

```
intent constraints hold
offer signature is valid
quote class is allowed
state/oracle bounds hold
settlement accounts match declared roots
maker bond is sufficient when required
expiry has not passed
```

6.4 FairAuctionBatch

A batch auction contains multiple intents and offers.

```
FairAuctionBatch {
    batch_id,
    domain_id,
    intent_root,
    offer_root,
    clearing_rule_id,
    solver_policy_hash,
```

```
    state_snapshot_root,
    account_fence_root,
    expiry_slot,
    batch_signature_or_validator_commitment
}
```

The clearing rule is deterministic. It may implement best-output selection, uniform clearing, pro-rata allocation, solver-surplus maximization, or another registered rule.

6.5 FairSettlement

The settlement object represents the on-chain execution plan.

```
FairSettlement {
    settlement_id,
    intent_ids_root,
    winning_offer_ids_root,
    clearing_result_hash,
    token_transfer_plan_hash,
    downstream_cpi_plan_hash?,
    fee_plan_hash,
    surplus_split_hash,
    refund_plan_hash?,
    execution_report_hash,
    settlement_program_signature?
}
```

`FairSettlement` is not a promise to settle. It is the deterministic outcome that the on-chain program either commits or rejects.

6.6 FairExecutionReport

Every fair DeFi settlement emits an execution report.

```
FairExecutionReport {
    tx_signature,
    batch_id?,
    intent_ids,
    fair_lane,
    settlement_mode,
    aggregator_id?,
    solver_ids?,
    venue_ids,
    route_hash?,
    quote_ids?,
    promised_out,
    actual_out,
    user_min_out,
    user_surplus,
    solver_surplus,
    fair_pool_surplus,
    slot_intent_created,
    slot_executed,
    success,
    failure_reason?,
    report_signature
}
```

7 Native Validator-Client Architecture

7.1 Pipeline

The validator client adds a pre-banking fair path next to the ordinary transaction path:

```
Network QUIC/RPC ingress
  -> Packet demux
      -> NormalTxPath
      -> FairEnvelopePath

FairEnvelopePath
  -> Envelope precheck
  -> Receipt/quorum verification
  -> Fair buffer by domain
  -> Deterministic admission
  -> AdmissionCommit publication
  -> Threshold randomness request/reveal
  -> Fair ordering
  -> OrderCommit publication
  -> Threshold decryption
  -> Post-reveal validation
  -> Fair/normal account fence
  -> Dependency-graph fair banking
  -> Execution reports
  -> Fair Proof Bundle generation and checkpoint
  -> PoH / broadcast as ordinary ledger entries
```

The normal path remains available, but normal transactions touching protected fair-domain accounts cannot bypass the fair batch boundary.

7.2 Required modules

The validator client must implement:

1. fair ingress endpoint and packet demux;
2. envelope parser and canonicalization checker;
3. fair buffer indexed by domain, expiry, nullifier, account set, receipt quorum, and fee ticket;
4. deterministic admission engine;
5. admission-commit publisher;
6. committee registry client;
7. threshold randomness aggregator;
8. order-root builder;
9. threshold decryption aggregator;
10. post-reveal transaction validator;
11. fair/normal fence manager;
12. dependency-graph scheduler;
13. fair atomic sequence executor;
14. Fair Proof Bundle builder;
15. checkpoint publisher;
16. fair status RPC and subscriptions;
17. metrics and evidence exporters.

7.3 Public scheduling header and privacy model

The deployable Solana version is account-visible and payload-private.

Field class	Visibility before order	Reason
Fee payer	Public	Fee and anti-spam checks.
Account metas	Public	Account locks and conflict scheduling.
Writable account set	Public	Fair/normal fence and local fee estimation.
Compute budget	Public	Capacity admission and scheduling.
Expiry	Public	Liveness and blockhash validity.
Route details	Encrypted or committed	Economically sensitive.
Quote details	Encrypted or committed	Economically sensitive.
Instruction data	Encrypted	Hides side, amount, and intent details where account metas do not reveal them.
Inner transaction signatures	Encrypted until reveal	Verified post-reveal.

The protocol does not claim full route privacy. DeFi account metas can reveal venues, pools, vaults, or token pairs. The privacy claim is limited to payload privacy and content non-interference before ordering, except where content is inferable from public headers.

7.4 Fair batch lifecycle

```
BATCH_OPEN
ADMISSION_LOCKED
RANDOMNESS_REVEALED
ORDER_COMMITTED
REVEALING
VALIDATED
FENCED
EXECUTING
CHECKPOINTED
CLOSED
```

The client exposes the state machine through RPC and Fair Proof Bundles.

8 Validator-Side Fair Sequencing Protocol

8.1 Two-phase validation

Pre-reveal validation checks envelope-level conditions:

```
envelope signature
receipt quorum
policy hash
fee ticket
reveal bond
expiry header
duplicate nullifier
replacement sequence
public header syntax
account and compute bounds
payload commitment format
```

Post-reveal validation checks the inner signed transaction:

```
inner Solana transaction signatures
recent blockhash or durable nonce validity
fee payer validity
account metas matching public header
compute budget within declared bounds
program/domain policy validity
quote and route reference commitments
payload hash matching tx_commitment
```

Invalid reveals are not executed. They remain in the proof trail and are treated as failed predecessors for dependency scheduling.

8.2 Deterministic admission

Admission is deterministic and auditable. Fees may buy capacity admission under a public policy; they may not buy relative order inside an admitted batch.

```
for envelope in fair_buffer:
    if expired(envelope): exclude(EXPIRED)
    elif invalid_envelope_signature(envelope): exclude(INVALID_ENVELOPE_SIGNATURE)
    elif invalid_policy(envelope): exclude(POLICY_MISMATCH)
    elif invalid_receipt_quorum(envelope): exclude(RECEIPT_QUORUM_INVALID)
    elif invalid_fee_ticket(envelope): exclude(ADMISSION_FEE_INVALID)
    elif invalid_or_duplicate_nullifier(envelope): exclude(NULLIFIER_DUPLICATE)
    elif invalid_reveal_bond(envelope): exclude(REVEAL_BOND_INVALID)
    elif invalid_public_header(envelope): exclude(HEADER_INVALID)
    else: eligible.append(envelope)

admitted = valid_forced_inclusion_entries(forced_queue)
remaining_capacity = capacity - cost(admitted)
ranked = sort_by_hash(eligible - admitted,
                      H("admission" || admission_seed || tx_commitment))
for tx in ranked:
    if cost(tx) <= remaining_capacity:
        admit(tx)
    else:
        exclude(CAPACITY)
```

The leader then publishes:

```
FairAdmissionCommit {
    slot,
    leader,
    domain_id,
    batch_id,
    policy_hash,
    admitted_root,
    excluded_root,
    exclusion_reasons_root,
    receipt_quorum_root,
    public_headers_root,
    nullifier_root,
    forced_inclusion_root,
    cutoff_slot,
    cutoff_time_bucket_ms,
    leader_signature
}
```

8.3 Post-admission randomness and ordering

The ordering seed must not be known before the leader commits admission. The safe construction is:

```
fair_randomness = ThresholdVRFReveal(
    epoch, slot, domain_id, batch_id, admission_commit_hash)

ordering_seed = H("fair-order-v1" || fair_randomness ||
                  slot || domain_id || batch_id || admission_commit_hash)

order_key(tx) = H("order" || ordering_seed || tx.tx_commitment)
ordered_list = sort(admitted_set, by=(order_key, tx_commitment))
ordered_root = MerkleRoot([tx.tx_commitment for tx in ordered_list])
```

The leader publishes:

```
FairOrderCommit {
    batch_id,
    admission_commit_hash,
    randomness_proof_root,
    ordering_seed_hash,
    ordered_root,
    leader_signature
}
```

A replaying validator recomputes the ordered root from the Fair Proof Bundle. Any mismatch is a fair-domain validity failure.

9 Committee Protocol

Fair Outcomes uses committees for threshold randomness and threshold decryption. The committee protocol is part of the validator-client specification, not a vague off-chain service.

9.1 Committee selection

Each epoch has independent randomness and decryption committees per domain. Committees are selected deterministically from eligible certified nodes.

```
CommitteeSelectionInput {
    epoch,
    domain_id,
    previous_epoch_bank_hash,
    certified_validator_set_root,
    certified_independent_node_set_root,
    committee_policy_hash
}
committee = WeightedVRFSample(input, eligible_nodes,
                              target_size=N,
                              max_weight_per_operator,
                              region_caps)
```

Default production parameters:

```
N_randomness = 31
T_randomness = 21
N_decryption = 31
T_decryption = 21
max_operator_weight = 10% of committee seats
```

```
min_regions = 4
key_rotation_period = 1 epoch
activation_delay = 1 epoch after DKG completion
```

A node is eligible only if it has registered keys, endpoints, bonds, version information, operator identity, and audit signatures in the committee registry.

9.2 DKG and key rotation

Committee members run publicly verifiable DKG for the next epoch. The transcript is checkpointed before activation.

```
FairCommitteeTranscript {
    epoch,
    domain_id,
    committee_type,
    committee_members_root,
    public_key,
    encrypted_share_commitments_root,
    complaints_root,
    qualified_set_root,
    threshold,
    transcript_hash,
    activation_epoch,
    registry_signature
}
```

A committee key is valid only if the qualified set has at least threshold members, all complaints are resolved, the transcript is checkpointed, members have posted bonds, and the key is not reused beyond its rotation period. If DKG fails for epoch `e+1`, a predeclared backup committee activates for one epoch. Two consecutive DKG failures disable certification for the affected domain until a new key activates.

9.3 Randomness release

Randomness shares are released only after a valid `FairAdmissionCommit` exists.

```
RandomnessShare {
    epoch,
    slot,
    domain_id,
    batch_id,
    admission_commit_hash,
    member_id,
    share,
    member_signature
}
```

An aggregate randomness proof is valid only if it includes at least `T_randomness` valid shares over the exact batch tuple. A member that signs shares for conflicting admission commits for the same batch is slashable for equivocation.

9.4 Threshold decryption and release gate

The encrypted payload uses hybrid encryption under the active per-domain committee key. Decryption shares are released only after order is committed.

```
release_decryption_share iff
    valid(FairAdmissionCommit) AND
```

```
    valid(RandomnessProof) AND
    valid(FairOrderCommit) AND
    FairOrderCommit.ordered_root == recompute_ordered_root(admitted_set, ordering_seed)
```

A premature decryption share is slashable. Withholding is slashable if a member fails to release a share by deadline while online and without an accepted outage proof.

The batch reveal object is:

```
FairBatchReveal {
    batch_id,
    ordered_root,
    reveal_root,
    decryption_share_root,
    aggregate_decryption_proof,
    failed_decryption_root,
    reveal_slot,
    validator_signature
}
```

9.5 Committee failure rules

Failure	Deterministic rule
Fewer than `T_randomness` shares by deadline	Batch becomes `COMMITTEE_UNAVAILABLE_RANDOMNESS`; admitted envelopes receive forced-inclusion priority if still executable.
Fewer than `T_decryption` shares by deadline	Batch becomes `COMMITTEE_UNAVAILABLE_DECRYPTION`; envelopes enter forced inclusion or expiry evidence.
Invalid shares	Slash invalid-share members and exclude affected shares.
Equivocation	Slash equivocating members; invalidate affected batch unless a unique valid aggregate proof exists.
Repeated committee failure	Disable certification for the domain until a fresh committee key activates.

The validator must not silently reroute fair transactions to the normal lane after committee failure. Wallet fallback policy controls labeled retry behavior.

9.6 Committee failure ladder

Committee failure is not a binary condition. The validator client uses a deterministic ladder so every node reaches the same state under slow shares, missing shares, equivocation, invalid shares, or DKG failure.

```
T0_NORMAL_DEADLINE:
    release shares by domain_manifest.normal_reveal_deadline_ticks.

T1_GRACE_PERIOD:
    accept late but valid shares until grace_deadline_ticks.
    record LateShareEvidence for each late member.

T2_FAILURE_EVIDENCE:
    if threshold is still unavailable, emit CommitteeFailureEvidence.
```

```
    classify RANDOMNESS_UNAVAILABLE, DECRYPTION_UNAVAILABLE, INVALID_SHARE_SET,
    EQUIVOCATION, or DKG_UNQUALIFIED.

T3_BATCH_ABORT_OR_FORCE:
    if payloads were not revealed, abort the batch and move executable envelopes to
    forced inclusion with priority.
    if some payloads were revealed but ordering remains valid, execute only under
    the domain's partial-reveal policy; otherwise abort.

T4_DOMAIN_CERTIFICATION_PAUSE:
    if failures exceed threshold in a rolling window, pause FairSequenced certification
    for the affected domain.

T5_BACKUP_COMMITTEE_ACTIVATION:
    activate the predeclared backup committee for one epoch if its DKG transcript is valid.

T6_EMERGENCY_GOVERNANCE_MODE:
    after repeated DKG or reveal failures, governance may disable the domain or force
    a new committee epoch. Emergency mode cannot retroactively change the order or
    contents of an already committed batch.
```

The failure ladder preserves safety over liveness. A stalled fair batch may abort, force, or expire; it must not silently leak payloads, reorder transactions, or reroute fair-domain traffic to normal or private paths.

9.7 Default committee timing parameters

Initial production defaults for testnet and guarded mainnet rollout:

```
normal_randomness_deadline_ticks = 24
randomness_grace_ticks = 12
normal_decryption_deadline_ticks = 32
decryption_grace_ticks = 16
committee_failure_window_slots = 512
max_failures_before_domain_pause = 3
backup_committee_activation_delay_slots = 64
minimum_online_share_attestation_quorum = T
```

These defaults should be empirically tuned, but the deterministic ladder is mandatory.

9.8 Slashable events

```
PREMATURE_RANDOMNESS_SHARE
PREMATURE_DECRYPTION_SHARE
CONFLICTING_RANDOMNESS_SHARE
CONFLICTING_DECRYPTION_SHARE
INVALID_SHARE
WITHHOLDING_AFTER_DEADLINE
DKG_EQUIVOCATION
KEY_REUSE_AFTER_EXPIRY
FALSE_OUTAGE_PROOF
```

Penalties are governance parameters, but premature reveal and equivocation penalties must exceed the expected profit from revealing or biasing a high-value batch.

10 Fair/Normal Account Fence

The fair/normal fence is mandatory for certified DeFi domains. Without it, a leader could order fair transactions fairly relative to each other while inserting normal or private transactions before, between, or after them on the same pool, vault, oracle, or venue accounts.

10.1 Fence scope

A fence protects:

1. every writable account declared by admitted fair envelopes;
2. domain-defined account groups such as AMM pool state, token vaults, oracle accounts, venue inventory accounts, and PropAMM quote-state accounts;
3. accounts added by validated inner transactions that were predeclared in the public header; and
4. optional read-sensitive accounts if the domain marks them as state-observable inputs.

10.2 Fence interval

```
fence_start = after FairOrderCommit and before any fair tx in the batch executes
fence_end = after every valid fair tx completes or fails and required reports commit
```

The validator publishes:

```
FairBatchFence {
    batch_id,
    slot,
    domain_id,
    protected_writable_accounts_root,
    protected_read_sensitive_accounts_root?,
    protected_account_group_root,
    fence_start_tick,
    fence_end_condition,
    normal_conflict_policy,
    oracle_exception_policy?,
    liquidation_exception_policy?,
    leader_signature
}
```

10.3 Normal-conflict policy

Every fair domain chooses one policy in its domain manifest.

Policy	Rule
`DELAY_UNTIL_FENCE_RELEASE`	Queue conflicting normal transactions until the fence releases if blockhash remains valid.
`REJECT_FOR_SLOT`	Reject conflicting normal transactions for this slot with `FAIR_FENCE_CONFLICT`.
`CONVERT_IF_ELIGIBLE`	Accept the transaction only if it was submitted as a fair envelope before cutoff and satisfies the domain policy.

`ALLOW` is not a valid policy for protected accounts in a certified fair DeFi domain.

```
for fence in active_fair_fences:
    if !conflicts(normal_tx, fence.protected_accounts): continue
```

```
    if qualifies_for_domain_exception(normal_tx, fence):
        return apply_exception_policy(normal_tx, fence)
    if fence.policy == DELAY_UNTIL_FENCE_RELEASE:
        record FenceDelayProof(normal_tx, fence.batch_id)
        return delay_until(fence.release_condition)
    if fence.policy == REJECT_FOR_SLOT:
        record FenceRejectProof(normal_tx, fence.batch_id)
        return reject(FAIR_FENCE_CONFLICT)
    if fence.policy == CONVERT_IF_ELIGIBLE:
        if submitted_as_valid_fair_envelope_before_cutoff(normal_tx):
            return convert_required
        return reject(FAIR_ENVELOPE_REQUIRED)
return allow
```

10.4 Oracle, risk, and liquidation exceptions

Some domains need fresh oracle or risk-state updates. Exceptions are allowed only through predeclared exception manifests:

```
FairFenceExceptionPolicy {
    domain_id,
    allowed_program_ids,
    allowed_account_groups,
    max_compute_units,
    no_user_trade_instructions,
    no_token_balance_delta_except_oracle_accounts,
    must_execute_before_batch? | must_execute_after_batch?,
    report_required
}
```

An oracle exception cannot include user trade instructions or unrelated token-balance deltas. A liquidation exception must use `CONVERT_IF_ELIGIBLE` unless the domain defines a separate fair liquidation auction.

10.5 Fence abuse controls

A malicious user must not be able to freeze hot accounts cheaply. The domain manifest defines:

```
max_protected_accounts_per_envelope
max_fenced_accounts_per_batch
max_fence_duration_ticks
min_reveal_bond_per_protected_account
per_account_fair_capacity
fence_griefing_penalty
```

If a batch exceeds account or duration limits, deterministic admission excludes lower-ranked envelopes with `FENCE_CAPACITY` before the fence is created.

10.6 Launch parameters for fences and griefing controls

The fence is the mechanism that converts internally fair ordering into outcome-fair execution. It is also the mechanism most vulnerable to griefing. Therefore every fair domain must publish launch parameters rather than leaving them to validator discretion.

Initial guarded-mainnet defaults:

```
max_fence_duration_ticks = 96
max_protected_accounts_per_envelope = 48
max_fenced_accounts_per_batch = 512
```

```
max_candidate_offers_per_intent = 16
max_split_fills_per_intent = 4
min_reveal_bond_per_hot_account = domain_hotness_fee * 2
invalid_reveal_hot_account_penalty = min(max(reveal_bond, hotness_fee * 4), bond_cap)
max_consecutive_fenced_slots_per_account_group = 3
fence_budget_decay_half_life_slots = 64
oracle_exception_max_compute_units = 80_000
liquidation_exception_policy = CONVERT_IF_ELIGIBLE unless domain declares auction exception
```

A batch that would exceed these values is not partially fenced by validator choice. Deterministic admission excludes lower-ranked envelopes with `FENCE_CAPACITY` before the fence is installed. The exclusion is included in the Fair Proof Bundle and can be audited.

10.7 Hot-account anti-freeze rule

If the same account group is fenced repeatedly while producing invalid reveals, failed settlement, or low valid execution ratio, the domain manifest applies an anti-freeze multiplier:

```
fence_bond_multiplier(account_group) =
    1 + recent_invalid_reveals + recent_aborted_batches + recent_unsettled_fences
```

The multiplier increases reveal bonds and can temporarily move the account group to `FAIR_CAPACITY_PROTECTED` where only retail quota, forced inclusion, oracle exceptions, and certified app-sponsored flow may fence it. This prevents low-cost freezing of popular pools, vaults, oracle accounts, or liquidation state.

11 Dependency-Graph Fair Scheduler

A simple active-lock scheduler is insufficient. Example: `T1` locks `A`, `T2` locks `A,B`, and `T3` locks `B`. If `T1` starts, `T2` blocks on `A`, and `T3` appears compatible with active locks, then active-lock scheduling can start `T3` before `T2` even though `T2` conflicts with `T3` and was earlier in committed order. That violates fair ordering.

The correct scheduler builds dependency edges from committed order:

```
edge(tx_i -> tx_j) iff
    order(tx_i) < order(tx_j)
AND conflict(tx_i, tx_j)
```

A transaction is runnable only when all predecessors complete or fail and its locks are compatible with active locks.

```
for each tx_i in ordered_transactions:
    deps[tx_i] = empty_set
for each tx_i in ordered_transactions:
    for each tx_j after tx_i:
        if conflict(tx_i, tx_j): deps[tx_j].add(tx_i)

ready = [tx where deps[tx] is empty]
active_locks = empty
completed_or_failed = empty_set

while ready not empty or running not empty:
    for tx in ready in committed_order:
        if locks(tx) compatible with active_locks:
            start_parallel(tx)
            active_locks.add(locks(tx))
            ready.remove(tx)
            record start_event(tx)
```

```
    wait for at least one running tx to finish_or_fail
    for finished_tx in newly_finished:
        active_locks.remove(locks(finished_tx))
        completed_or_failed.add(finished_tx)
        record finish_event(finished_tx)
    for tx in ordered_transactions not started:
        if deps[tx] subset completed_or_failed:
            ready.add(tx)
```

Independent transactions still execute in parallel. Conflicting transactions cannot jump over earlier conflicting predecessors.

12 Validator-Produced Fair Proof Bundles, Checkpoints, and SolanaCDN Data Availability

Prior drafts described this evidence payload as a proof sidecar. That terminology is intentionally removed here because, in the Solana ecosystem, a sidecar often means an auxiliary process running next to a validator. Fair Outcomes does not require a separate validator-adjacent program to be trusted as a source of truth. The source of truth is the validator client and replay verifier.

The protocol object is a **Fair Proof Bundle**:

`FairProofBundle = validator-authored evidence for one fair-domain batch or sequence.`

The lifecycle is:

1. The validator client constructs the FairProofBundle during fair-domain execution.
2. The validator commits compact roots and the bundle manifest hash in the ledger checkpoint.
3. The validator publishes the bundle to SolanaCDN and certified validator peers.
4. SolanaCDN erasure-codes, replicates, indexes, and serves the bundle.
5. Replay validators verify bundle reconstructability before voting or finality for the fair-domain segment.
6. Auditors, wallets, indexers, registries, and dispute resolvers verify the bundle against checkpoint roots.

This is therefore a validator-authored proof object with SolanaCDN data availability, not an external process that makes fairness claims on behalf of the validator.

Checkpoint roots alone are insufficient. Auditors need the Fair Proof Bundle content that reconstructs admission, exclusion, randomness, order, reveal, validation, fence, schedule, settlement, execution-report, and forced-inclusion proofs. A validator cannot claim fair sequencing while withholding the evidence needed to verify the claim.

```
FairProofCheckpoint {
    slot,
    leader,
    bank_hash,
    domain_roots,
    batch_roots,
    admission_root,
    exclusion_root,
    ordering_root,
    randomness_root,
    reveal_root,
    validation_root,
    fence_root,
    schedule_root,
    execution_report_root,
    fair_proof_bundle_manifest_hash,
```

```
    validator_signature
}
```

The Fair Proof Bundle manifest is:

```
FairProofBundleManifest {
    slot,
    leader,
    batch_ids,
    encoding,
    erasure_coding_scheme,
    bundle_content_hash,
    chunk_roots,
    required_chunks,
    solana_cdn_storage_attestations_root,
    peer_validator_storage_attestations_root?,
    availability_window_epochs,
    validator_signature
}
```

The Fair Proof Bundle must include:

```
admission objects;
exclusion objects and reason codes;
receipt quorum objects;
public scheduling headers;
randomness shares and aggregate randomness proof;
order proof;
decryption shares and aggregate reveal proof;
failed reveal records;
post-reveal validation objects;
fair/normal fence objects;
dependency schedule trace;
normal conflict delay/reject/convert proofs;
FairDeFi settlement objects;
execution reports;
forced-inclusion and expiry evidence;
SolanaCDN availability attestations.
```

The data-availability rule is:

```
A FairSequenced or FairDeFiSettled batch is valid only if:
  1. its Fair Proof Bundle manifest hash is included in the checkpoint;
  2. the manifest is retrievable from SolanaCDN or certified validator peers;
  3. enough erasure-coded chunks are available to reconstruct the bundle;
  4. the reconstructed bundle hash matches the checkpoint commitment;
  5. the bundle remains available for the dispute window; and
  6. missing bundle data makes the fair-domain segment unverifiable.
```

Default production parameters:

```
k_solana_cdn_storage_providers = 5
availability_window_epochs = 64
proof_bundle_erasure_code = ReedSolomon(n=16, k=10) or equivalent
max_proof_bundle_publish_delay_slots = 2
```

SolanaCDN is the primary availability and retrieval layer for Fair Proof Bundles. Validator peers may also cache and gossip bundles, but SolanaCDN provides the durable retrieval interface used by wallets, indexers, registries, and dispute systems.

13 Fair Proof Bundle Vote and Finality Rules

Fair Proof Bundles are not optional audit logs. In protocol-enforced fair domains, they are validator-produced data-availability objects committed by the ledger and distributed by SolanaCDN. A checkpoint root without a reconstructable Fair Proof Bundle is insufficient because replay validators, auditors, wallets, and dispute resolvers must reconstruct admission, exclusion, randomness, reveal, validation, fence, schedule, settlement, and execution-report proofs.

13.1 Vote eligibility rule

A validator must not vote on a fair-domain segment unless the Fair Proof Bundle manifest and the minimum reconstructable chunk set are available through SolanaCDN or certified validator-peer bundle gossip before the voting deadline.

```
VoteEligibleFairSegment(batch) iff
    FairProofCheckpointPresent(batch)
    AND FairProofBundleManifestPresent(batch)
    AND FairProofBundleRootMatchesCheckpoint(batch)
    AND FairProofBundleReconstructableBeforeVote(batch)
    AND SolanaCDNAvailabilityAttested(batch)
    AND PublishDelay <= max_proof_bundle_publish_delay_slots
```

If this predicate fails before the vote deadline, the segment cannot claim `FairSequenced` or `FairDeFiSettled` status. Depending on the domain manifest, the deterministic consequence is fair-domain replay rejection, fair-domain abort, or conversion to ordinary non-fair replay without the fair label. A validator must not both withhold the Fair Proof Bundle and claim the fair reward.

13.2 Deterministic missing-bundle consequences

Fair Proof Bundle failures have deterministic outcomes:

Failure	Consequence
Manifest missing from checkpoint	fair-domain replay rejection
Manifest present but bundle unreconstructable before vote	fair-domain segment not finalizable as fair
Bundle hash mismatch	fair-domain replay rejection
Bundle unavailable during dispute window	reward clawback and certification violation
Bundle delayed beyond `max_proof_bundle_publish_delay_slots`	batch loses fair validity unless the domain fallback explicitly permits abort

13.3 SolanaCDN propagation parameters

```
k_solana_cdn_storage_providers = 5
k_peer_validator_caches = 2
availability_window_epochs = 64
proof_bundle_erasure_code = ReedSolomon(n=24, k=12) or equivalent
max_proof_bundle_publish_delay_slots = 2
vote_deadline = domain_manifest_defined
bundle_chunk_max_bytes = domain_manifest_defined
bundle_gossip_retry_window_slots = 4
```

The validator client constructs the bundle. SolanaCDN stores, replicates, and serves it. The ledger checkpoint commits to it. Replay validators verify it. These roles must not be collapsed into a vague external evidence service.

13.4 Fair Proof Bundle finality invariant

```
FairSegmentFinalizable(batch) => FairProofBundleReconstructableBeforeVote(batch)
```

14 Forced Inclusion and Expiry

Forced inclusion is a deterministic queue for eligible witnessed envelopes that were omitted or excluded without an accepted reason.

```
ForcedInclusionEntry {
    tx_commitment,
    intent_nullifier,
    receipt_quorum_hash,
    first_seen_slot,
    omitted_by_leader,
    eligible_after_slot,
    expires_at_slot,
    domain_id,
    executable_mode,
    status,
    evidence_root
}
```

Forced inclusion has three executable modes.

Mode	Condition	Semantics
`SIGNED_TX_RECENT_BLOCKHASH`	Inner transaction blockhash still valid.	Next fair leader must admit it or prove invalidity.
`SIGNED_TX_DURABLE_NONCE`	Durable nonce remains valid and unadvanced.	Next fair leader must admit it unless nonce state proves invalidity.
`FAIR_INTENT_RESIGN`	Wallet/app supplied a re-signable intent template.	Regenerate fresh signed transaction under same nullifier and no new ordering ticket.

If no executable mode remains, the output is `CENSORSHIP_EVIDENCE_ONLY`. This proves omission but does not claim an expired transaction or stale quote can execute.

15 Anti-Grinding and Spam Resistance

Hash-based admission and ordering are vulnerable if an attacker can submit many commitment variants. The fair client therefore implements these rules as admission validity conditions:

- one active envelope per `intent_nullifier` before cutoff;
- public replacement sequence for legitimate updates;
- fair admission fee or app subsidy per nullifier;
- reveal bond burned or slashed for invalid reveals;
- per-fee-payer and per-account rate limits under congestion;
- quote ID and route ID binding for DeFi flows;
- canonical payload construction in wallet SDKs;
- optional institution identity or stake-based quotas for high-volume flow.

The nullifier must be derived from the economic intent rather than transaction bytes. Otherwise a user could grind by changing blockhashes, nonces, fee payers, wrappers, or harmless instruction data.

The proof system uses sorted or sparse Merkle trees keyed by `tx_commitment` and `intent_nullifier`, allowing auditors to prove both membership and absence. Plain unsorted Merkle trees are not sufficient for non-membership proofs.

16 On-Chain Fair DeFi Execution Layer

16.1 Program suite

Fair DeFi is implemented as a set of on-chain programs.

Program	Responsibility
`FairIntent`	Creates, validates, cancels, and consumes user intents
`FairOffer`	Registers maker, solver, RFQ, PropAMM, or AMM offers
`FairAuction`	Clears multi-intent batches under deterministic rules
`FairSettlement`	Executes transfers, fees, rebates, refunds, and downstream CPIs
`FairReport`	Emits canonical execution reports
`FairBond`	Manages venue, solver, reveal, and anti-spam bonds
`FairInsurance`	Pays user refunds for infrastructure or venue failures

The programs may be separate or combined into a single program suite. The separation is conceptual.

16.2 Single-intent settlement

The simplest mode is single-intent best-fill settlement.

```
Input:
    one FairIntent
    one or more FairOffers

Selection:
    choose the valid offer with highest user surplus
    tie-break deterministically

Settlement:
    debit user input
    credit maker or venue input
    debit maker or venue output
    credit user output
    transfer fees and rebates
    split surplus if applicable
    update venue bond or inventory accounts
    emit FairExecutionReport
```

If no valid offer satisfies the intent, the program returns a deterministic error. It does not silently degrade into a worse fill unless the user explicitly chose best-effort mode.

16.3 Split-fill settlement

A split-fill settlement combines multiple offers if the user's intent permits it.

```
ValidSplitFill iff:
    sum(input_debited) <= user amount_in_max
    sum(output_received) >= user amount_out_min
    all offers are valid
    all route/program constraints hold
    max_fee_bps is respected
    split count <= domain limit
    account and compute limits are respected
```

Split fills are useful for liquidity aggregation, but they create more account and compute pressure. The domain manifest may cap split count and require higher bonds.

16.4 Batch auction settlement

A batch auction takes many intents and many offers, then computes a clearing result.

```
ClearBatch(S, Intents, Offers, Rule) -> ClearingResult
```

A clearing result is valid only if:

```
every filled intent meets its constraints
no intent is filled twice beyond authorization
no offer is consumed beyond committed inventory
fees and rebates satisfy declared bounds
surplus split is deterministic
state/oracle bounds hold at settlement
all token movements balance
```

Batch auctions reduce the importance of within-batch ordering because the batch itself defines execution semantics. They are especially useful for retail swaps and correlated orderflow.

16.5 Uniform clearing and surplus maximization

For fungible two-sided markets, the clearing rule may compute a uniform price. For multi-asset routing, the clearing rule may optimize user surplus subject to feasibility constraints.

```
Objective:
    maximize total user surplus
Subject to:
    intent constraints
offer constraints
inventory constraints
account and compute limits
state bounds
fee bounds
```

The protocol should start with simpler deterministic rules before deploying complex optimization. A complex solver may propose a clearing solution, but the on-chain program must verify feasibility and objective constraints within compute limits.

16.6 On-chain verification of off-chain solver work

Solvers may compute routes or batch clears off-chain. The on-chain program verifies the proposed result.

```
Off-chain solver computes candidate clearing.
Solver submits CandidateClearing with bond.
```

```
On-chain program verifies constraints and objective certificate.
If valid, settlement executes.
If invalid, candidate fails and bond may be penalized.
```

This preserves on-chain finality without forcing every search computation onto the chain.

17 Account-Set Closure and Execution Bounds

The most important Solana-specific requirement for Fair DeFi is account-set closure. A Solana transaction cannot dynamically settle against arbitrary accounts that were not declared in the transaction or sequence. The validator needs public account metas for locks, fair/normal fences, dependency scheduling, and replay validity. Therefore Fair DeFi cannot select a winning offer or downstream program path unless every account that may be read, written, debited, credited, invoked, reported, refunded, or penalized is closed before fair ordering.

17.1 Account-set closure rule

```
AccountSetClosed(x) iff
    CandidateAccounts(x) subset PublicHeaderAccounts(x)
    AND SettlementAccounts(x) subset PublicHeaderAccounts(x)
    AND ReportAccounts(x) subset PublicHeaderAccounts(x)
    AND BondAndRefundAccounts(x) subset PublicHeaderAccounts(x)
    AND DownstreamCPIAccounts(x) subset PublicHeaderAccounts(x)
    AND AddressLookupTablesResolvedBeforeAdmission(x)
```

A fair-domain settlement object is replay-invalid if it selects, invokes, reads, writes, debits, credits, refunds, reports, or penalizes an account outside the closed set. This rule applies to single-intent settlement, split fills, batch auctions, liquidation auctions, and staged escrow settlement.

17.2 Candidate universe

For offer selection and batch clearing, the candidate universe must be committed before fair ordering.

```
CandidateUniverse {
    universe_id,
    domain_id,
    intent_ids_root,
    offer_ids_root,
    candidate_account_root,
    candidate_program_root,
    allowed_cpi_plan_root,
    max_candidate_offers_per_intent,
    max_split_fills,
    max_total_compute_units,
    max_loaded_accounts,
    max_loaded_account_data_bytes,
    expiry_slot,
    policy_hash,
    commitment_signature
}
```

The Fair DeFi program may choose among valid candidates inside this universe. It may not discover a new venue, vault, token account, oracle, lending account, or bond account after fair ordering. If the best economic offer is outside the universe, it is not eligible for that settlement.

17.3 Public header compatibility

The `FairPublicHeader` must include enough account and resource information to let validators schedule safely:

```
closed_account_set_root
address_lookup_table_roots
candidate_universe_hash?
max_loaded_accounts
max_loaded_account_data_bytes
max_total_compute_units
max_cpi_depth
max_cpi_account_infos
max_instruction_data_bytes
settlement_mode
auction_size_class?
```

Post-reveal validation verifies that the inner transaction or fair atomic sequence exactly matches the committed header. A mismatch is an invalid reveal and burns the reveal bond according to the domain manifest.

17.4 Execution-bound rule

```
ExecutionBounded(x) iff
    declared_compute_units <= domain.max_compute_units
    AND declared_loaded_accounts <= domain.max_loaded_accounts
    AND declared_loaded_account_data_bytes <= domain.max_loaded_account_data_bytes
    AND declared_cpi_depth <= domain.max_cpi_depth
    AND declared_cpi_account_infos <= domain.max_cpi_account_infos
    AND declared_instruction_data_bytes <= domain.max_instruction_data_bytes
```

The validator client rejects envelopes that exceed domain bounds at admission and rejects revealed payloads that exceed their public-header declaration. This converts runtime limits into pre-execution fairness constraints.

17.5 Account-closure proof object

```
AccountSetClosureProof {
    batch_id,
    tx_commitment_or_intent_id,
    public_header_account_root,
    candidate_universe_hash?,
    selected_offer_ids_root?,
    downstream_cpi_plan_root?,
    settlement_account_root,
    report_account_root,
    bond_refund_account_root,
    all_accounts_subset_check,
    address_lookup_resolution_root,
    validator_signature
}
```

Replay validators recompute the subset checks from the Fair Proof Bundle. Any selected account outside the closure root is a fair-domain replay failure.

18 Atomicity Modes

Fair Outcomes supports three atomicity modes. The mode must be declared before admission and is part of the replay predicate. This prevents the system from claiming native single-transaction atomicity for workflows that actually require multiple transactions or staged settlement.

18.1 Mode A: native single-transaction atomic settlement

```
settlement_mode = NATIVE_SINGLE_TX
```

Mode A is the MVP and the strongest claim. The entire Fair DeFi settlement fits in one Solana transaction. All instructions succeed or all state changes revert. This mode is limited by transaction size, account count, compute budget, instruction count, CPI depth, CPI account-info count, and loaded account data.

Valid examples:

```
one user intent;
one or bounded-N candidate offers;
one winning offer or bounded split fill;
all accounts closed before order;
settlement and report emitted in the same transaction.
```

18.2 Mode B: validator-enforced fair atomic sequence

```
settlement_mode = FAIR_ATOMIC_SEQUENCE
```

Mode B supports a bounded sequence of multiple Solana transactions treated as one fair-domain atomic object by the new validator client. The validator simulates the full sequence under the fair fence and includes the sequence only if all members pass. Replay rejects partial inclusion, third-party insertion, account-closure violation, or schedule mismatch.

```
FairAtomicSequenceValid(seq) iff
    all sequence members are ordered as one fair object
    AND no unrelated transaction executes between members on protected accounts
    AND every member's account set is closed
    AND pre-state and post-state commitments match
    AND either all members execute or none execute under deterministic abort rules
```

Mode B is not ordinary Solana transaction atomicity. It is fair-domain atomicity enforced by the validator client and replay rules.

18.3 Mode C: staged escrow settlement

```
settlement_mode = STAGED_ESCROW
```

Mode C is for large auctions or workflows that cannot fit in a single transaction or bounded atomic sequence. Users escrow assets or authorization under deterministic rules, solvers submit clearing certificates, and settlement occurs across multiple transactions or slots with explicit finalization and refund paths.

Mode C is not instant one-transition atomicity. Its safety property is escrowed determinism:

```
assets cannot be misdirected;
intents cannot be overfilled;
refunds are deterministic;
failed stages unlock or refund according to the manifest;
clearing certificates are verifiable;
reports identify every settled and unsettled intent.
```

18.4 Mode labels

Only Mode A may be labeled `NativeAtomic`. Mode B may be labeled `FairAtomicSequence`. Mode C may be labeled `StagedFairSettlement`. All three may be `FairSequenced` if the validator proof is valid, but only the mode-specific label describes the atomicity guarantee.

19 Batch-Auction Size Classes

Batch auctions are strategically important, but a naive attempt to clear every intent and every offer in one transaction will not scale. Fair Outcomes therefore defines size classes.

19.1 Small auction

```
auction_size_class = SMALL_NATIVE
```

A small auction fits in one native Solana transaction. All accounts are closed, all candidate offers are declared, and clearing plus settlement plus reporting occur in a single atomic transaction.

Initial bounds:

```
max_intents = 8
max_offers = 32
max_split_fills_per_intent = 4
max_downstream_programs = 3
max_total_compute_units = domain.max_compute_units
```

19.2 Medium auction

```
auction_size_class = MEDIUM_FAIR_SEQUENCE
```

A medium auction uses a validator-enforced fair atomic sequence. The sequence is ordered and fenced as one fair object. It can use multiple transactions but must satisfy partial-inclusion rejection, closed account sets, and sequence Fair Proof Bundles.

Initial bounds:

```
max_sequence_transactions = 5
max_intents = 32
max_offers = 128
max_split_fills_per_intent = 4
max_consecutive_fenced_slots = 1
```

19.3 Large auction

```
auction_size_class = LARGE_STAGED_ESCROW
```

A large auction uses staged escrow settlement. It does not claim single-transition atomicity. It claims deterministic escrow, deterministic clearing certificates, deterministic settlement attempts, and deterministic refunds.

Initial bounds:

```
max_escrow_window_slots = 150
max_clearing_certificate_bytes = domain_defined
max_settlement_batches = domain_defined
refund_deadline_slots = escrow_window + domain.refund_grace_slots
```

19.4 Auction class replay rule

```
AuctionClassValid(batch) iff
    settlement_mode matches auction_size_class
    AND account_set_closure holds for the class
    AND size limits are not exceeded
    AND Fair Proof Bundle includes class-specific proof objects
```

A large staged auction must not be advertised as native atomic. A medium fair sequence must not be advertised as ordinary Solana transaction atomicity. This label discipline is part of the user-protection model.

20 Atomic Composability With the State Machine

20.1 Composability principle

A fair trade may need to compose with other DeFi state:

```
swap -> deposit collateral -> update health factor
borrow -> swap -> repay
liquidate -> sell collateral -> repay debt -> return surplus
oracle update -> quote validation -> swap
bridge mint -> settlement -> vault accounting
```

Because Solana transactions are atomic at the instruction level and CPIs enable program-to-program invocation [5,6], Fair DeFi can make these workflows one state transition when account and compute limits permit.

20.2 ComposabilityPlan

The user intent may include a composability plan:

```
ComposabilityPlan {
    plan_id,
    allowed_programs_root,
    required_preconditions_root,
    downstream_instruction_hashes,
    downstream_account_roots,
    max_total_compute_units,
    abort_policy,
    report_policy
}
```

The settlement program checks that the downstream instructions match the signed plan. A solver cannot add arbitrary downstream behavior.

20.3 Lending and margin integration

For lending or margin flows, the intent can require:

```
post_trade_health_factor >= threshold
collateral_deposit_amount >= amount
borrowed_asset_repaid >= amount
no unauthorized debt increase
lending_program_id in allowed_programs
```

Settlement fails if the trade would leave the account below the user’s required risk threshold.

20.4 Liquidation integration

Liquidations are time-sensitive and can create MEV. Fair DeFi supports liquidation auctions:

```
LiquidationIntent {
    position_id,
    repay_asset,
    collateral_asset,
    max_repay_amount,
    min_collateral_out,
    protocol_required_discount,
    fair_liquidation_domain,
    expiry_slot
}
```

A fair liquidation batch can clear multiple liquidation candidates deterministically, with protocol-owned surplus rules and account fences around risk-state accounts.

20.5 Oracle integration

Oracle updates may be prerequisites for fair settlement. The domain graph should usually place affected oracle updates before dependent swaps or liquidations. A quote may include oracle bounds:

```
oracle_price in [min_price, max_price]
oracle_age_slots <= max_age_slots
oracle_account == declared_account
```

If the oracle bound fails, a firm quote can fail cleanly without venue penalty.

20.6 Vault and structured-product integration

Vaults and structured products often perform multiple actions in sequence. Fair DeFi can require that:

```
vault share price update occurs before deposit or withdrawal
swap settlement uses declared vault accounting state
fees are charged under declared rules
post-settlement vault invariant holds
```

The execution report records both trade output and downstream accounting output.

21 Quote Classes and Venue Commitments

21.1 Quote classes

Quote class	Meaning	Settlement behavior
`FIRM`	Venue commits to at least promised output until expiry	Fill-or-revert; violation if worse fill settles
`RESERVED`	Venue reserves inventory for short window	Fill-or-revert while reservation holds
`CONDITIONAL`	Quote valid only if state/oracle bounds hold	Fill-or-revert if conditions hold; clean fail otherwise
`INDICATIVE`	Not binding	Cannot be displayed as guaranteed execution
`BEST_EFFORT`	May degrade within user bounds	Must be explicitly labeled and scored separately

21.2 VenueQuoteCommitment

```
VenueQuoteCommitment {
    venue_id,
    quote_id,
    quote_class,
    user?,
    input_mint,
    output_mint,
    amount_in,
    promised_amount_out,
    max_fee_bps,
    valid_from_slot,
    expires_at_slot,
    state_bound_hash?,
    oracle_bound_hash?,
    quote_nonce,
    bond_id?,
    venue_signature
}
```

The FairSettlement program verifies the quote commitment before settlement. If the quote is firm and valid, actual output below the promised amount is invalid.

21.3 RouteCommitment

Aggregators may still be useful for route discovery. A route commitment must be signed:

```
RouteCommitment {
    aggregator_id,
    route_id,
    quote_set_hash,
    route_plan_hash,
    displayed_expected_out,
    displayed_min_out,
    fees_bps,
    rebates_bps,
    fair_domain,
    expires_at_slot,
    aggregator_signature
}
```

The route commitment is not enough by itself. A fair DeFi transaction must either settle through on-chain offer selection or prove that the executed route matches the signed route and user constraints.

21.4 Slippage is not a degradation license

Slippage tolerance is a safety bound. It is not permission for a firm quote venue to degrade execution.

Example:

```
Firm promised output: 18,420 units
Emergency min_out:    18,300 units
```

A settlement at 18,350 may satisfy the emergency bound, but it violates the firm quote unless a declared state/oracle condition invalidated the firm commitment. The execution report evaluates the venue against 18,420, not merely 18,300.

22 Solver Competition Without Private Extraction

22.1 SolverCommitment

Searchers and solvers are not eliminated. They are redirected into public surplus creation.

```
SolverCommitment {
    solver_id,
    target_intent_hash_or_batch_id,
    route_constraints_hash,
    offer_ids_root,
    proposed_clearing_hash,
    user_surplus_min,
    solver_fee_bps,
    fair_pool_share_bps,
    expiry_slot,
    bond_id,
    solver_signature
}
```

The solver competes to improve the user's outcome under the user's constraints. The solver does not get to insert arbitrary transactions around the user.

22.2 Surplus sharing

If settlement produces surplus above the user's guaranteed output:

```
Surplus = ActualOut - GuaranteedOut
```

Then:

```
UserShare     = lambda * Surplus
SolverShare   = sigma  * Surplus
FairPoolShare = mu     * Surplus
```

with:

```
lambda + sigma + mu = 1
```

The default should favor the user. The solver is paid for creating surplus, not for extracting from transaction position.

22.3 No third-party insertion

A FairIntent may set:

```
no_third_party_insertion = true
```

If set, settlement cannot include unrelated third-party instructions, bundle legs, or transfers. Solver participation must be delegated by the intent and bound by the settlement plan.

23 Fair/Normal Fences and On-Chain Settlement Safety

23.1 Why fences are required

Even if fair transactions are ordered correctly among themselves, normal-lane transactions can manipulate state around them. A normal transaction can update an AMM pool, oracle account, vault, or venue account before or between fair transactions. Therefore the validator must fence protected accounts.

23.2 Fence semantics

For each fair batch:

```
FenceStart = after FairAdmissionCommit and before fair settlement inputs execute
FenceEnd   = after FairSettlement completion or deterministic failure
```

During that interval, conflicting normal-lane mutations are invalid unless expressly permitted by the domain manifest.

23.3 Fence exceptions

Some state updates may be necessary:

Exception	Example	Rule
Required oracle update	Fresh price before settlement	Must be in `FAIR_ORACLE` precedence edge
Protocol maintenance	Emergency pause	Must be governance-authorized and reportable
User cancellation	Intent cancel before admission	Allowed only before batch lock
Expiry cleanup	Expired intent cleanup	Allowed after fair batch no longer uses intent

23.4 Fence griefing controls

Fences can be abused to delay hot accounts. The protocol imposes:

```
per-account fence budgets
per-domain fair capacity
intent bonds
invalid reveal penalties
failed-settlement penalties
retail quota protection
oracle/liquidation exception rules
```

An attacker should not be able to repeatedly fence a hot pool with invalid fair envelopes at low cost.

24 Forced Inclusion, Expiry, and Durable Intents

A witnessed eligible transaction that is repeatedly excluded without a valid reason enters the forced-inclusion queue.

```
ForcedInclusionEntry {
    tx_commitment,
    receipt_quorum_hash,
    first_seen_slot,
    eligible_after_slot,
    expires_at_slot,
    domain_id,
    status
}
```

Forced inclusion guarantees deterministic treatment, not guaranteed successful execution. If the underlying blockhash, nonce, intent, quote, or state-bound condition has expired, the protocol emits censorship or expiry evidence rather than pretending the transaction can still execute.

Valid outcomes:

Condition	Outcome
Still executable	Must be admitted in forced quota
Blockhash expired	Censorship/expiry evidence
Quote expired	Quote-expiry report
Intent canceled before forced eligibility	Cancellation proof
State bounds impossible	State-bound failure report

25 Committee Protocol for Fair Sequencing and Fair DeFi

25.1 Roles

The committee provides threshold randomness and threshold decryption.

```
Randomness committee: releases post-admission seed.
Decryption committee: releases payload decryption shares after order commit.
```

The committees may be the same set or separate sets. Separation reduces correlated failure risk.

25.2 DKG and key rotation

For each epoch:

```
1. Registry selects committee under stake, operator, and geography caps.
2. Committee runs DKG.
3. Public key and verification commitments are registered.
4. Members post availability and misbehavior bonds.
5. Keys rotate after epoch or emergency event.
```

25.3 Release gates

Randomness release gate:

```
FairAdmissionCommit published
admitted_root fixed
excluded_root fixed
policy_hash fixed
```

Decryption release gate:

```
FairOrderCommit published
ordered_root fixed
randomness proof valid
```

No committee member should release decryption shares before the order root is committed.

25.4 Liveness and secrecy assumptions

Let `n` be committee size and `t` be the threshold.

Secrecy before reveal holds if fewer than `t` members collude or release shares early.

Liveness holds if at least `t` members are online and willing to release valid shares after the release gate.

Penalties apply for:

```
premature share release
invalid share
withholding after valid release gate
```

```
equivocation
failure to participate in DKG
```

26 Fair DeFi Safety Invariants

The protocol defines the following invariants.

26.1 I1: Committed conflicting order cannot be inverted

If two fair transactions conflict on account locks and `tx_i` precedes `tx_j` in committed fair order, then `tx_j` cannot execute before `tx_i` completes or fails.

26.2 I2: Normal-lane fences cannot be bypassed

A normal or private transaction cannot mutate a fenced account during the fence interval unless the domain manifest includes a valid exception.

26.3 I3: On-chain intent constraints cannot be bypassed

A settlement is invalid if it debits more than authorized, credits less than required, uses an unauthorized venue, exceeds fee bounds, violates state bounds, or omits a required report.

26.4 I4: Winning offer selection is deterministic

Given the same state, intent set, offer set, and clearing rule, all replay nodes compute the same winning offer or clearing result.

26.5 I5: Firm quote degradation cannot settle

If a firm quote is valid at settlement, actual output below promised output is invalid.

26.6 I6: Atomic composability is all-or-nothing

If any required downstream CPI or state update fails, the entire settlement fails and state changes revert according to the runtime's transaction atomicity.

26.7 I7: Fair Proof Bundle non-availability has deterministic consequences

If required Fair Proof Bundle data is unavailable after the challenge window, the fair-domain block segment is rejected, downgraded, or marked invalid according to the manifest. The consequence is deterministic.

26.8 I8: Off-chain discovery cannot alter settlement

An off-chain quote, route, solver computation, or venue model has no effect unless represented by an on-chain-verifiable commitment accepted by Fair DeFi programs.

27 Fair-Domain and Fair-DeFi Replay Rules

Under sufficient client adoption, fair-domain validation is part of replay. A block segment that claims `FairSequenced` is invalid for the domain if:

1. it lacks a matching batch commit and Fair Proof Bundle;
2. admitted, excluded, public-header, receipt-quorum, nullifier, or reason-code roots do not match the Fair Proof Bundle;
3. randomness was released before admission commit;
4. ordered root is not the deterministic sort over admitted commitments and post-admission randomness;
5. a payload is decrypted or revealed before `FairOrderCommit`;
6. a decrypted transaction fails post-reveal validation but is executed;
7. a valid reveal is omitted without accepted reason;
8. a normal/private transaction executes against protected accounts during the fence interval;
9. the schedule violates dependency-graph order for conflicting transactions;
10. a fair atomic sequence is partially included or permits third-party insertion;
11. a required execution report is missing for quote-committed DeFi flow; or
12. Fair Proof Bundle data is unavailable during the dispute window.

These rules define the boundary between an ordinary valid Solana state transition and a valid certified fair-domain transition.

27.1 Additional replay rules for on-chain atomic composable trading

A block segment or transaction that claims `FairDeFiSettled` is invalid for the fair domain if any of the following holds:

1. the user intent is missing, malformed, expired, duplicated by nullifier, or not authorized by the user;
2. the selected offer or clearing result is not derivable from the on-chain-valid input set;
3. the settlement does not satisfy the user's minimum output, max fee, allowed venue, allowed program, quote class, or expiry constraints;
4. a solver receives surplus not permitted by the published surplus-sharing rule;
5. a venue receives inventory or fees inconsistent with its on-chain offer or quote commitment;
6. downstream lending, margin, vault, oracle, liquidation, or governance instructions execute outside the declared `ComposabilityPlan`;
7. the execution report omits promised output, actual output, route hash, offer hash, quote IDs, solver ID, venue IDs, or failure reason when required;
8. an atomic sequence partially settles, permits unrelated third-party insertion, or fails to roll back all state changes when a required predicate fails; or
9. the Fair Proof Bundle claims fair settlement while the on-chain program state proves a different selected offer, clearing rule, or settlement result.

These additional rules are what change Fair Outcomes from a fair sequencing protocol into a fair DeFi state-machine protocol. The validator enforces admission, ordering, reveal, fencing, scheduling, and replay proof validity. The on-chain Fair DeFi programs enforce economic settlement validity.

28 Security Properties and Proof Sketches

28.1 Content non-interference before ordering

If payload encryption is secure, fewer than threshold decryption members collude before release, and account headers do not themselves reveal the protected content, the leader cannot choose order as a function of encrypted side, amount, route details, quote openings, or signed instruction data. The leader sees only commitments and public scheduling headers before order is fixed.

28.2 Leader-unbiasable ordering

Because the leader publishes `FairAdmissionCommit` before threshold randomness is released, it cannot alter the admitted set after learning the ordering seed. Replay validators recompute the

ordered root from the Fair Proof Bundle and reject mismatches.

28.3 Conflict-order preservation

The dependency-graph scheduler gives every later conflicting transaction an edge from every earlier conflicting transaction. Therefore a later transaction cannot execute before an earlier conflicting predecessor completes or fails, while independent transactions can still execute in parallel.

28.4 Normal-lane non-interference

The fence blocks, delays, rejects, or converts normal transactions that touch protected accounts during the fence interval. Therefore protected state can be modified during the interval only by fair-batch transactions and declared domain exceptions.

28.5 Censorship detectability

A witnessed envelope with a valid receipt quorum before cutoff that is absent from admitted and reason-coded excluded sets produces omission evidence. Sorted or sparse commitment structures support non-membership proofs.

28.6 Quote integrity

For participating firm or reserved quotes, a venue cannot settle below committed output without program-level failure or a violation report, because the program or wrapper verifies the quote commitment and fill terms.

28.7 On-chain settlement determinism

Claim. If the Fair DeFi settlement program is deterministic, and every binding intent, offer, quote, route, and composability constraint is represented in on-chain-verifiable state or signed instruction data, then two honest replay validators compute the same selected fill, settlement result, execution report, refund path, and bond consequence for the same fair-domain input set.

Sketch. The selected output is a pure function of the on-chain state, revealed fair batch, valid offers, domain manifest, and deterministic program logic. Off-chain discovery may influence which offers are submitted, but it does not influence the final settlement once the on-chain input set is fixed. Any divergence is a program determinism bug or replay validity failure.

28.8 Atomic composability correctness

Claim. A `FairDeFiSettled` transaction or fair atomic sequence cannot leave partially applied state across token, AMM, lending, liquidation, vault, oracle, or report programs.

Sketch. Single-transaction settlement uses Solana's native all-or-nothing transaction semantics. Multi-transaction fair atomic sequences are simulated and committed by the validator only if the entire sequence satisfies the fair atomic executor rules; under the adoption assumption, partial inclusion is a fair-domain replay failure. Downstream program effects are valid only if they appear in the declared `ComposabilityPlan` and satisfy all preconditions.

28.9 Off-chain discovery cannot alter settlement

Claim. An off-chain solver, market maker, aggregator, or PropAMM model cannot change the final user outcome after the on-chain intent and offer set are fixed.

Sketch. The solver may choose which commitments to submit. Once submitted, the Fair DeFi program verifies offer validity, quote bounds, expiry, inventory authorization, fee limits, and surplus-sharing rules. The selected fill is determined by on-chain program logic. Any off-chain instruction

inconsistent with the on-chain commitment fails validation or becomes attributable in the execution report.

29 Formal Specification and Bounded Model Checking

This section turns the validator-client design into a small transition-system specification and records the first bounded model-checking pass. The purpose is to make the fairness claims executable and falsifiable. The model intentionally abstracts signatures, hashes, threshold encryption, and Merkle trees as ideal primitives with binding, unforgeability, and threshold assumptions. It checks the protocol control logic: admission binding, order construction, conflict scheduling, replay rejection, fair/normal fences, forced inclusion determinism, committee reveal gates, and Fair Proof Bundle availability consequences.

The model is not a substitute for cryptographic proofs, implementation audits, or large-scale performance testing. It is a finite-state safety model over bounded accounts, transactions, committee sizes, and tamper cases. It is designed to catch protocol bugs of the kind that existed in the active-lock-only scheduler.

29.1 Abstract state

Let:

```
TxId        = finite set of transaction identifiers
Account     = finite set of account identifiers
Domain      = {FAIR_SWAP, FAIR_RFQ, FAIR_PROPAMM, FAIR_LIQUIDATION, FAIR_ORACLE,
               FAIR_ATOMIC, FAIR_GOVERNANCE, NORMAL}
Committee   = finite set of committee members
Reason      = {EXPIRED, INVALID, DUPLICATE, POLICY_MISMATCH, CAPACITY,
               FORCED_OVER_CAPACITY, DECRYPTION_FAILED, POST_REVEAL_INVALID}
```

Each fair envelope is modeled as:

```
Envelope(tx) = {
    domain,
    commitment,
    declared_locks,
    expiry_slot,
    receipt_quorum_valid,
    fee_ticket_valid,
    policy_valid,
    encrypted_payload,
    envelope_signature_valid,
    forced_entry_valid
}
```

After reveal, the inner transaction is modeled as:

```
InnerTx(tx) = {
    signatures_valid,
    blockhash_or_nonce_valid,
    fee_payer_valid,
    account_header_matches_envelope,
    compute_budget_within_bound,
    payload_commitment_matches,
    program_constraints_valid,
    quote_constraints_valid,
    locks
}
```

A fair batch is modeled as:

```
Batch = {
    slot,
    domain,
    cutoff,
    policy_hash,
    admitted_set,
    excluded_set,
    exclusion_reason,
    admission_commit_posted,
    admitted_root,
    excluded_root,
    randomness,
    randomness_revealed,
    ordered_list,
    ordered_root,
    reveal_share_count,
    reveal_root,
    failed_decryption_set,
    post_reveal_valid_set,
    post_reveal_invalid_set,
    protected_accounts,
    fair_schedule,
    normal_pre_boundary,
    fair_proof_bundle_available,
    fair_proof_bundle_root_matches
}
```

A replay node accepts a claimed fair-domain segment only if all verifier predicates hold. In protocol-enforced mode, an invalid predicate rejects fair-domain replay; it is not merely a reputation event.

29.2 Transition rules

The model uses the following transition rules.

29.2.1 SubmitEnvelope

A user, app, or gateway inserts an envelope into the fair buffer.

```
Preconditions:
    current_slot <= envelope.expiry_slot
    envelope.domain != NORMAL

Effects:
    fair_buffer' = fair_buffer union {envelope}
```

29.2.2 WitnessReceipt

A witness signs a receipt for the envelope commitment and policy hash.

```
Preconditions:
    envelope in fair_buffer
    current_slot <= envelope.expiry_slot

Effects:
    receipts'(tx) = receipts(tx) union {witness_signature}
```

29.2.3 DeterministicAdmission

The leader computes admitted and excluded sets from forced entries and eligible envelopes.

```
eligible(tx) iff
    envelope_signature_valid(tx) and
    policy_valid(tx) and
    fee_ticket_valid(tx) and
    receipt_quorum_valid(tx) and
    current_slot <= expiry_slot(tx) and
    not duplicate(tx)

forced_valid(tx) iff
    eligible(tx) and forced_entry_valid(tx)

admitted_set =
    deterministic_prefix(
        sort(forced_valid, H("forced", tx)),
        capacity
    ) followed by
    deterministic_prefix(
        sort(eligible - admitted_forced, H("admission", admission_seed, tx)),
        remaining_capacity
    )

excluded_set = all_seen - admitted_set
exclusion_reason(tx) = first deterministic matching reason
```

The rule is deterministic over sets, not arrival order. Equivalent inputs must produce identical admitted/excluded sets and reason codes.

29.2.4 AdmissionCommit

The leader commits the admission result before ordering randomness is available.

```
Preconditions:
    admitted_set and excluded_set computed
    randomness_revealed = false

Effects:
    admission_commit_posted' = true
    admitted_root' = Commit(admitted_set)
    excluded_root' = Commit(excluded_set)
    exclusion_reason_root' = Commit(exclusion_reason)
```

29.2.5 RandomnessReveal

Committee randomness becomes available only after admission commitment.

```
Preconditions:
    admission_commit_posted = true
    threshold_randomness_shares >= t

Effects:
    randomness_revealed' = true
    randomness' = CombineRandomnessShares(shares)
```

29.2.6 OrderCommit

The ordered list is a deterministic function of the committed admitted set and randomness.

```
Preconditions:
    admission_commit_posted = true
    randomness_revealed = true

Effects:
    ordered_list' = sort(admitted_set, H("order", randomness, tx.commitment), tx.commitment)
    ordered_root' = Commit(ordered_list')
```

No transition after `AdmissionCommit` may modify `admitted_set`. No transition after `OrderCommit` may modify `ordered_list`.

29.2.7 DecryptionReveal

The decryption committee releases shares only after admission and order are both committed.

```
Preconditions:
    admission_commit_posted = true
    ordered_root committed
    decryption_shares_available >= t

Effects:
    reveal_root' = Commit(decrypted_payloads)
    failed_decryption_set' = {tx in admitted_set : decrypt(tx) failed}
```

29.2.8 PostRevealValidation

The validator checks the revealed inner Solana transactions.

```
post_reveal_valid(tx) iff
    tx in admitted_set and
    tx notin failed_decryption_set and
    signatures_valid(tx) and
    blockhash_or_nonce_valid(tx) and
    fee_payer_valid(tx) and
    account_header_matches_envelope(tx) and
    compute_budget_within_bound(tx) and
    payload_commitment_matches(tx) and
    program_constraints_valid(tx) and
    quote_constraints_valid(tx)
```

Invalid transactions are recorded in the invalid root and are not executed. Their invalidity can trigger bonds, reputation penalties, or forced-exclusion semantics.

29.2.9 InstallFairFence

The validator installs the protected account set before executing the fair batch.

```
protected_accounts = union(locks(tx) for tx in post_reveal_valid_set)
fence_active = true
```

A domain policy may add accounts to the fence, but it may not omit any account locked by a valid fair transaction.

29.2.10 ExecuteFairTx

A fair transaction is ready only if all earlier conflicting valid fair transactions have terminated.

```
rank(tx) = position of tx in ordered_list
conflict(tx_i, tx_j) iff locks(tx_i) overlap locks(tx_j)

deps(tx_i) = {tx_j in post_reveal_valid_set : rank(tx_j) < rank(tx_i) and conflict(tx_j, tx_i)}

ready(tx_i) iff
    tx_i in post_reveal_valid_set and
    tx_i notin executed_set and
    deps(tx_i) subset terminated_set and
    locks(tx_i) compatible with currently active locks
```

This is the dependency-graph scheduler. It replaces the unsafe active-lock-only scheduler.

29.2.11 ExecuteNormalTx

A normal transaction can execute before the fair boundary only if it does not conflict with the active fair fence.

```
Pre-boundary normal execution allowed iff
    domain(tx) = NORMAL and
    (fence_active = false or locks(tx) disjoint protected_accounts)
```

If the normal transaction conflicts with protected accounts, deterministic policy must either delay it until the fair boundary closes, reject it for the segment, or require conversion into the fair domain. It may not execute before or between the protected fair batch.

29.2.12 CloseFairBoundary

The fair boundary closes after all valid fair transactions have terminated or after deterministic abort/fallback rules fire.

```
Preconditions:
    post_reveal_valid_set subset terminated_set

Effects:
    fence_active' = false
    fair_boundary_closed' = true
```

29.2.13 ReplayVerify

Replay accepts a fair-domain segment only if all proof predicates hold:

```
ReplayAccept(batch) iff
    fair_proof_bundle_available and
    fair_proof_bundle_root_matches and
    admitted_root = Commit(admitted_set) and
    ordered_root = Commit(
        sort(admitted_set,
             H("order", randomness, tx.commitment),
             tx.commitment)) and
    reveal_root matches revealed payload commitments and
    post_reveal_valid_set exactly matches validation predicates and
    protected_accounts includes every lock of every post_reveal_valid transaction and
    fair_schedule preserves conflicting committed order and
    no normal_pre_boundary transaction conflicts with protected_accounts and
    forced inclusion rules and exclusion reasons are deterministic and valid
```

In protocol-enforced mode:

```
not ReplayAccept(batch) => RejectFairDomainReplay
```

29.3 Invariants and proof obligations

29.3.1 P1. Committed conflicting order cannot be inverted

For all valid fair transactions `tx_i`, `tx_j` in a batch:

```
rank(tx_i) < rank(tx_j) and conflict(tx_i, tx_j)
    => execution_position(tx_i) < execution_position(tx_j)
```

This follows from the dependency definition: every earlier conflicting transaction is in the later transaction's dependency set. A later conflicting transaction cannot become ready until every earlier conflicting transaction has terminated.

29.3.2 P2. Leader cannot bias ordering after admission commitment

After `AdmissionCommit`, the admitted set is immutable. After randomness is revealed, the order is uniquely determined by the committed admitted set and randomness:

```
admitted_set_after_randomness = admitted_set_at_commit
ordered_list = sort(admitted_set_at_commit,
                    H("order", randomness, tx.commitment),
                    tx.commitment)
```

Therefore a leader cannot alter the admitted set or order after seeing randomness without producing an invalid proof. This property assumes the commitment is binding and the randomness was unavailable before admission commitment.

29.3.3 P3. Invalid fair-domain replay is rejected

A fair-domain segment is replay-valid if and only if all proof predicates hold. Any mismatch in admitted roots, ordering roots, reveal roots, validation roots, fence coverage, schedule proof, forced-inclusion proof, or Fair Proof Bundle availability rejects fair-domain replay.

```
exists invalid_predicate(batch) => not ReplayAccept(batch)
```

29.3.4 P4. Normal-lane fences cannot be bypassed

While a fair fence is active:

```
normal_pre_boundary(tx) and locks(tx) overlap protected_accounts => replay_invalid
```

This prevents normal or private transactions from touching protected fair-domain accounts before or between fair-batch execution.

29.3.5 P5. Forced-inclusion rules are deterministic

For equivalent forced queues, fair buffers, expiry data, validity data, capacity, and admission seed:

```
Admission(input_1) = Admission(input_2)
```

where `input_1` and `input_2` differ only by arrival order or internal collection ordering. Determinism is achieved by canonical deduplication, explicit reason precedence, and sorted hash keys.

29.3.6 P6. Committee reveal rules preserve liveness and secrecy under threshold assumptions

Assume a threshold `t` and fewer than `t` corrupt committee members.

Secrecy before the reveal gate:

```
not (admission_commit_posted and ordered_root committed) => shares_available_to_adversary < t
```

Liveness after the reveal gate:

```
admission_commit_posted and ordered_root committed and available_shares >= t
    => Reveal transition is enabled
```

This is a protocol-level liveness condition under weak fairness of enabled share-release transitions. It still requires production engineering against DoS, partitions, and key-management failure.

29.3.7 P7. Fair Proof Bundle non-availability has deterministic consequences

In protocol-enforced mode:

```
not fair_proof_bundle_available or not fair_proof_bundle_root_matches => RejectFairDomainReplay
```

In non-enforced certification mode, the same condition may accept ordinary state replay but must revoke `FairSequenced` certification. The protocol-enforced validator-client mode specified by this paper uses the stronger replay-rejection rule for certified fair domains.

P8. Account-set closure cannot be bypassed For any selected offer, settlement transfer, execution report, refund, bond update, or downstream CPI in a `FairDeFiSettled` object:

```
UsedAccounts(x) subset ClosedAccountSet(x)
```

If the settlement program touches an account outside the closure root, replay rejects the fair-domain segment. This property is necessary because Solana transactions and validator scheduling require the relevant account set before execution.

P9. Atomicity-mode labels are truthful A settlement labeled `NativeAtomic` must fit and settle in one native Solana transaction. A settlement labeled `FairAtomicSequence` must include all-or-none sequence proof and no third-party insertion. A settlement labeled `StagedFairSettlement` must include escrow, clearing, finalization, and refund proofs.

```
Label(x) = NativeAtomic => settlement_mode(x) = NATIVE_SINGLE_TX
Label(x) = FairAtomicSequence =>
    settlement_mode(x) = FAIR_ATOMIC_SEQUENCE AND SequenceProofValid(x)
Label(x) = StagedFairSettlement => settlement_mode(x) = STAGED_ESCROW AND EscrowProofValid(x)
```

P10. Fair Proof Bundle finality is deterministic In protocol-enforced mode:

```
VoteEligibleFairSegment(batch) => FairProofBundleReconstructableBeforeVote(batch)
not FairProofBundleReconstructableBeforeVote(batch) => not FairSegmentFinalizable(batch)
```

This property prevents fair-domain finality from depending on unverifiable proof roots.

P11. Fair DeFi settlement respects user constraints For any filled intent:

```
actual_output >= signed_min_output
AND fees <= signed_max_fee
AND selected_venue in allowed_venues
AND selected_programs subset allowed_programs
AND expiry_slot >= execution_slot
AND quote_class_rules_satisfied
```

If any predicate fails, the settlement either reverts atomically or emits a deterministic failure report under the declared mode. It cannot silently degrade to a worse execution.

29.4 Bounded model checker

An executable bounded model checker accompanies this paper. It exhaustively checks the properties above over the following finite bounds:

```
accounts: {A, B, C}
scheduler transactions: up to 4
admission transactions: 4
committee sizes: n in 3..7
```

```
committee threshold: t in 2..n
committee corruption assumption: corrupted < t
```

The checker enumerates all relevant account-lock assignments and committed-order permutations for the scheduler property, all post-randomness admitted-set and order tamper cases for the order-binding property, all arrival-order permutations for forced inclusion, all small committee threshold states, and explicit replay tamper cases for invalid order, missing Fair Proof Bundle, root mismatch, invalid payload execution, missing fair transaction, missing fence account, and normal fence bypass.

The checked result is:


```
{
  "ALL_PROPERTIES_HOLD_IN_BOUNDED_MODEL": true,
  "P1_committed_conflicting_order_not_inverted": {
    "active_lock_bug_detected": true,
    "cases": 59787,
    "schedules_checked": 139983,
    "ok": true
  },
  "P2_no_leader_bias_after_admission_commit": {
    "valid_cases": 120,
    "tamper_rejections_checked": 2072,
    "ok": true
  },
  "P3_invalid_fair_domain_replay_rejected": {
    "tamper_cases_rejected": 7,
    "ok": true
  },
  "P4_normal_lane_fences_not_bypassed": {
    "cases": 49,
    "ok": true
  },
  "P5_forced_inclusion_deterministic": {
    "set_cases": 15,
    "permutation_cases": 984,
    "ok": true
  },
  "P6_committee_liveness_and_secrecy": {
    "cases": 2020,
    "liveness_enabled_cases": 180,
    "ok": true
  },
  "P7_fair_proof_bundle_non_availability_deterministic": {
    "cases": 8,
    "protocol_enforced_missing_fair_proof_bundle": "REJECT_FAIR_DOMAIN_REPLAY",
    "ok": true
  }
}
```


The `active_lock_bug_detected` field is intentional. It confirms that the checker detects the old unsafe schedule in which `T1` locks `A`, `T2` locks `A,B`, and `T3` locks `B`; an active-lock-only scheduler could start `T1`, block `T2`, then start `T3`, thereby inverting `T2` and `T3`. The dependency-graph scheduler rejects that schedule.

29.5 What remains outside this model

The bounded model establishes that the specified control logic has no counterexample under the selected finite bounds. The following items still require separate work:

1. Cryptographic proofs for commitment binding, signature unforgeability, threshold encryption secrecy, DKG correctness, and randomness unbiasability.
2. Larger state-space checking in TLA+, Ivy, Alloy, Apalache, or a custom Rust model linked to validator-client types.
3. Differential tests between the model and the production scheduler, replay verifier, and Fair Proof Bundle verifier.
4. Adversarial network tests for committee DoS, delayed shares, Fair Proof Bundle withholding, gateway equivocation, and partition recovery.
5. Performance benchmarks under realistic account-lock distributions and Solana-style slot timing.

The model nonetheless converts the core design from prose into executable invariants. Any implementation that claims `FairSequenced` validity should include these properties as conformance tests and should fail CI on any counterexample.

29.6 Atomic Fair DeFi model extension

The bounded validator-control model is extended with a state-machine settlement model.

```
Intent = {
    intent_id,
    user,
    input_mint,
    output_mint,
    amount_in,
    min_amount_out,
    max_fee_bps,
    allowed_venues,
    allowed_programs,
    quote_mode,
    settlement_mode,
    expiry_slot,
    nullifier,
    signature_valid
}

Offer = {
    offer_id,
    maker_or_solver,
    intent_id,
    amount_out,
    fee_bps,
    inventory_authorized,
    quote_commitment_valid,
    expiry_slot,
    bond_valid,
    signature_valid
}

Settlement = {
    selected_offer,
    actual_out,
    user_fee,
```

```
    solver_surplus,
    venue_payment,
    downstream_calls,
    execution_report,
    success
}
```

The settlement transition is:

```
SelectValidOffer(intent, offers, state, policy) =
    deterministic_argmax(
        offers satisfying ValidOffer(intent, offer, state, policy),
        objective = user_output_then_fee_then_tiebreaker
    )

ExecuteFairSettlement(intent, selected_offer, composability_plan) =
    require intent.signature_valid
    require current_slot <= intent.expiry_slot
    require selected_offer.amount_out >= intent.min_amount_out
    require selected_offer.fee_bps <= intent.max_fee_bps
    require selected_offer.venue in intent.allowed_venues
    require every downstream call in intent.allowed_programs
    require token transfers, fees, rebates, bonds, and reports balance
    commit all state changes atomically or revert all state changes
```

Additional invariants checked by the model extension are:

P8. Deterministic offer selection. Equivalent valid input sets produce the same selected offer and tie-break result, independent of arrival order.

P9. User constraints cannot be bypassed. No successful settlement has `actual_out < min_amount_out`, `fee_bps > max_fee_bps`, an unapproved venue, or an unapproved downstream program.

P10. Firm quote degradation cannot settle. If the quote class is `FIRM` or `RESERVED`, settlement below the committed output is rejected or classified as a violation funded by the venue bond.

P11. Atomic composability is all-or-nothing. If any required downstream call fails, the model either reverts the entire settlement or records a deterministic failure report without partial user settlement.

P12. Solver surplus follows the published split. A solver cannot receive surplus outside the on-chain surplus-sharing rule.

These properties do not replace program-level formal verification. The production Fair Intent, Fair Offer, Fair Auction, and Fair Settlement programs should be verified separately at the Rust/SBF-program level, including arithmetic bounds, token-account authority, CPI target restrictions, replay protection, and account ownership checks.

30 Economic Equilibrium and Market Design

The technical protocol makes fair sequencing enforceable. The economic design must make fair sequencing stable. A protocol that prevents content-based reordering but underpays validators, overburdens venues, prices out users, or recreates a new admission auction will be bypassed by rational participants. Fair Outcomes therefore requires a market design that redirects existing private-orderflow and priority-fee expenditure toward verifiable work without selling transaction position.

This section specifies the economic mechanism. Its objective is not to eliminate all market profit. It is to remove extractive discretion from the transaction supply chain and replace it with com-

pensation for observable services: admission capacity, witness receipts, threshold reveal, proof generation, Fair Proof Bundle availability, quote guarantees, and execution-quality reporting. The existing technical enforcement layer defines fair envelopes, admission proofs, batch ordering, conflict fences, dependency scheduling, forced inclusion, and quote commitments. This section defines the incentive system required to make that enforcement path an equilibrium.

30.1 Economic objective

The economic objective is to maximize certified fair execution share subject to all of the following constraints:

```
no paid intra-batch ordering;
validator revenue adequacy;
user fee tolerability;
venue quote sustainability;
bounded congestion externalities;
bounded migration to residual MEV channels.
```

The protocol must not reproduce priority gas auctions inside the fair lane. MEV research shows that fee bidding for priority ordering can create adversarial trading dynamics and consensus-layer risks [1]. Fair Outcomes sells capacity and proof work, not rank. Solana's current fee structure includes a base fee and an optional prioritization fee that can affect scheduling likelihood in ordinary traffic [8]. Fair Outcomes keeps ordinary resource pricing, but fair-domain payments must not determine the relative order of admitted fair transactions.

30.2 Economic participants and decisions

The economic system includes seven adaptive participant classes.

Participant	Decision variable	Desired equilibrium behavior
Validators	Run fair client, produce valid proofs, reject private fair-domain ordering	Fair operation dominates private-orderflow deviation
Users	Choose fair, balanced, or normal execution	Users choose fair path when expected execution quality exceeds fee and delay cost
Wallets and apps	Subsidize fair routing, expose quote class, choose fallback policy	Apps route sensitive flow through fair domains by default
Aggregators	Sign route commitments, avoid hidden rerouting, sponsor users	Aggregators compete on realized execution quality
RFQ venues and PropAMMs	Choose quote class, bond firm quotes, update prices	Venues quote tightly when compensated for inventory and adverse-selection risk
Searchers and solvers	Submit solver commitments, arbitrage residual mispricing	Searchers compete for surplus creation, not user extraction
Delegators and stakers	Delegate to certified fair validators	Stake premium rewards fair behavior and punishes private-orderflow deviation

Private infrastructure provides economically valuable services, including low-latency submission, MEV protection, and cross-transaction atomicity. Jito's public documentation describes bundles as sequential and atomic units of up to five transactions within a slot [12]. The fair-market design must replace useful private services with public, proof-valid equivalents rather than merely condemning private infrastructure.

30.3 Fee separation principle

Fair Outcomes uses strict fee separation. A fair-domain transaction may pay for:

```
resource consumption;
admission capacity;
witnessing;
threshold reveal;
proof generation;
Fair Proof Bundle availability;
quote insurance;
app or venue certification.
```

A fair-domain transaction may not pay for:

```
earlier rank inside an admitted fair batch;
private fair-domain insertion;
normal-lane conflict bypass;
selective exclusion of competitors.
```

This principle is replay-enforced for certified fair domains. A fair-domain block segment is invalid if transaction order depends on fee size after admission. Fees can affect whether a transaction obtains scarce fair-lane capacity. Fees cannot affect where the transaction lands after admission. The order is determined only by the fair ordering function over committed transaction commitments and post-admission randomness.

30.4 Four distinct markets

Fair Outcomes separates the economy into four markets that must not collapse into one another.

30.4.1 Resource market

The resource market pays for ordinary computation, signatures, account locks, and runtime costs. This corresponds to the base blockchain fee model. Solana transaction fees include ordinary resource fees and optional compute-priority fees in normal traffic [8]. Fair-domain transactions still pay resource fees. They do not get free computation.

30.4.2 Admission market

The admission market allocates scarce fair-lane capacity. It does not allocate order. The admission fee is paid for a capacity ticket:

$$FairAdmissionFee_{tx} = f_d(t) \cdot CU_{declared} + g_d(t, A_{write}) + q_d(c)$$

where:

- $f_d(t)$ is the domain base admission fee;
- $CU_{declared}$ is the declared compute budget;
- $g_d(t, A_{write})$ is a capped account-hotness surcharge for protected writable accounts;
- $q_d(c)$ is the fixed fee for the selected capacity class c.

The account-hotness component is justified because Solana congestion and prioritization are account-sensitive. Solana's `getRecentPrioritizationFees` method can return recent prioritization-fee samples filtered to transactions that lock specified writable accounts [9]. Admission fees determine whether the transaction is eligible for a capacity class. They do not determine intra-batch order.

30.4.3 Fair-work market

The fair-work market pays validators, witnesses, committees, indexers, and data-availability providers for verifiable work. Work units include:

```
valid admission proof;
valid exclusion proof;
ordering proof;
threshold reveal proof;
dependency-schedule proof;
fair/normal fence proof;
execution report root;
available Fair Proof Bundles.
```

The fair-work market replaces private-orderflow revenue with proof-valid compensation. This is necessary because private infrastructure already creates explicit reward channels; public Jito ecosystem material describes MEV and priority-fee distribution as a validator/staker reward stream [16].

30.4.4 Quote-risk market

The quote-risk market compensates venues for taking firm, reserved, or conditional quote risk. PropAMMs and RFQ venues may use off-chain predictive models and on-chain price updates to quote more tightly than passive AMMs. Solana's PropAMM explainer describes PropAMMs as maintaining predictive price models off-chain and sending market prices on-chain [14].

The quote-risk market defines:

```
firm quote spread;
reservation fee;
quote bond;
revert-insurance premium;
venue score reward.
```

This lets venues quote tightly without absorbing unbounded stale-price or latency risk.

30.5 Admission mechanism

Fair-domain capacity is divided into deterministic quotas:

$$C_d = C_{forced} + C_{retail} + C_{app} + C_{timecritical} + C_{open} + C_{safety}.$$

Quota	Purpose
`C_forced`	Previously omitted eligible fair transactions
`C_retail`	Wallet/user flow protected from wholesale crowd-out
`C_app`	App-sponsored or aggregator-sponsored flow
`C_timecritical`	Time-sensitive but non-order-buying flow
`C_open`	General fair-lane demand
`C_safety`	Oracle, liquidation, and protocol maintenance flows when domain rules allow

Within each quota, admission follows public eligibility rules. If demand exceeds quota capacity, transactions are selected by deterministic hash over transaction commitment and admission randomness, not by arrival speed and not by paid order.

30.5.1 Capacity classes

The fair lane uses fixed published classes rather than continuous priority bidding.

Class	User meaning	Fee property	Ordering property
`RETAIL_FAIR`	Normal user protection	Low or app-subsidized	Randomized after admission
`STANDARD_FAIR`	Default fair execution	Dynamic base admission fee	Randomized after admission
`TIME_CRITICAL_FAIR`	Short-expiry or liquidation-sensitive	Higher fixed class fee	Domain-specific fair order
`FORCED_FAIR`	Prior eligible omission	No extra urgency bid	Deterministic forced-inclusion order
`SPONSORED_FAIR`	App/venue pays	App-funded	Randomized after admission

There is no per-transaction tip to move earlier inside any class.

30.5.2 Dynamic base admission fee

Each fair domain maintains a base admission fee:

$$b_{d,t+1} = clamp\left(b_{d,t} \cdot exp\left(k_d \cdot \frac{u_{d,t} - u_d^*}{u_d^*}\right), b_d^{min}, b_d^{max}\right)$$

where:

- $u_{d,t}$ is observed utilization in domain d;
- u_d^* is target utilization;
- k_d is an adjustment speed;
- b_d^{min} and b_d^{max} bound volatility.

This follows the market-design intuition of adaptive base fees: congestion should change the cost of scarce blockspace, but the mechanism should avoid continuous rank bidding. Transaction-fee-mechanism research around EIP-1559 studies how base-fee mechanisms differ from first-price auctions and how active block producers change the incentive analysis [17].

30.5.3 Local account congestion fee

Fair domains also maintain a local account-hotness surcharge:

$$h_{a,d,t+1} = clamp\left(h_{a,d,t} + \eta_d \cdot (lock\_util_{a,d,t} - lock\_target_{a,d}), h_a^{min}, h_a^{max}\right).$$

A transaction touching writable accounts A_{write} pays:

$$g_d(t, A_{write}) = min\left(G_d^{max}, \sum_{a \in TopK(A_{write})} h_{a,d,t}\right).$$

This prices congestion around hot pools, vaults, oracle accounts, and liquidation accounts without letting a user buy earlier rank. The cap prevents complex transactions from becoming prohibitively expensive merely because they touch many accounts.

30.6 Validator revenue replacement

Validators will not abandon private-orderflow revenue unless fair operation pays competitively. The equilibrium condition for validator v is:

$$\begin{aligned} & R_v^{fair} - Cost_v^{fair} - Risk_v^{fair} \\ \geq\ & R_v^{private} - Cost_v^{private} - Risk_v^{private} + \epsilon. \end{aligned}$$

Where:

$$\begin{aligned} R_v^{fair} = {} & R_v^{base} + R_v^{normal\_priority} + R_v^{fair\_work} \\ & + R_v^{app\_subsidy} + R_v^{stake\_premium} + R_v^{certification}. \end{aligned}$$

and:

$$\begin{aligned} R_v^{private} = {} & R_v^{base} + R_v^{normal\_priority} + R_v^{MEV\_tips} \\ & + R_v^{private\_relay} + R_v^{offchain\_sidepayments}. \end{aligned}$$

The fair design does not require eliminating all normal priority fees. It requires that fair-domain order cannot be purchased. Validators may still earn ordinary fees for non-fair domains, but fair-domain rewards are paid only for valid proofs and available Fair Proof Bundles.

30.6.1 Fair Work Reward Pool

The protocol maintains a Fair Work Reward Pool:

$$\begin{aligned} Pool_t = {} & AdmissionFees_t + AppSubsidies_t + VenueCertificationFees_t \\ & + SolverSurplusFees_t + ProtocolSubsidy_t + PenaltyInflows_t. \end{aligned}$$

Each validator receives:

$$Reward_{v,t} = Pool_t \cdot \frac{ProofWeight_{v,t}}{\sum_j ProofWeight_{j,t}}.$$

The proof weight is:

$$\begin{aligned} ProofWeight_{v,t} = {} & \alpha_1 FairCU_{v,t} + \alpha_2 ValidFairSlots_{v,t} \\ & + \alpha_3 FairProofBundleAvailability_{v,t} + \alpha_4 ForcedInclusionService_{v,t} \\ & + \alpha_5 Uptime_{v,t} - \alpha_6 Violations_{v,t}. \end{aligned}$$

Rewards vest after a challenge window. If a proof is invalid, the reward is withheld and the validator bond is slashed.

30.6.2 Revenue adequacy controller

The system maintains an estimated private-revenue baseline:

$$\widehat{R}_t^{private}.$$

This estimate is derived from observed priority-fee revenue, known tip-router distributions, bundle-market telemetry where available, and certified validator opportunity cost. Public Jito documentation frames MEV and priority-fee distribution as a validator/staker reward stream, so Fair Outcomes treats revenue replacement as a first-class mechanism rather than an optional moral appeal [16].

The reward controller targets:

$$\widehat{R}_t^{fair} \geq \rho \cdot \widehat{R}_t^{private},$$

where $\rho \geq 1$ during migration and may decrease after fair routing becomes the dominant market structure.

If fair validator revenue is below target for E epochs, the protocol can increase:

```
app subsidies;
venue certification fees;
fair admission base fees;
solver surplus share to the pool;
temporary protocol emissions;
stake-delegation incentives.
```

If fair validator revenue exceeds target, the protocol reduces subsidies before increasing user fees.

30.7 Fee distribution

A fair-domain payment is split by service role.

Recipient	Paid for	Suggested initial share
Leader validator	Admission, fair scheduling, proof generation	35-45%
Witness/gateway network	Ingress receipts and propagation proofs	10-20%
Decryption committee	Randomness and reveal shares	10-15%
Fair Proof Bundle DA / SolanaCDN-indexers	Proof storage and retrievability	10-15%
Insurance/rebate pool	Revert refunds and user rebates	10-20%
Protocol treasury/burn	Anti-manipulation buffer	0-15%

The exact split is governance-controlled and should be adjusted based on measured cost. A nonzero burn or treasury component helps reduce incentives for validators to manipulate congestion in order to increase their own fee revenue.

30.8 Venue and quote economics

Quote integrity will fail if firm quotes make venues unprofitable. Venues must be allowed to price inventory risk while being prevented from silently degrading execution. A venue chooses quote class q to maximize:

$$U^{venue}(q) = Spread(q) + VolumeBenefit(q) + RoutingScoreBenefit(q) + CertificationBenefit(q) - InventoryRisk(q) - AdverseSelectionRisk(q) - BondRisk(q) - OperationalCost(q).$$

The protocol should make firm and reserved quotes attractive when venues can manage risk, and make indicative quotes clearly labeled when they cannot.

30.8.1 Quote classes as economic instruments

Quote class	Venue risk	User protection	Economic tool
Firm	High	Highest	Wider spread, short expiry, bond
Reserved	Medium-high	High	Reservation fee, inventory hold
Conditional	Medium	High if conditions hold	State/oracle bounds
Indicative	Low	Low	Must be labeled
Best effort	Low	Slippage-bound only	Scorecard monitoring

Aggregator quote APIs commonly produce route plans from inputs such as input mint, output mint, amount, and slippage [15]. Fair Outcomes adds a signed economic commitment layer around that quote/route boundary.

30.8.2 Firm quote bond

Each venue maintains a bond account:

$$Bond_v \geq \lambda \cdot VaR_v(\Delta t, size, volatility) + PenaltyFloor.$$

For a firm quote violation:

$$Penalty = min(Bond_v,\ \kappa_1(PromisedOut - ActualOut)^+ + \kappa_2 UserFees + \kappa_3 ReputationPenalty).$$

The bond should be large enough to make intentional degradation unprofitable but not so large that only the largest venues can quote.

30.8.3 State-bound and oracle-bound quotes

Firm quotes may include explicit bounds:

```
state_bound_hash;
oracle_bound_hash;
max_age_slots;
max_price_move_bps;
expires_at_slot.
```

A quote is enforceable only while those conditions hold. This is essential for PropAMMs because their pricing depends on fresh off-chain model updates and on-chain price updates [14].

30.8.4 Venue score

Routers and wallets should rank venues by execution quality, not only by displayed expected output.

$$\begin{aligned} VenueScore_v = {} & w_1 Tightness_v + w_2 FillReliability_v + w_3 LowViolationRate_v \\ & + w_4 LowRevertRate_v + w_5 QuoteAvailability_v \\ & + w_6 FairProofBundleCompleteness_v - w_7 ComplaintRate_v. \end{aligned}$$

A venue that displays aggressive quotes but reverts frequently should lose score. A venue that quotes slightly wider but honors firm commitments should gain routing share.

30.9 User revert tolerance and insurance

Fair execution may increase reverts in some cases because the system refuses to silently degrade a firm quote. This is a protocol-level feature but a UX cost for users.

The wallet must present three execution modes.

Mode	Behavior	User profile
`GUARANTEED`	Firm or reserved quote only; fill-or-revert	User wants certainty
`BALANCED`	Firm where available; conditional/best-effort allowed with labels	Default retail mode
`FAST`	Normal public execution; no full fair guarantees	User prioritizes speed/landing

User utility is:

$$\begin{aligned} U_{fair}^{user} = {} & ExpectedFillImprovement + ReducedSandwichLoss + QuoteCertainty \\ & - FairFees - DelayCost - RevertCost. \end{aligned}$$

The fair path is adopted when:

$$U_{fair}^{user} \geq U_{normal}^{user}.$$

30.9.1 Revert refund rules

The protocol classifies fair-domain reverts.

Revert cause	User fee treatment	Economic responsibility
Firm quote violated by venue	Refund fair admission and witness fees	Venue bond
Committee reveal failure	Refund fair admission fee	Committee/insurance pool
Validator fair proof invalid	Refund fair fee; slash validator	Validator bond
User quote expired before submission	No full refund	User/app
State-bound condition failed	Partial refund depending on mode	Quote class rules
User transaction invalid	No refund except unused insurance	User/app

Revert cause	User fee treatment	Economic responsibility
Normal market movement under best-effort quote	No firm-quote refund	User accepted best-effort class

This makes users more willing to choose strict quote integrity because avoidable reverts are insured by the party that caused them.

30.9.2 Revert insurance premium

Users or apps may pay:

$$Premium_{tx} = p_{revert} \cdot ExpectedRefund + RiskMargin.$$

The premium can be app-subsidized for retail flow. Insurance should cover infrastructure-caused reverts, not speculative market losses.

30.10 Solver and searcher migration

The protocol should not treat every searcher as malicious. Some searchers discover arbitrage, improve routing, liquidate unsafe positions, or provide backstop liquidity. The design problem is to remove user-extractive ordering control while preserving surplus-creating competition.

Fair Outcomes introduces public solver commitments.

```
SolverCommitment {
    solver_id,
    domain_id,
    target_intent_hash,
    surplus_share_bps,
    max_user_loss_bps,
    route_constraints_hash,
    fee_constraints_hash,
    expiry_slot,
    encrypted_solution,
    bond,
    signature
}
```

Solvers compete to improve a user-authored intent subject to route and quote constraints. They do not get to insert arbitrary private bundles around the user. Any solver surplus is split by a published rule:

$$Surplus = ActualOut - UserGuaranteedOut.$$

$$UserShare = \lambda \cdot Surplus,$$

$$SolverShare = (1 - \lambda - \mu) \cdot Surplus,$$

$$FairPoolShare = \mu \cdot Surplus.$$

The validator receives fair-work compensation, not the right to sell top-of-block order. This keeps economically useful search while eliminating privileged transaction positioning around users.

30.11 Cross-domain MEV migration

If fair swaps are protected but oracle updates, liquidations, bridges, or governance instructions are not coordinated, sophisticated actors may move extraction into cross-domain strategies.

Examples:

```
FAIR_ORACLE -> FAIR_SWAP;
FAIR_SWAP -> FAIR_LIQUIDATION;
BridgeEvent -> FAIR_SWAP;
GovernanceAction -> DeFiState.
```

The solution is a domain-interaction graph:

$$G_D = (D, E),$$

where an edge $d_i -> d_j$ means activity in domain d_i can materially affect execution in domain d_j.

For each slot, the validator computes:

$$AffectedDomains_s = Closure(G_D, ActiveDomains_s).$$

If two domains interact through protected account overlap or declared dependency edges, they must be executed under one of three policies.

Policy	Meaning
`COMPOUND_BATCH`	Domains share admission, ordering, and fence boundary
`PRECEDENCE_EDGE`	Domain A must complete before Domain B
`ISOLATED`	No material cross-domain conflict; execute independently

30.11.1 Domain precedence defaults

Interaction	Default policy
Oracle update affects swap pool	`FAIR_ORACLE` before affected `FAIR_SWAP`
Swap changes liquidation health	Compound or precedence by account set
Liquidation conflicts with oracle	`FAIR_ORACLE` before `FAIR_LIQUIDATION`
Governance changes DeFi control state	Governance fence before affected domain
Bridge mint/burn affects pool	Compound if same protected asset set

The purpose is not to eliminate cross-market information advantages. It is to prevent attackers from exploiting inconsistent sequencing across protected domains. The motivation parallels financial-market batch-auction arguments: discrete batching can reduce the value of tiny speed advantages and convert speed races into price or surplus competition [18].

30.12 Fair-lane congestion and anti-griefing

The fair/normal fence is necessary, but it can be abused. An attacker might submit fair envelopes touching hot AMM pools to fence normal traffic.

The protocol therefore imposes a fence budget:

$$\begin{aligned} FenceBudget_{a,d,t} = {} & BaseFenceBudget_{a,d} + \phi \cdot HonestDemand_{a,d,t} \\ & - \psi \cdot RecentFailedReveal_{a,d,t} - \omega \cdot RecentInvalidTx_{a,d,t}. \end{aligned}$$

A transaction consumes fence budget according to:

$$FenceCost(tx) = CU_{declared} + \sum_{a \in A_{write}} HotnessWeight_a + ExpiryUrgencyWeight + DomainRiskWeight.$$

If the budget is exhausted, additional fair envelopes are either:

1. deferred to the next fair batch;
2. rejected with `FENCE_BUDGET_EXCEEDED`;
3. admitted only with a higher bond;
4. converted to a less restrictive quote class if the user permits.

30.12.1 Anti-grinding and anti-spam bonds

Each fair intent must include a nullifier:

$$Nullifier = H(user\_scope, intent\_id, domain, quote\_id, nonce).$$

The protocol rejects duplicate nullifiers within the validity window.

For high-risk domains, the user or app posts a small bond:

$$Bond_{intent} = BaseBond_d + \chi_1 CU_{declared} + \chi_2 FenceCost + \chi_3 UrgencyClass.$$

The bond is returned if the transaction reveals and validates. It is partially burned or paid to the insurance pool if the transaction fails reveal, violates declared account bounds, or repeatedly consumes fence budget without valid execution.

30.13 Admission fee market safeguards

A fair-lane admission market can still become harmful if it prices out normal users. Fair Outcomes uses five safeguards.

30.13.1 Retail quota

A fixed fraction of fair-domain capacity is reserved for low-fee retail flow:

$$C_{retail} \geq r_d \cdot C_d.$$

Apps may sponsor this quota. Within the quota, selection is deterministic-random among eligible transactions.

30.13.2 Fee caps by domain

Each domain has a maximum admission fee:

$$b_d(t) \leq b_d^{max}.$$

When demand exceeds capacity at the cap, rationing is random or forced-inclusion based, not fee-based.

30.13.3 App sponsorship

Apps and aggregators may purchase bulk capacity credits:

$$SponsorshipCredit_{app,d}$$

that subsidize users but do not grant app-specific order preference. Sponsored transactions remain subject to the same ordering function after admission.

30.13.4 Uniform overflow clearing

For the open quota only, users may submit a maximum admission fee. The protocol computes a uniform clearing fee for capacity. All admitted open-quota transactions pay the same price. The clearing fee affects inclusion, not order.

30.13.5 No private fallback by default

Wallets must not silently fall back from fair execution to private or normal execution. Fallback mode must be user-approved and signed in the envelope.

30.14 App and aggregator incentives

Apps and aggregators should subsidize fair execution when it reduces user harm, support burden, failed trades, quote disputes, or reputational loss.

Aggregator utility is:

$$\begin{aligned} U^{agg} = {} & RoutingFees + UserRetention + ScoreBenefit + VenueRebates \\ & - FairSubsidies - ViolationPenalties - LostPrivateFlowRevenue. \end{aligned}$$

Fair routing is stable when:

$$\begin{aligned} UserRetention + ScoreBenefit + ReducedDisputeCost + CertifiedFlowPremium \\ \geq FairSubsidies + LostPrivateFlowRevenue. \end{aligned}$$

Aggregators that sign route commitments receive a public score. Hidden rerouting, undisclosed rebates, or private fallback reduce score and can trigger penalties.

30.15 LP and PropAMM incentives

Liquidity providers and PropAMMs need protection from stale-price adverse selection. Otherwise firm quotes widen or disappear. Fair Outcomes provides:

1. short quote expiry;
2. state-bound and oracle-bound quote validity;
3. fair oracle update domains;
4. reservation fees for inventory hold;
5. venue bonds sized to realistic risk, not unlimited liability;
6. execution reports that distinguish venue failure from market movement;
7. conditional quote classes for volatile conditions.

PropAMMs in particular depend on low-latency price updates. The protocol should not fence oracle updates behind swaps when those updates are necessary for quote validity. The default rule is:

$$FAIR\_ORACLE \prec FAIR\_SWAP$$

for affected venues and accounts.

30.16 Equilibrium conditions

The intended equilibrium has six conditions.

30.16.1 E1. Validators prefer certified fair operation

$$R_v^{fair} - Cost_v^{fair} - Risk_v^{fair} \geq R_v^{private} - Cost_v^{private} - Risk_v^{private} + \epsilon.$$

This is achieved through fair-work rewards, app subsidies, stake premiums, slashing for deviation, and loss of certification for private fair-domain behavior.

30.16.2 E2. Users prefer fair execution for sensitive flow

$$ExpectedFillImprovement + ReducedExtraction + InsuranceValue \geq FairFees + DelayCost + ExpectedRevertCost.$$

Wallets should default high-risk swaps, RFQs, and liquidation-sensitive actions to fair execution when this inequality is positive.

30.16.3 E3. Venues continue quoting tightly

$$Spread + VolumeBenefit + RoutingScoreBenefit + ReservationFees \geq InventoryRisk + AdverseSelectionRisk + BondRisk + OperationalCost.$$

Firm quotes should not be mandatory for all venues. Venues can choose conditional or indicative quote classes, but wallets must label them correctly.

30.16.4 E4. Searchers migrate to surplus creation

$$SolverProfit_{fair} \geq ExtractorProfit_{private} - PenaltyRisk - LostCertificationAccess.$$

Searchers should find it more profitable to compete in public solver markets than to attack protected fair domains.

30.16.5 E5. Admission does not become order auction

For any two admitted transactions tx_i, tx_j:

$$Fee(tx_i) > Fee(tx_j) \nRightarrow Order(tx_i) < Order(tx_j).$$

Extra payment can buy a class of capacity only. It cannot buy intra-batch rank.

30.16.6 E6. Congestion does not destroy accessibility

For retail flow:

$$P(admission\ within\ N\ slots) \geq TargetRetailSLO.$$

If this fails, governance must increase retail quota, app subsidy, capacity, or fee caps.

30.17 Mechanism-proof sketches

30.17.1 Proposition 1: No priority gas auction inside admitted fair batches

Given the ordering function:

$$Order(tx) = Sort(H(seed, tx\_commitment)),$$

and given that $seed$ is fixed after admission commitment but before reveal, no user can improve intra-batch rank by paying a higher fair admission fee after admission. A rational user pays only for desired capacity eligibility, not for order. This removes the core priority-auction dynamic inside the fair batch.

30.17.2 Proposition 2: Validator compliance is individually rational under revenue adequacy

If the Fair Work Reward Pool and stake premium satisfy:

$$R_v^{fair} \geq R_v^{private} + \epsilon,$$

and deviation risks slashing, loss of certification, loss of fair routing, and loss of delegation premium, then a rational validator prefers fair compliance.

30.17.3 Proposition 3: Firm quote markets remain viable under bounded obligation

If firm quotes are short-expiry, state-bound, oracle-bound, and compensated by spread, reservation fee, routing score, and certification benefits, then venues can price quote risk rather than withdrawing from the market. The bond deters intentional degradation without requiring venues to absorb unbounded market movement.

30.17.4 Proposition 4: Fair-lane congestion cannot create paid order

Even if high demand raises admission fees, the ordering function remains independent of fee. Congestion may create a capacity market, but not a rank market. Retail quotas, fee caps, app sponsorship, forced inclusion, and deterministic-random rationing bound the user-access problem.

30.17.5 Proposition 5: Cross-domain migration is bounded by compound batches and domain precedence

If interacting domains are detected by protected account overlap or declared domain edges, then the validator must apply compound batching or precedence rules. This prevents an attacker from bypassing a fair swap domain by manipulating oracle, liquidation, or governance transactions in a conflicting domain.

30.18 Governance parameters

The following parameters must be public, versioned, and slow-moving.

- u_d^*: target utilization per domain.
- k_d: admission base-fee adjustment speed.
- b_d^{min}, b_d^{max}: admission fee bounds.
- Capacity split: C_{forced}, C_{retail}, C_{app}, $C_{timecritical}$, C_{open}, and C_{safety}.
- G_d^{max}: account-hotness surcharge cap.
- $FenceBudget_{a,d}$: per-account fence budget.
- $Bond_{intent}$: anti-spam and reveal-failure bond.
- $Bond_v$: venue quote bond.

- λ: user share of solver surplus.
- μ: fair pool share of solver surplus.
- Challenge window: time to dispute invalid proofs.
- Fair Proof Bundle retention period: proof data availability requirement.
- Reward vesting period: prevents reward capture before disputes.
- Insurance reserve ratio: revert-refund solvency target.
- Venue score weights: router ranking policy.

Governance must not be able to change these parameters within the same short horizon as the trades they affect. Otherwise parameter governance itself becomes an MEV surface.

30.19 Monitoring and empirical tuning

The protocol should publish an economic dashboard with the following metrics.

30.19.1 Validator economics

```
FairRevenue / EstimatedPrivateRevenue;
ProofValidSlotRate;
FairProofBundleAvailabilityRate;
SlashingEvents;
CertificationLossEvents.
```

30.19.2 User outcomes

```
QuoteToFillDelta;
SandwichLossAvoided;
FairFeePaid;
RevertRate;
RefundRate;
AdmissionDelay.
```

30.19.3 Venue outcomes

```
FirmQuoteSpread;
FirmQuoteRevertRate;
ViolationRate;
ConditionalInvalidationRate;
VenueScore;
QuoteAvailability.
```

30.19.4 Congestion outcomes

```
DomainUtilization;
AccountHotnessFee;
RetailAdmissionSLO;
FenceBudgetExhaustion;
ForcedInclusionQueueDepth.
```

30.19.5 MEV migration outcomes

```
PrivatePathViolationRate;
CrossDomainConflictRate;
SolverSurplusCreated;
ResidualExtractionEstimate;
NormalLaneBypassAttempts.
```

30.20 Agent-based simulation plan

Before mainnet-scale rollout, the economics should be tested in an agent-based simulation.

30.20.1 Agents

The simulator should include validators choosing fair compliance or deviation; users choosing `GUARANTEED`, `BALANCED`, `FAST`, or private fallback; apps choosing subsidy level; venues choosing quote class and spread; searchers choosing solver participation or adversarial extraction; delegators choosing stake allocation; and governance choosing fee and reward parameters.

30.20.2 State variables

```
Demand_d_t;
PrivateMEV_t;
FairFees_t;
VenueRisk_t;
UserRevertTolerance_t;
CrossDomainOpportunity_t;
ValidatorRevenue_t;
FairCaptureShare_t.
```

30.20.3 Experiments

The simulator should stress:

1. normal demand;
2. burst demand around a hot token;
3. oracle volatility;
4. venue outage;
5. committee delay;
6. private MEV revenue spike;
7. fair-lane spam attack;
8. cross-domain liquidation cascade;
9. app subsidy withdrawal;
10. validator deviation attempt.

30.20.4 Acceptance criteria

The economic design is acceptable only if:

```
FairCaptureShare remains high;
FairRevenue / PrivateRevenue >= 1 for certified validators;
RetailAdmissionSLO is met;
QuoteViolationRate is low;
VenueQuoteAvailability does not collapse;
ResidualExtractionEstimate does not migrate primarily into cross-domain bypass.
```

30.21 Deployment sequence for economic stability

The economic system should launch in stages.

30.21.1 Stage 1: Measurement

Launch proof-valid fair sequencing with low volume and publish fair fees, validator costs, quote/fill deltas, reverts, Fair Proof Bundle costs, committee costs, and user conversion.

30.21.2 Stage 2: Subsidized adoption

Apps, aggregators, venues, and protocol funds subsidize retail flow. Validator fair-work rewards are deliberately above estimated private opportunity cost.

30.21.3 Stage 3: Dynamic pricing

Enable adaptive domain fees, account-hotness fees, quota tuning, and venue bonds.

30.21.4 Stage 4: Solver migration

Open public solver commitments and surplus sharing. Prohibit private fair-domain bundles by replay rule and certification rule.

30.21.5 Stage 5: Equilibrium hardening

Reduce subsidies only after `FairRevenue / EstimatedPrivateRevenue` is stable, `RetailAdmissionSLO` is stable, and venue quote availability remains high.

30.22 Residual MEV policy

Fair Outcomes should explicitly classify residual MEV.

MEV type	Protocol treatment
Sandwiching within fair domain	Prohibited by fair ordering and fences
Paid intra-batch order	Prohibited
Private fair-domain bundle	Prohibited
Hidden quote degradation	Prohibited for firm/reserved quotes
Public arbitrage after fair execution	Allowed
Solver surplus that improves user execution	Allowed and shared
Cross-domain conflict bypass	Prohibited when detected by domain graph
Off-chain/CEX latency advantage	Not eliminated; monitored
Inventory-risk compensation	Allowed through spread/quote class
Best-effort slippage usage	Allowed only with explicit label

This prevents overclaiming. The protocol eliminates specific extractive mechanisms, not all economic profit.

30.23 Summary

The Fair Outcomes economic design rests on five rules.

First, fees buy capacity and work, not order.

Second, validators are paid more for valid public fairness than for private ordering deviation.

Third, venues are not forced to offer unbounded firm quotes; they are paid and bonded according to quote risk.

Fourth, users receive explicit execution modes, revert protection, and refunds when infrastructure or venue commitments fail.

Fifth, searchers are redirected from private extraction into public surplus creation.

Under these rules, fair sequencing can become an equilibrium rather than a subsidy-dependent public good. The core economic claim is:

Fair operation is stable when proof-valid revenue, routing share, venue score, user trust, and stake premium exceed the private-orderflow alternative.

The protocol therefore does not rely on altruism. It makes fair behavior the revenue-maximizing strategy for validators, the reputation-maximizing strategy for apps, the volume-maximizing strategy for honest venues, and the execution-quality-maximizing path for users.

31 Implementation Checklist

The minimal complete validator implementation is:

1. Fair envelope ingress and packet demux.
2. Envelope canonicalization and public-header parser.
3. Receipt/quorum and propagation verification.
4. Fair buffer and nullifier index.
5. Deterministic admission and reason-coded exclusion.
6. Admission commit before randomness.
7. Committee registry, DKG transcript verification, and share aggregation.
8. Threshold randomness and ordered-root construction.
9. Threshold reveal and post-reveal validation.
10. Fair/normal account fence with exception policy.
11. Dependency-graph scheduler.
12. Fair atomic sequence executor.
13. Forced-inclusion and expiry evidence engine.
14. Quote/report verification hooks.
15. Fair Proof Bundle builder, erasure-coded publication, and checkpoint root.
16. Fair-domain replay verifier.
17. Public fair status RPC and evidence APIs.
18. Metrics, scorecards, and dispute evidence exporter.
19. Account-set closure verifier and subset-proof generator.
20. Fair Proof Bundle vote/finality gate and reconstructability checker.
21. Atomicity-mode verifier for native, fair-sequence, and staged-escrow modes.
22. Auction-size-class verifier and launch-parameter gate.
23. Committee failure ladder state machine.
24. Hot-account anti-freeze and fence-budget controller.

Without the fence, the protocol is internally fair but not outcome-fair. Without dependency-graph scheduling, the scheduler can invert conflicting order. Without post-admission randomness, leaders can bias order. Without Fair Proof Bundle availability, auditors cannot verify fairness. Without quote-aware programs, quote integrity does not apply to arbitrary venues.

31.1 Additional on-chain Fair DeFi implementation checklist

The minimal complete on-chain Fair DeFi implementation is:

1. `FairIntentProgram` with intent creation, nullifiers, expiry, allowed venues, allowed programs, quote mode, settlement mode, and user authorization checks.
2. `FairOfferProgram` with maker, solver, RFQ, PropAMM, and AMM offer commitments; inventory authorization; offer expiry; offer bonds; and deterministic offer indexing.
3. `FairAuctionProgram` with batch construction, valid-offer filtering, clearing objective, tie-break rule, surplus calculation, and clearing result commitment.
4. `FairSettlementProgram` with token transfers, fee settlement, rebate accounting, venue inventory updates, solver surplus distribution, refund path, and execution report emission.
5. `FairComposabilityProgram` or equivalent account schema for downstream lending, margin, vault, liquidation, oracle, bridge, and governance interactions.
6. `FairReportProgram` with canonical execution reports, settlement hashes, selected offer IDs,

quote IDs, route IDs, actual output, promised output, fees, failure reason, and CPI result digest.
7. `FairBondProgram` for reveal bonds, venue quote bonds, solver bonds, refund insurance, slashing, and deterministic penalty distribution.
8. SDK support for wallets, swap apps, aggregators, venues, solvers, and validators to construct canonical intents, offers, settlement transactions, and report queries.
9. `CandidateUniverse` construction and verification.
10. `AccountSetClosureProof` generation for every selected offer, settlement, report, bond, refund, and downstream CPI account.
11. Atomicity-mode-specific settlement paths for `NATIVE_SINGLE_TX`, `FAIR_ATOMIC_SEQUENCE`, and `STAGED_ESCROW`.
12. Auction-size-class enforcement for small, medium, and large auctions.
13. Deterministic refund and escrow finalization paths for staged settlement.
14. Launch-parameter enforcement hooks that reject oversized candidate sets before fair ordering.

32 Evaluation and Test Plan

32.1 Microbenchmarks

Measure gateway submit latency, witness latency, quorum aggregation, admission CPU cost, ordering-root construction, threshold randomness, threshold decryption, post-reveal validation, fence overhead, scheduler overhead, Fair Proof Bundle construction, and checkpoint submission.

32.2 End-to-end benchmarks

Measure p50/p95/p99 fair transaction landing latency, throughput under burst load, committee-failure behavior, fair-vs-normal fence delay, forced-inclusion delay, quote/fill delta, and validator revenue relative to private-orderflow baseline.

32.3 Required adversarial tests

The validator test suite must include:

1. admission determinism across clients;
2. admission commit before randomness reveal;
3. ordering-root recomputation;
4. premature randomness and premature decryption slashing;
5. committee withholding fallback;
6. post-reveal header mismatch detection;
7. duplicate nullifier rejection;
8. reveal bond penalty path;
9. `T1(A), T2(A,B), T3(B)` scheduler counterexample;
10. normal transaction delay/reject/convert under fence;
11. oracle exception misuse attempt;
12. hot-account fence griefing attempt;
13. Fair Proof Bundle withholding invalidation;
14. expired blockhash forced-inclusion evidence;
15. durable nonce forced inclusion;
16. fair intent re-signing;
17. fair atomic all-or-none execution;
18. route mismatch detection;
19. firm quote below-promised fill rejection;
20. wallet silent-fallback violation.

32.4 Fair DeFi program tests

The on-chain Fair DeFi test suite must include:

1. intent nullifier duplicate rejection;
2. expired intent rejection;
3. unapproved venue rejection;
4. unapproved downstream program rejection;
5. min-output violation rejection;
6. max-fee violation rejection;
7. firm quote below-promised fill rejection;
8. conditional quote invalidation when state bounds fail;
9. deterministic offer selection under arrival-order permutations;
10. split-fill conservation of token balances;
11. batch auction tie-break determinism;
12. solver surplus split correctness;
13. venue bond penalty correctness;
14. CPI target restriction enforcement;
15. downstream lending/margin/vault integration success path;
16. downstream integration failure rollback;
17. report emission completeness;
18. Fair Proof Bundle/report mismatch rejection;
19. normal-lane insertion attempt during settlement fence;
20. private-bundle emulation attempt around a user intent;
21. account outside closure root selected by winning offer;
22. downstream CPI attempts to use undeclared account;
23. medium auction partial-inclusion attempt;
24. large staged auction advertised as native atomic;
25. missing Fair Proof Bundle chunks before vote deadline;
26. committee failure ladder convergence across replay nodes;
27. hot-account anti-freeze activation under invalid reveal spam;
28. launch-parameter rejection for oversized candidate universe.

32.5 Product acceptance criteria

The combined validator-client plus Fair DeFi product is acceptable only if:

```
fair-domain replay invalidity rejects proof violations;
on-chain settlement cannot bypass user constraints;
atomic composability rolls back on downstream failure;
validator throughput remains within launch SLO;
retail fair admission SLO is met;
venue quote availability does not collapse;
quote/fill violations fall materially for participating flow;
Fair Proof Bundle availability remains above threshold;
solver surplus is shared according to policy;
residual extraction does not migrate primarily to cross-domain bypass.
```

33 Completeness Conditions and Residual Non-Claims

A fair-domain slot is complete only if it contains or references all of the following:

```
FairValidatorPolicy
FairAdmissionCommit
RandomnessProof
FairOrderCommit
CommitteeTranscriptRoot
```

```
FairBatchReveal or CommitteeFailureEvidence
PostRevealValidationRoot
FairBatchFence
DependencyScheduleProof
ExecutionResultRoot
ExecutionReportRoot where applicable
ForcedInclusion or ExpiryEvidence for omitted witnessed envelopes
FairProofBundleManifest and bundle chunks
CheckpointRoot
public RPC/indexer status for every tx_commitment and intent_nullifier
FairProofBundleVoteEligibilityProof
FairProofBundleReconstructabilityProof
AccountSetClosureProof where applicable
AtomicityModeProof where applicable
AuctionSizeClassProof where applicable
CommitteeFailureLadderState where applicable
```

This protocol is complete for the defined fair-domain claim. It does not claim full account privacy, universal DeFi quote enforcement without app/program adoption, execution of expired blockhash transactions, elimination of all arbitrage, or removal of legitimate market-maker spread.

The residual privacy claim is account-visible and payload-private. Full account privacy would require a different execution model. The residual execution-integrity claim is participation-bound: non-participating venues may exist but must be labeled `Normal`, `Indicative`, `BestEffort`, or `Unverified`, not `FairQuoteCommitted`.

33.1 Completeness conditions for `FairDeFiSettled`

A `FairDeFiSettled` transaction, batch, or sequence is complete only if it contains or references all of the following:

```
FairIntent or FairAtomicIntent
Valid offer set or clearing batch input root
Selected offer or clearing result proof
Settlement program version and policy hash
ComposabilityPlan where downstream programs are invoked
Token, fee, rebate, refund, and surplus accounting digest
Venue bond and solver bond consequence where applicable
ExecutionReportRoot
FairValidatorPolicy
FairAdmissionCommit
RandomnessProof
FairOrderCommit
CommitteeTranscriptRoot
FairBatchReveal or CommitteeFailureEvidence
PostRevealValidationRoot
FairBatchFence
DependencyScheduleProof
ForcedInclusion or ExpiryEvidence for omitted witnessed envelopes
FairProofBundleManifest and bundle chunks
CheckpointRoot
Public RPC/indexer status for every tx_commitment, intent_id, and intent_nullifier
AccountSetClosureProof
CandidateUniverse root
AtomicityModeProof
AuctionSizeClassProof when applicable
FairProofBundleVoteEligibilityProof
FairProofBundleReconstructabilityProof
```

If any of these objects is required by the domain manifest but missing, the transaction may still be an ordinary Solana state transition, but it must not be labeled `FairDeFiSettled`.

34 Minimum Viable Launch Scope and Launch Gates

The first production build should not attempt arbitrary batch auctions or arbitrary composable workflows. The correct MVP is deliberately narrow.

34.1 MVP scope

```
MVP =
    NATIVE_SINGLE_TX mode
    + one user intent
    + bounded candidate offer set
    + one winning offer
    + optional split fill up to max_split_fills_per_intent
    + quote-aware settlement program
    + execution report
    + fair validator sequencing
    + account-set closure proof
    + Fair Proof Bundle finality rule
```

The MVP excludes:

```
unbounded batch auctions;
arbitrary downstream CPI graphs;
multi-slot staged escrow;
large liquidation cascades;
complex cross-domain compound auctions;
unbounded solver certificates.
```

34.2 Launch gates

A domain may move from devnet to guarded mainnet only if all gates pass:

```
G1: deterministic replay across at least three independent validator builds;
G2: account-set closure fuzzing finds no bypass;
G3: scheduler property tests reject the active-lock counterexample;
G4: Fair Proof Bundle reconstructability before vote is demonstrated under partition tests;
G5: committee failure ladder reaches identical states across replay nodes;
G6: hot-account fence griefing stays below domain SLO;
G7: NativeSingleTx settlement fits compute/account/CPI limits under p99 load;
G8: quote violation and refund paths settle deterministically;
G9: no private fallback occurs without signed user fallback policy;
G10: economic pilot shows retail admission SLO and venue quote availability remain acceptable.
```

34.3 Hard default launch parameters

The following defaults are mandatory for the initial guarded launch unless governance explicitly changes them before activation:

```
settlement_mode_allowed = NATIVE_SINGLE_TX only
max_candidate_offers_per_intent = 16
max_split_fills_per_intent = 4
max_downstream_programs = 3
max_cpi_depth = domain runtime limit minus safety margin
max_fenced_accounts_per_batch = 512
```

```
max_fence_duration_ticks = 96
max_proof_bundle_publish_delay_slots = 2
min_proof_bundle_storage_providers = 7
reveal_deadline_ticks = 32
reveal_grace_ticks = 16
max_consecutive_committee_failures_before_pause = 3
retail_capacity_floor = 20% of domain capacity
forced_inclusion_capacity_floor = 10% of domain capacity
```

Later phases may enable `FAIR_ATOMIC_SEQUENCE` and `STAGED_ESCROW`, but only after the mode-specific replay, Fair Proof Bundle, and refund tests pass.

35 Deployment Path

Phase 0: Gateway and witness launch. Deploy fair submission API, witness receipts, receipt quorum, no-private-path routing policy, transparency checkpoints, and landing correlation.

Phase 1: Quote and route commitments. Integrate wallets, aggregators, PropAMMs, RFQ venues, and wrapper programs. Produce execution reports and scorecards.

Phase 2: Native fair validator devnet. Deploy fair ingress, deterministic admission, admission commits, threshold randomness, ordered roots, threshold reveal, fair fences, dependency-graph scheduling, Fair Proof Bundles, and checkpoint roots.

Phase 3: Adversarial testnet and audit. Test admission grinding, committee withholding, early reveal, invalid reveals, hot-account fence abuse, normal-lane insertion, quote mismatch, Fair Proof Bundle withholding, forced-inclusion expiry, and fair atomic partial inclusion.

Phase 4: Protocol-enforced mainnet rollout. Enable fair-domain replay verification for certified domains once enough client installs and routing support exist. Blocks that claim `FairSequenced` for a domain must include valid proof checkpoints and available Fair Proof Bundles.

35.1 Revised deployment path for on-chain atomic composable trading

The revised product path should prioritize the Fair DeFi primitive before generalized batch auctions:

1. **Single-intent atomic settlement MVP.** Users sign on-chain intents; makers and solvers submit binding offers; the program selects the best valid offer and settles atomically.
2. **Quote-aware swap integration.** Existing swap products route high-risk trades through Fair Intent and Fair Settlement programs, while preserving normal swap UX.
3. **Native fair validator devnet.** The validator client enforces fair admission, ordering, reveal, fences, dependency scheduling, Fair Proof Bundles, and replay rules for Fair DeFi domains.
4. **Composable DeFi integrations.** Lending, margin, vault, liquidation, oracle, and bridge integrations use `ComposabilityPlan` and CPI restrictions to settle in the same state transition.
5. **Batch auction mode.** Multi-intent batches clear under deterministic rules, allowing uniform clearing, split fills, and solver competition without private insertion.
6. **Mainnet protocol-enforced fair domains.** Once validator adoption is sufficient, fair-domain replay rules make invalid fair claims rejectable rather than merely reputation-damaging.
7. **Economic hardening.** Dynamic admission fees, fair-work rewards, venue quote bonds, revert insurance, and solver surplus markets are tuned using production metrics.

36 Bounded Claims and Residual Non-Claims

36.1 Full privacy is not achieved

For Solana-style scheduling, account metas and compute metadata may need to remain visible. This can reveal venue, pool, or token-pair information. The protocol prevents content-based rank manipulation through commitment-based order and account fences, but it does not claim full transaction privacy.

36.2 On-chain verification is bounded by compute and account limits

Complex batch clearing may be expensive. The practical design should let solvers compute candidates off-chain while requiring on-chain verification of feasibility, objective certificates, and settlement constraints.

36.3 Atomicity mode determines the claim

Not every desired composition can fit into a single transaction or compute budget. Mode A provides native one-transaction atomicity. Mode B provides validator-enforced fair atomic sequence semantics. Mode C provides deterministic staged escrow settlement. The paper does not claim that Mode B or Mode C are identical to ordinary Solana single-transaction atomicity.

36.4 Venue participation remains necessary

The validator client can enforce fair order. It cannot force every venue to provide firm quotes. Fair DeFi labels quote classes and enforces commitments where venues participate.

36.5 Residual MEV remains

The system reduces specific extractive mechanisms. It does not eliminate public arbitrage, informed trading, market-maker spread, inventory risk, or cross-market latency advantages.

37 Strategic Framing

The key message is:

```
Fair sequencing is not enough.
The trade itself must be a state-machine primitive.
```

Fair Outcomes becomes stronger when it is not framed only as an anti-MEV validator client. The more complete product is:

```
Fair DeFi: an on-chain atomic composable trading layer for Solana,
protected by a fair validator client.
```

This framing directly addresses the strongest version of the problem. The issue is not just that validators can order transactions. It is that today's DeFi execution path often splits economic truth across off-chain quotes, routing APIs, private relays, maker systems, and on-chain settlement. Fair DeFi collapses that split:

```
the user signs constraints;
the chain verifies offers;
the program selects or clears;
the runtime settles atomically;
the report proves what happened.
```

The final execution outcome is no longer whatever a private supply chain chooses to deliver. It is a deterministic state transition.

38 Conclusion

This paper rewrites Fair Outcomes around a bounded and implementable version of 100% on-chain atomic composable trading with the rest of the Solana state machine. The native fair validator client remains necessary: it provides witnessed ingress, encrypted pre-transaction envelopes, deterministic admission, post-admission randomness, threshold reveal, fair/normal account fencing, dependency-graph scheduling, forced inclusion, replay rules, Fair Proof Bundle finality, and Fair Proof Bundle availability. But fair sequencing alone is not the full product.

The complete product is Fair DeFi: on-chain intents, closed candidate universes, binding offers, deterministic selection and auction rules, account-set closure proofs, bounded atomicity modes, atomic or staged settlement, downstream composability, execution reports, venue bonds, solver surplus sharing, and state-machine-enforced quote integrity. In this model, off-chain systems may discover opportunities, but they cannot decide the final user outcome unless their commitments are accepted, account-closed, and verified on-chain.

The final claim is deliberately bounded:

```
Fair Outcomes works for certified fair-domain flows whose validator proofs,
account-set closure, atomicity mode, Fair Proof Bundle data, committee reveal state,
settlement program, and execution report all validate under the domain manifest.
```

The protocol does not claim unbounded full privacy, arbitrary native atomicity for all DeFi workflows, universal quote enforcement for non-participating venues, or elimination of all economic profit. It claims that the harmful mechanisms inside its certified domain - paid intra-batch order, private fair-domain insertion, normal-lane bypass, hidden quote degradation, invalid partial fair atomicity, Fair Proof Bundle withholding, and settlement outside user constraints - are either structurally prevented or replay-rejected.

This architecture changes the market structure. Validators earn for proof-valid fair execution instead of private ordering. Apps compete on realized execution quality instead of hidden routing. Venues price quote risk explicitly instead of silently degrading fills. Solvers earn by improving user outcomes instead of inserting private bundles. Users receive the state-machine outcome they authorized or a deterministic, reason-coded failure.

That is the stronger and now bounded Fair Outcomes thesis: the fair transaction path and the fair trading primitive should be built together, with explicit runtime bounds and replay-verifiable proof objects.

References

[1] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. “Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges.” IEEE Symposium on Security and Privacy, 2020. arXiv:1904.05234.

[2] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels. “Order-Fairness for Byzantine Consensus.” IACR ePrint 2020/269; CRYPTO 2020.

[3] M. Kelkar et al. “Themis: Fast, Strong Order-Fairness in Byzantine Consensus.” ACM CCS 2023; IACR ePrint 2021/1465.

[4] A. R. Choudhuri et al. “Mempool Privacy via Batched Threshold Encryption.” USENIX Security 2024.

[5] K. Mu et al. “SpeedyFair: A Faster Order-Fairness Byzantine Consensus.” NDSS 2024.

[6] H. Nagda, M. Sadoghi, and collaborators. “Rashnu: Data-Dependent Order-Fairness.” Proceedings of the VLDB Endowment, 2024.

[7] Solana Foundation. “Transactions.” Solana Documentation, accessed May 2026. https://solana.com/docs/co

[8] Solana Foundation. “Fees.” Solana Documentation, accessed May 2026. https://solana.com/docs/core/fees

[9] Solana Foundation. “getRecentPrioritizationFees.” Solana RPC Documentation, accessed May 2026. https://solana.com/docs/rpc/http/getrecentprioritizationfees

[10] Solana Foundation. “A Guide to Stake-weighted Quality of Service on Solana.” Solana Developer Guides, accessed May 2026.

[11] Anza. “Transaction Processing Unit in a Solana Validator.” Agave Validator Documentation, accessed May 2026.

[12] Jito Labs. “Low Latency Transaction Send.” Jito Labs Documentation, accessed May 2026.

[13] Jito Labs. “What is Jito?” Jito Labs Documentation, accessed May 2026.

[14] Solana Foundation. “Understanding Proprietary AMMs.” Solana Media, accessed May 2026.

[15] Jupiter Developer Platform. “Get Quote.” Jupiter Swap API Documentation, accessed May 2026.

[16] Jito Network. “TipRouter Overview.” Jito Network Documentation, accessed May 2026.

[17] T. Roughgarden. “Transaction Fee Mechanism Design for the Ethereum Blockchain: An Economic Analysis of EIP-1559.” arXiv:2012.00854, 2020.

[18] E. Budish, P. Cramton, and J. Shim. “The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response.” Quarterly Journal of Economics, 2015.

[19] Solana Foundation. “Cross Program Invocation.” Solana Documentation, accessed May 2026. https://solana.com/docs/core/cpi