DEOS Tessera

Verifiable Shared Sequencer for Ethereum L2s

TECHNICAL WHITEPAPER

Version 1.0 · March 2026
deoscomputing.io/tessera

Table of Contents

Abstract

Today's Ethereum L2 sequencers are trusted black boxes. They order transactions, produce state roots, and post them to L1, but nobody can verify that the sequencer executed correctly without re-running the entire computation. Users trust the sequencer by convention, not by proof.

DEOS Tessera is a verifiable shared sequencer built on DEOS, a deterministic event-sourced operating system. Every syscall-level state transition inside the sequencer node gets logged to a kernel event ring and is provable through a three-tier zero-knowledge proof system. The top tier produces a BN254 Groth16 proof verifiable on chain for roughly 300K gas. DEOS Tessera uses a hybrid VDF blind sequencing protocol: epoch seeds come from a class-group Verifiable Delay Function over Cl(Δ) with a Wesolowski proof (no trusted setup, formal sequential hardness), while intra-epoch ordering relies on a Poseidon2 hash chain seeded from the class-group output (ZK-efficient, around 200 constraints per step). The hybrid gives you formal sequential hardness at epoch boundaries and ZK-provable high-frequency ordering within epochs. Transactions using the blind path commit to their arrival order before plaintext is visible, which eliminates content-based MEV. The result is a sequencer whose execution is cryptographically verifiable and whose ordering carries formal hardness at epoch boundaries, backed by proof instead of convention.

---

1. Background: Ethereum, L2s, and Sequencers

1.1 Ethereum and the Scalability Problem

Ethereum is a global, decentralized ledger: a single computer shared by everyone on the internet. Every transaction is processed by thousands of nodes worldwide, verified, and permanently recorded. That decentralization is what makes Ethereum trustworthy. No single company can alter history, freeze accounts, or forge transactions.

The cost of that decentralization is throughput. Ethereum's base layer processes about 15-30 transactions per second. During periods of high demand (a popular NFT mint, a DeFi liquidation cascade, a token launch) the network gets congested. Transaction fees ("gas") spike to tens or hundreds of dollars, pricing out small transactions entirely.

1.2 Layer 2s: Scaling Without Sacrificing Security

Layer 2s (L2s) solve this by moving execution off the main Ethereum chain (Layer 1 or "L1") while keeping L1 as the ultimate source of truth for security.

The core idea: most users care about *eventual* L1 finality, knowing that their funds are provably safe and cannot be stolen, but they are willing to accept a fast "soft confirmation" from a trusted party while waiting for full L1 settlement. L2s exploit this by executing thousands of transactions off-chain, batching them into a compact summary, and posting that summary to L1 periodically.

The two main L2 families:

1. **Optimistic Rollups** (Optimism, Arbitrum, Base): Post transaction batches to L1 with a state root and assume they are correct unless challenged. Anyone who finds fraud can submit a "fraud proof" and slash the dishonest party. The dispute window is typically 7 days.
2. **ZK Rollups** (zkSync, Starknet, Polygon zkEVM): Post transaction batches along with a cryptographic validity proof that mathematically proves the state root is correct. No dispute window needed. If the proof verifies, the state root is final. Currently more expensive to generate proofs but faster to finalize.

Both approaches have reduced Ethereum transaction costs by 10-100x while inheriting L1 security for final settlement.

1.3 What Is a Sequencer?

Every L2 has a **sequencer**: a server that sits between users and the L1 blockchain. The sequencer handles:

1. **Accepting transactions** from users (wallets, dApps, smart contract calls)
2. **Ordering them** into a specific sequence
3. **Executing them** to compute updated account balances and contract states
4. **Batching** many transactions into a single compressed package
5. **Posting to L1** by submitting the batch and a cryptographic commitment ("state root") to the Ethereum base layer
6. **Providing fast receipts** that confirm to users within milliseconds that their transaction will be included, before L1 finality

Without a sequencer, users would have to wait for L1 block times (12 seconds) for every interaction. With a sequencer, transactions confirm in under a second. The sequencer trades off some decentralization for dramatically better user experience.

The state root is a compact 32-byte fingerprint that summarizes the entire state of the L2 (all account balances, all contract storage) after executing a batch of transactions. It is computed as the root of a Merkle tree over all state entries. Two identical state roots mean identical states; a different root means the state changed. L2 bridges and withdrawal contracts use the state root to verify claims about L2 balances.

1.4 The Sequencer Trust Problem

Today's L2 sequencers are operated by the L2 teams themselves: Optimism runs the Optimism sequencer, Coinbase runs the Base sequencer, Offchain Labs runs Arbitrum's. Users connect to these servers and trust them to be honest.

The trust assumption here is significant. The sequencer can:

- **Front-run users** by seeing a pending trade and inserting its own transaction first to profit
- **Extract MEV**: "Maximal Extractable Value" covers profits the sequencer can capture by reordering, inserting, or censoring transactions. Examples include buying a token before a large purchase drives the price up (sandwich attack), or liquidating a loan before the debtor can top up their collateral.
- **Censor transactions** by silently dropping them from competitors, political adversaries, or anyone it dislikes
- **Go offline**, during which no one can use the L2 at all
- **Equivocate** by showing one ordering to the L2 bridge and a different one to some users

Fraud proofs and validity proofs prove that *if* the sequencer included transactions A, B, C in that order, *then* the resulting state root X is correct. They do not prove that the sequencer was honest about *which* transactions it chose to include, or in what order.

1.5 What Is a Shared Sequencer?

A dedicated sequencer is a single server (or cluster) owned and operated by one L2. A shared sequencer is a neutral third-party sequencing service that multiple L2s plug into, similar to how multiple airlines share the same air traffic control system.

Shared sequencers offer several advantages:

- **Neutrality.** No single L2 team controls ordering, reducing conflicts of interest.
- **Decentralization.** Multiple independent operators form a committee, so no single entity can censor or front-run.
- **Cross-chain atomicity.** Because one sequencer sees transactions from multiple L2s simultaneously, it can guarantee that two transactions on different chains either both happen or neither does (atomic cross-rollup execution).
- **Operator separation.** L2 teams can focus on execution and proving; sequencing becomes a commodity service.

Current shared sequencer projects include Espresso Systems, Astria, and Radius. These use BFT (Byzantine Fault Tolerant) consensus where a committee of nodes vote on ordering, and the ordering is final if 2/3 agree. That is an improvement over a single operator, but the committee's honesty is still a social guarantee: users trust that at least 2/3 of the committee members are not colluding.

1.6 Security Model, Assumptions, and Non-Goals

Understanding what DEOS Tessera proves and what it assumes is essential for evaluating its guarantees.

Trust assumptions the protocol makes:

- **SP1 trusted setup soundness.** The Tier A Groth16 proof depends on the Aztec Ignition Powers of Tau ceremony for its security. If every one of the 214 participants colluded and retained toxic waste, a prover could generate false proofs. The scale makes this practically negligible, but it is a ceremony-level assumption, not a mathematical impossibility.
- **Poseidon2 collision resistance.** Merkle tree commitments, the VDF chain, and the Tier B transcript all rely on Poseidon2 being collision resistant. Standard cryptographic practice based on current analysis; an assumption, not a proved hardness result.
- **Data availability for disputes.** A batch's event log must remain downloadable throughout the dispute window. DEOS publishes via EIP-4844 blobs (roughly 18-day availability) and a CAS event log URI. If both are simultaneously unavailable, disputes cannot be submitted.
- **Blind sequencing is currently optional.** The VDF blind path prevents content-based MEV only for transactions that explicitly use it. Plain-path transactions are ordered at the operator's discretion, subject to slashing. Mandatory blind mode is a planned governance option, not yet active.
- **VDF ordering is not yet Groth16-proven for intra-epoch ticks.** The Poseidon2 tick chain enforces arrival-tick ordering at the kernel level and logs it to the auditable event record. Proving the tick chain inside the SP1 circuit is on the roadmap. Class-group VDF epoch seeds, however, are verified on-chain via Wesolowski proof included in L1 batch calldata. Epoch-seed integrity is L1-verifiable today.
- **Class-group discriminant is 256-bit (production upgrade planned).** The current discriminant $\Delta = -p$ where p is the secp256k1 prime provides roughly 128-bit practical security. The planned upgrade to a 2048-bit discriminant will provide a formal reduction to computing the class number of Cl(Δ), matching the hardness of RSA without trusted setup.
- **Ethereum L1 liveness assumed.** Dispute submission, slashing, and withdrawal settlement all depend on Ethereum mainnet functioning correctly during the dispute window.

Non-goals (explicitly out of scope):

- Front-running protection for transactions that do not use the blind path
- Legal or infrastructure coercion resistance (economic penalties do not prevent compelled compliance)
- Cross-domain MEV elimination (atomic cross-rollup reduces but does not eliminate cross-chain extraction)

- **Indefinite censorship resistance** (forced inclusion bounds censorship over a 16-slot window, not permanently)

2. The Problem: Sequencer Centralization

2.1 L2s Depend on a Single Trusted Party

Every major L2 today (Optimism, Arbitrum, Base, zkSync, Starknet) relies on a centralized sequencer to:

- Accept user transactions
- Determine ordering (first-come-first-served, priority fee, or private)
- Produce an ordered batch
- Compute a post-execution state root
- Post the batch and root to Ethereum L1

The sequencer's word is taken on faith. Fraud proofs (Optimistic Rollups) and validity proofs (ZK Rollups) verify *execution correctness*, meaning the state root follows from the transactions. Neither proves that the *sequencer itself* ran correctly. The sequencer can still:

1. **Reorder transactions** to extract MEV without detection
2. **Censor transactions** silently (delay indefinitely)
3. **Equivocate** and show different orderings to different parties
4. **Go offline** with no liveness guarantee
5. **Collude** with block builders to front-run users

The "upgrade key" problem compounds this: most L2 sequencers are upgradeable via a multisig, meaning the security model ultimately rests on a small number of keyholders.

2.2 Shared Sequencers Do Not Solve the Trust Problem

Existing shared sequencer proposals (Espresso, Astria, Radius) distribute trust across multiple nodes via BFT consensus. That reduces single-operator risk but introduces a new trust assumption: the committee. Users still cannot verify what any individual node actually executed. If 2/3 of the committee is honest, the system works, but that is a social guarantee, not a cryptographic one.

2.3 What's Actually Needed

A sequencer that can produce a **cryptographic proof of its own execution**. Not just that the output state follows from the input transactions, but that the sequencer software running on real hardware executed the state transitions exactly as claimed, with no tampering, reordering, or censorship.

---

3. DEOS Tessera: The OS Is the Proof

3.1 Core Insight

DEOS Tessera inverts the traditional approach. Instead of adding a proof layer on top of an untrusted sequencer, DEOS makes the operating system itself the unit of provability.

The sequencer runs inside DEOS. Every syscall it makes (reading from the transaction queue, writing to CAS storage, committing to L1) is logged to a kernel event ring buffer with nanosecond timestamps, Lamport clocks for causal ordering, and cryptographic commitments. The complete execution trace can be replayed deterministically: boot DEOS, feed the same event log, and you get the exact same state root.

3.2 Merkleized OS State

All DEOS kernel state is organized into four sparse Merkle trees. A sparse Merkle tree is a data structure that stores a mapping from keys to values, where the root of the tree is a single hash that commits to all entries. Proving that a value is (or is not) in the tree requires only `log₂(N)` hash operations. For a 32-level tree, that is 32 hashes regardless of how many entries exist.

Tree	Key Formula	What It Tracks
Memory	`addr / 4096`	Virtual memory pages (content hash + flags)
Process	`pid`	Process entries (state, memory root, exit code)
File Descriptors	`(pid << 32) | fd`	Open file descriptors
IPC	`fd`	Pipe endpoints and IPC channels

Each tree root is a Poseidon hash commitment to the full OS state at that moment. State transitions are expressed as Merkle inclusion proofs: prove that leaf `old` was in the tree before the syscall, prove that leaf `new` is in the tree after. The overall OS state root is a commitment combining all four tree roots.

3.3 Syscall-Level Abstraction Boundary

Traditional zkVMs (SP1, RISC Zero, Jolt) prove every CPU instruction. At roughly 100K constraints per instruction, proving a realistic workload requires billions of constraints and can take hours on commodity hardware.

DEOS uses syscalls as the abstraction boundary instead. A syscall is a request from a user program to the operating system kernel: things like "open this file," "create a new process," "allocate memory," or "send data over the network." A syscall like `fork()` or `mmap()` does significant work internally (copying page tables, updating process state), but its *effect* on the Merkle tree is a single leaf transition: one old hash, one new hash, one Merkle path.

The R1CS circuit for a syscall transition is roughly 3K constraints, which is 30x fewer than instruction-level proving for the equivalent amount of work. That makes end-to-end OS proof generation practical on commodity hardware.

---

4. Three-Tier ZK Proof Architecture

DEOS produces three kinds of proofs for each batch, serving different verifier audiences with different cost/speed tradeoffs:

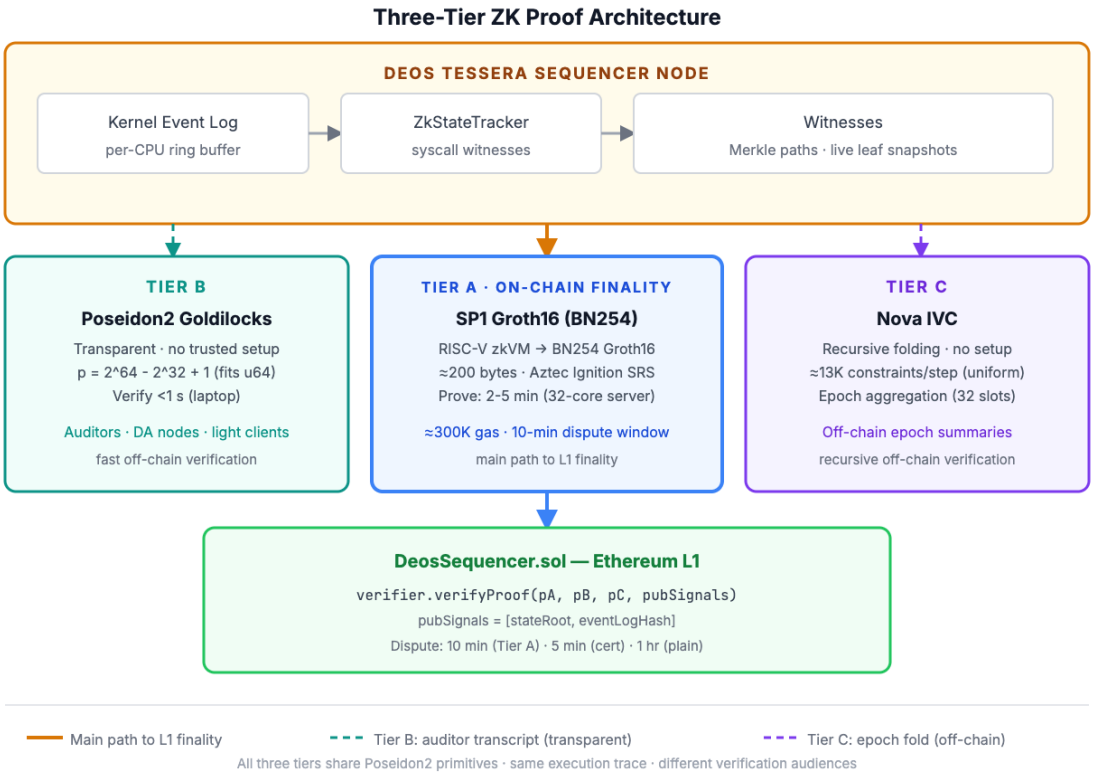


Figure 7: Three-Tier ZK Proof Architecture. Tier A (center, blue) produces the SP1 Groth16 proof submitted to Ethereum L1 for on-chain finality. Tier B (left, teal) provides transparent Poseidon2 Goldilocks transcripts for auditors and light clients. Tier C (right, purple) aggregates epochs via Nova IVC for off-chain recursive verification.

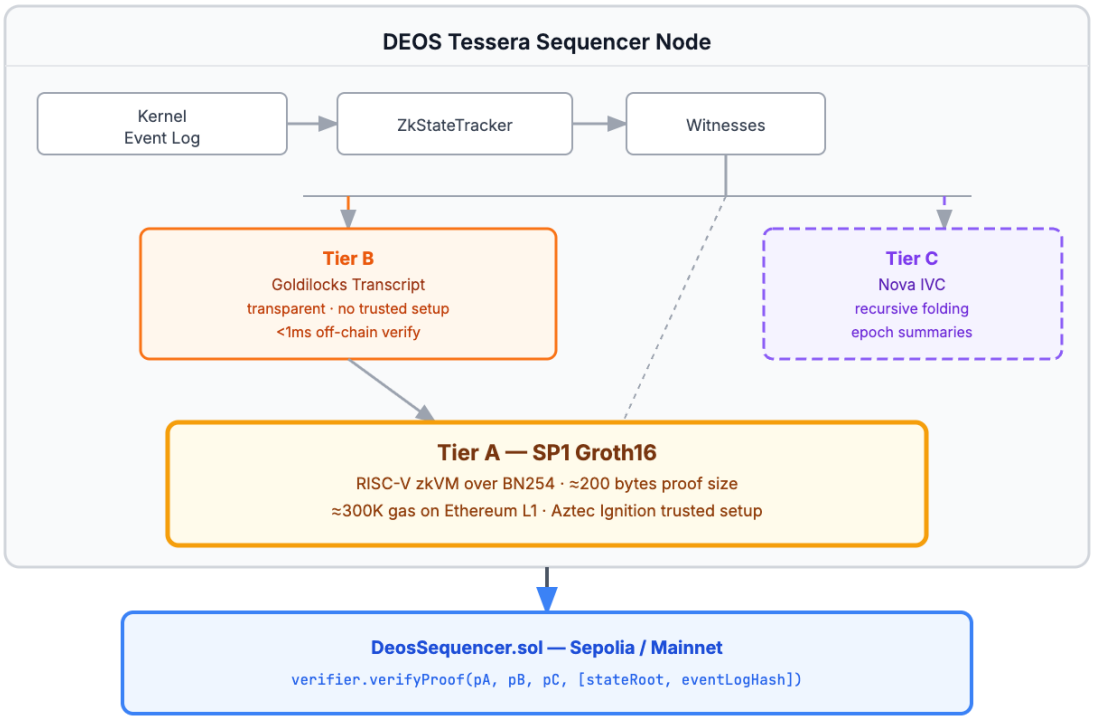


Figure 1: Three-Tier ZK Proof Architecture. Tier A (amber) produces the constant-size Groth16 proof submitted to Ethereum L1. Tier B (orange) enables fast transparent off-chain auditing. Tier C (violet, dashed) produces recursive epoch summaries via Nova IVC.

4.1 Tier A — SP1 Groth16 (On-Chain Finality)

What it is: A RISC-V zkVM proof (via Succinct's SP1) compiled to a BN254 Groth16 proof, verifiable on Ethereum L1 via the `ecAdd`, `ecMul`, and `ecPairing` precompiles.

To understand Groth16: a ZK proof lets a prover convince a verifier that they know some secret information satisfying certain constraints, without revealing the secret. Groth16 is a specific construction that produces a proof of just 3 elliptic curve points (roughly 200 bytes) regardless of the complexity of the computation being proven. The verifier checks a pairing equation over BN254 (an elliptic curve). The check costs roughly 300K gas on Ethereum, equivalent to about 3 ERC-20 token transfers.

What it proves: The sequencer knows an event log that, when replayed through the DEOS kernel state machine, produces the committed `(stateRoot, eventLogHash)` tuple. The circuit runs the DEOS state transition function (Poseidon Merkle tree updates, syscall witness verification) inside an SP1 guest program compiled to RISC-V.

The trusted setup: Groth16 requires a one-time "trusted setup" ceremony to generate cryptographic parameters (a Structured Reference String, or SRS). DEOS uses SP1's universal SRS, which comes from the Aztec Ignition ceremony, a Powers of Tau ceremony with 214 independent participants conducted in 2019. No DEOS-specific ceremony is required. The universal SRS is safe as long as at least one participant in the original ceremony was honest and deleted their toxic waste. A 214-of-214 adversarial collusion requirement, making compromise practically impossible.

Cost: Roughly 300K gas on-chain (EIP-197 pairing check, about $2-10 depending on gas prices). Proven batches get a reduced dispute window of 10 minutes versus 1 hour for unproven batches.

Proving time: Around 2-5 minutes on a 32-core commodity server for a typical DEOS batch (hundreds of syscall steps). Using SP1's remote proving network (Succinct Prover Network), proof generation takes 30-60 seconds. Proof generation runs in parallel with batch production, so the Tier A proof for slot N is ready before slot N+2 needs to be committed.

Public inputs: `[stateRoot, eventLogHash]`, which bind the proof to this exact batch's state and event log.

Key property: The proof is constant-size regardless of batch size. A 1,000-transaction batch has the same 200-byte proof as a 100,000-transaction batch.

```
// On-chain verification in DeosSequencer.sol:
uint[2] memory pubSignals = [uint256(stateRoot), uint256(eventLogHash)];
require(verifier.verifyProof(pA, pB, pC, pubSignals), "ZK proof verification failed");
```

4.2 Tier B - Poseidon2 Goldilocks Transcript (Transparent Auditor Verification)

What it is: A transparent (no trusted setup) proof transcript using Poseidon2 hashes over the Goldilocks field (`p = 2^64 - 2^32 + 1`). Every syscall step is hashed into a running transcript that any party can re-derive from the event log without any cryptographic ceremony.

What it proves: The same execution trace as Tier A, but in a form that can be verified by anyone with a computer (auditors, light clients, DA nodes) without touching Ethereum. Re-running verification takes seconds rather than the minutes required for full replay.

Why Goldilocks: All arithmetic fits in `u64` (no 256-bit bignum), making verification fast on standard hardware. Poseidon2 with `t=12`, `RF=8`, `RP=22`, `α=7` gives roughly 354 constraints per compression and roughly 11,328

constraints per 32-level Merkle path. Verification of a Tier B transcript for a 1,000-step batch takes under 1 second on a laptop.

No trusted setup: Unlike Groth16, the Goldilocks transcript requires no universal reference string. The hash function parameters are public domain. Anyone can verify without trusting any ceremony.

Use case: Data availability layers, rollup clients, indexers, and auditors use Tier B to verify execution locally before accepting the batch. Tier B is also the primary tool for the dispute process: verifiers download the event log and re-derive the Tier B transcript to confirm the state root.

4.3 Tier C - Nova IVC (Epoch Aggregation)

What it is: Incremental Verifiable Computation using the Nova folding scheme. Nova works differently from Groth16: instead of proving the entire computation at once, it *folds* each step into a running "relaxed" R1CS instance. After N steps, you have one compressed proof covering all N steps.

Think of it like a running checksum: instead of storing all N numbers to verify a sum, you maintain a single accumulator. Nova does the same for ZK proofs. Each step's proof is "folded" into the accumulator, and the final accumulator is the epoch proof.

What it proves: That a sequence of N syscall transitions was applied correctly in order, without producing N separate proofs. Nova's core property: the verifier cost is O(1) regardless of the number of folded steps.

Circuit structure: Every step circuit has the same uniform R1CS structure (always 32 Merkle levels, a `has_proof` selector gate makes steps without Merkle witnesses trivially satisfied `0 = 0`). This uniformity is required for Nova because the same circuit function must be applied at every step.

Constraints per step: Roughly 13,256, dominated by Poseidon Merkle path verification (11,328) plus auxiliary gates (roughly 1,928).

Gas cost and timeline: Currently too expensive for on-chain verification (roughly 1M-5M gas per Nova proof). Ethereum's current precompiles support only BN254 arithmetic; Nova works over Pasta curves (Pallas/Vesta), which have no precompile. EIP-2537 (BLS12-381 precompile) is a step toward cheaper curve operations. Once EVM-native Nova verifiers are deployed, or recursive wrapping via Groth16 makes it practical, Tier C becomes the cheapest per-batch option at scale: one proof covers an entire epoch of 32 slots. Today, Tier C is used for off-chain epoch summaries and rollup client verification.

---

5. The Sequencer Service

5.1 What the Sequencer Offers

Ordered Transaction Batches: The DEOS Tessera sequencer accepts transactions via a JSON-RPC API, assembles them into ordered batches per slot, and commits `(slot, chainId, txRoot, stateRoot, eventLogHash)` to `DeosSequencer.sol` on Ethereum L1.

Multi-Chain Support: A single DEOS node can sequence transactions for multiple L2 chains simultaneously. Each batch is tagged with a `chainId`, allowing multiple rollups to share one sequencer operator set. Cross-chain atomic transactions (where two operations on different L2s either both succeed or both fail) become possible because the shared sequencer sees both simultaneously.

Preconfirmation Certificates: Before a batch is finalized on L1 (which takes minutes to hours), users need a fast receipt confirming their transaction will be included. The DEOS sequencer issues preconfirmation certificates, which are signed commitments that a specific transaction at a specific position in a specific slot will be included:

- `slot`: which slot this transaction will appear in
- `position`: exact position within the slot's ordered set
- `txHash`: commitment to the transaction content
- `sequencer_sig`: Ed25519 signature over the certificate

These certificates provide sub-second "soft finality." If the sequencer later fails to include the transaction as promised, its stake is slashable. The economic penalty makes preconfirmations trustworthy even before cryptographic L1 finality.

Committee Ordering Certificates: A set of DEOS operators can co-sign an `orderHash` (a hash of the deterministic transaction ordering) via threshold ECDSA. When K-of-N operators agree on the same ordering, the resulting certificate is posted on-chain. Committee-certified batches get a 5-minute dispute window (the shortest available) because any conflicting ordering by the same committee is immediately detectable and slashable on-chain.

EIP-4844 Blob DA: Batch transaction data can be published via EIP-4844 blobs, a specialized data storage format in Ethereum that stores up to 128KB of data per blob at roughly 10x lower cost than regular calldata, but only for approximately 18 days. This is sufficient for the dispute window while dramatically reducing DA costs. The `blobHash` (versioned hash of the blob content) is committed on-chain, cryptographically binding the batch to its data.

Event Log URI: Every batch commits an `eventLogUri` pointing to the DEOS event log for that batch in content-addressed storage. Anyone can download the log, boot DEOS, and replay the exact execution to independently verify the state root. This is what makes the dispute mechanism work.

5.2 Dispute Resolution Ladder

DEOS batches are finalized via an optimistic dispute window. "Optimistic" means we assume the batch is correct and finalize it after a waiting period unless someone proves otherwise. The three paths to finality, in order of speed:

Batch Type	Dispute Window	How to Achieve
Plain batch	1 hour	Just commit the batch
Committee-certified	5 minutes	K-of-N operators co-sign orderHash
ZK-proven (Tier A)	10 minutes	Submit Groth16 proof with batch
Both certified + proven	5 minutes	Both mechanisms applied

Disputing a batch: Any party can call `disputeBatch(batchId, actualStateRoot)` during the window. They download the event log, replay it in DEOS, and if they compute a different state root, they post the claim on-chain and receive 50% of the operator's stake as a bounty. Since replay is deterministic, disputes are unambiguous. Both parties running the same log must reach the same result.

Equivocation slashing (100% stake): If an operator commits two different batches for the same `(slot, chainId)`, anyone can call `proveEquivocation(batchId1, batchId2)` for an instant full stake slash. Equivocation is the harshest offense because it enables double-spend attacks: convincing two different parties that different transactions were finalized.

Ordering equivocation (50% stake): If two committee-certified batches for the same `(slot, chainId)` have different `orderHash` values, the ordering committee signed conflicting statements. Anyone can call `proveOrderingEquivocation()` to slash 50% of the responsible operator's stake.

Liveness slashing (10% stake): If a designated operator misses their assigned slot and no batch is committed within 4 slots, anyone can call `reportMissedSlot()` to slash 10% of their stake. This creates economic pressure for uptime without requiring a separate liveness protocol.

5.3 Forced Inclusion (Censorship Resistance)

A transaction is "censored" when the sequencer refuses to include it. The user's transaction sits pending indefinitely while the sequencer processes other transactions around it.

Users who believe their transaction is being censored can submit it directly to L1 via `submitForcedTx(txHash, chainId)`. The transaction enters an on-chain queue with a `submitSlot` timestamp.

If the transaction is not provably included in a batch within 16 slots, anyone can call `slashForCensorship(queueIndex)` to slash the responsible operator 25% of their stake. Inclusion must be proven via Merkle proof against the batch's `txRoot`. An operator cannot claim inclusion without proving it on-chain. Censorship has a hard time limit with automatic financial consequences.

5.4 Withdrawal Bridge

The sequencer contract maintains an on-chain bridge:

1. **Deposit:** Users send ETH to `deposit(chainId)` to fund the bridge for a specific L2
2. **Withdraw:** After a batch containing a withdrawal is finalized, users call `proveWithdrawal(batchId, recipient, amount, nonce, merkleProof)` to claim their funds

Withdrawals require a Merkle inclusion proof against the batch's finalized `stateRoot`. This ties L1 withdrawals directly to the verified execution state. Users cannot withdraw funds that the sequencer didn't explicitly commit in a proven state transition. The `nonce` prevents replaying the same withdrawal twice.

5.5 Stake-Weighted Slot Assignment

Operators register by staking ETH (minimum 1 ETH). Slot assignment is deterministic and stake-weighted. An operator with 10 ETH staked wins roughly 10x more slots than one with 1 ETH staked:

```
epoch   = slot / 32
seed    = keccak256(epoch || slot)
point   = uint256(seed) % totalActiveStake
winner  = first operator where cumulative stake exceeds point
```

This works similarly to Ethereum's own validator selection by effective balance. The seed is entirely determined by publicly known values (epoch and slot number), making assignment predictable and verifiable by anyone. Deregistering operators (7-day unbonding period) are excluded from selection, which prevents last-minute stake removal to avoid penalties.

5a. Kernel-Witnessed Admission (KWA)

5a.1 The Mempool Problem

Every shared sequencer faces a structural accountability gap. Proofs of execution correctness begin at the point where transactions enter the sequencer's mempool, but the mempool itself is user-space software. The window between a transaction arriving at the host and that transaction appearing in the mempool is unwitnessed. Selective acceptance, hidden fast lanes, and silent censorship all live in that window, invisible to any proof system that starts at mempool admission.

This is not a failure of cryptography. It is a boundary problem: the proof surface begins too late.

5a.2 The KWA Mechanism

DEOS Tessera introduces Kernel-Witnessed Admission (KWA): a protocol mechanism that moves the accountable mempool boundary from user-space admission to kernel-witnessed transaction ingest.

When a transaction arrives at the sequencer's HTTP endpoint, dhttpd processes the request and stores the transaction blob in the Content-Addressed Store via the `cas_index_put` system call. This syscall is handled by the DEOS kernel. At the point of handling, before the syscall returns to user space, the kernel performs two additional operations atomically under the same CAS lock:

1. **Receipt mirror**: writes `seq-rx-N → tx_hash` where N is the same sequence number used for the user-facing `seq-tx-N` entry.
2. **SEQT event**: emits a `0x53455154` ("SEQT") custom event into the per-CPU kernel event ring, carrying `(seq_num, tx_hash)` with a hardware TSC timestamp.

Both actions complete before `cas_index_put` returns. The sequencer process regains control only after the kernel has already created an immutable record of the transaction's arrival.

Canonical ingest point: A transaction is KWA-seen when the kernel commits `seq-tx-N → hash` and emits the corresponding SEQT event. This is an unambiguous, machine-checkable state transition. There is no race window between arrival and witnessing.

5a.3 Slot Attribution

SEQT events carry hardware TSC timestamps and are ordered structurally within the kernel event ring. A transaction belongs to slot S if its SEQT event appears in the event ring between the seal event of slot S−1 and the seal event of slot S. This ordering comes from the event ring directly. No wall-clock arithmetic, no boundary ambiguity.

5a.4 The Five-Root Batch Commitment

Each sealed batch now carries five cryptographic commitments:

Root	Content	Purpose
`txRoot`	Merkle root of included tx hashes	Proves ordered execution set
`stateRoot`	Poseidon hash of execution trace	Proves honest execution
`eventLogHash`	keccak256 of raw event log	Binds batch to kernel event ring
`seenRoot`	Merkle root of KWA-seen set	Proves kernel-witnessed admission
`decisionRoot`	Merkle root of per-tx decisions	Proves declared fate of every seen tx

`seenRoot` is computed as:

```
leaf_i = keccak256(tx_hash ‖ seq_num_be8 ‖ slot_be8)
seenRoot = MerkleRoot(leaf_0, leaf_1, …)
```

Binding `seq_num` and `slot` into each leaf means every entry has a unique, positioned identity in the global sequence. That enables individual membership proofs without recomputing the full list.

5a.5 Exclusion Taxonomy

Every transaction in the KWA seen_set receives exactly one decision record, committed into `decisionRoot` at batch-seal time. The closed set of valid reason codes:

Code	Name	Condition
0x00	INCLUDED	Present in `txRoot`
0x01	INVALID_ENCODING	Not parseable as a valid EIP-2718 tx envelope
0x02	INVALID_SIGNATURE	ECDSA verification failed
0x03	BAD_NONCE	Nonce below or excessively above account nonce
0x04	INSUFFICIENT_BALANCE	Sender cannot cover gas × price + value
0x05	DUPLICATE	Same tx hash seen earlier in this or a prior batch
0x06	EXPIRED	Deadline field exceeded slot timestamp
0x07	CAPACITY_DEFERRED	Slot full; valid tx moved to next slot's seen_set
0x08	POLICY_REJECTED	Operator policy (requires `policy_rule_hash`, see below)

Each decision leaf is:

```
leaf_i = keccak256(tx_hash ‖ reason_code ‖ policy_rule_hash)
decisionRoot = MerkleRoot(leaf_0, leaf_1, …)
```

`POLICY_REJECTED` constraint: This code is only valid when accompanied by a non-zero `policy_rule_hash`, the keccak256 of a policy document pre-registered in the on-chain Policy Rule Registry (§5a.10) with a mandatory 2-hour delay before it becomes effective. An operator who wants to exclude certain transaction types must publish their policy rule on-chain and wait for the activation window before using this code. An unclassified `POLICY_REJECTED` (zero hash, or a hash not registered by the batch's operator) reverts on-chain and the refutation is rejected. Post-hoc policy justifications are inadmissible: the rule hash must be bound in the `decisionRoot` leaf at batch-seal time and must have been effective at that batch's timestamp.

5a.6 Censorship Condition

A verifier applies the following rule to every SEQT event extracted from the event log:

```
for each tx_hash in seenSet(slot):
    assert tx_hash ∈ txRoot
        OR ∃ record in decisionRoot: record.tx_hash == tx_hash
                                   ∧ record.reason_code ∈ VALID_CODES
                                   ∧ (record.reason_code ≠ POLICY_REJECTED
                                      OR record.policy_rule_hash is registered on-chain)
```

A KWA-seen transaction that satisfies neither condition is censorship evidence, verifiable on-chain from the committed roots alone, without replaying the event log.

The sequencer stores the full decision log under `seq-decision-{slot}` in its Content-Addressed Store. Any observer can retrieve it without access to the event ring.

5a.7 Trust Boundary

KWA does not claim to solve network-level fairness. Upstream infrastructure (reverse proxies, load balancers, NIC firmware, routing) remains outside the kernel boundary and therefore outside the KWA proof surface.

The precise claim is:

> **Any transaction that reaches the DEOS kernel's CAS ingest path is KWA-witnessed before the sequencer has discretion. Exclusion after that point requires a declared, verifiable reason.**

This moves the manipulation surface from application-layer mempool admission to kernel-witnessed ingest, a substantially lower and harder-to-exploit boundary than any conventional shared sequencer design.

5a.8 KWA Dispute Protocol

KWA censorship is not merely auditor-detectable. It is L1-disputable and operator-slashable through a three-step challenge-response protocol built into the `DeosSequencer` contract.

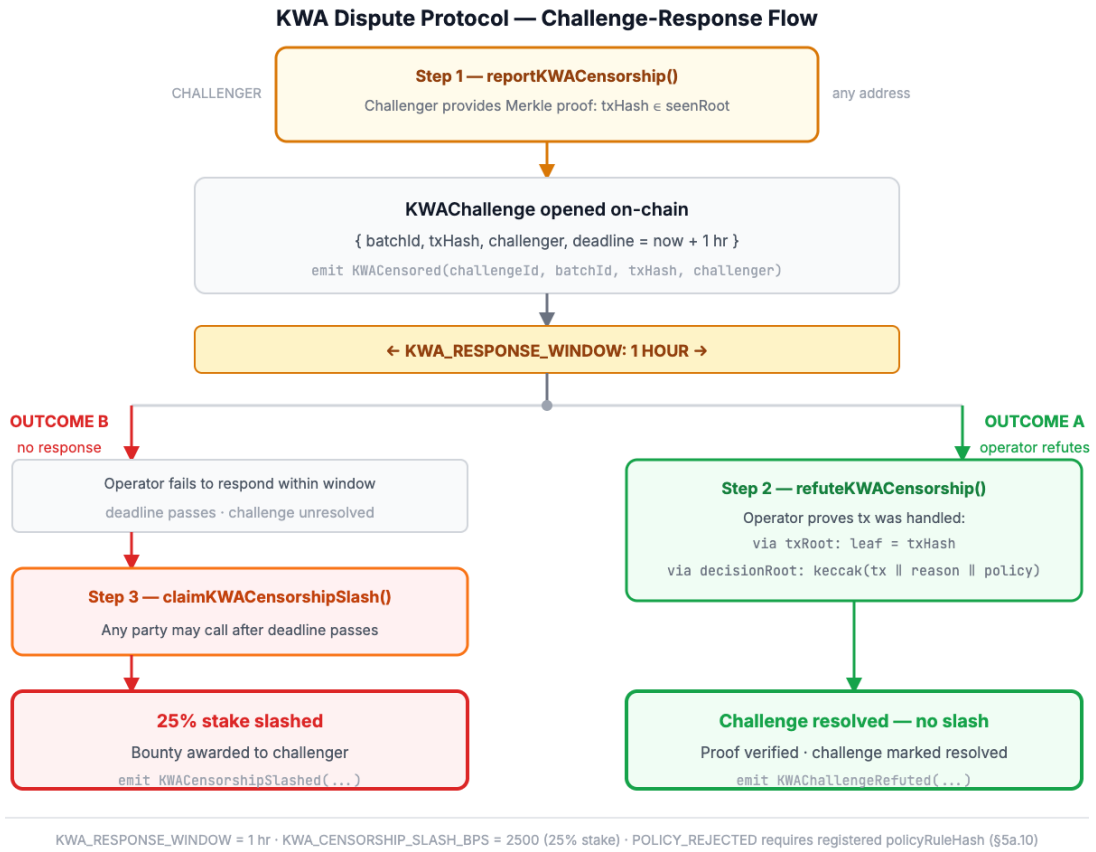


Figure 6: KWA Dispute Protocol Challenge-Response Flow. Step 1 — challenger submits a Merkle inclusion proof that txHash ∈ seenRoot. Within the 1-hour response window the operator either refutes (Outcome A: proof accepted, no slash) or fails to respond (Outcome B: any party claims a 25% stake slash awarded to the challenger).

Formal censorship predicate. A transaction `tx` is KWA-censored in batch `B` iff:

```
tx_hash ∈ leaves(B.seenRoot)
∧ tx_hash ∉ leaves(B.txRoot)
∧ ¬∃ (r, p) : keccak256(tx_hash ‖ r ‖ p) ∈ leaves(B.decisionRoot)
```

The first clause is a positive Merkle inclusion proof. The second and third require absence, handled via challenge-response rather than on-chain non-membership proofs.

Step 1: **`reportKWACensorship(batchId, txHash, seqNum, seenProof)`**

Challenger provides a Merkle inclusion proof that `txHash` is a leaf in `batch.seenRoot`:

```
leaf = keccak256(txHash ‖ seqNum_be8 ‖ slot_be8)
```

The contract verifies the proof, opens a `KWAChallenge` record, and emits `KWACensored`. The operator has `KWA_RESPONSE_WINDOW` (1 hour) to respond.

Step 2: **`refuteKWACensorship(challengeId, inTxRoot, reasonCode, policyRuleHash, proof)`**

Operator proves the tx was properly handled, either included in `txRoot` or recorded in `decisionRoot`:

- **txRoot refutation**: Merkle inclusion proof with leaf = `txHash`
- **decisionRoot refutation**: inclusion proof with the 65-byte leaf format: `leaf = keccak256(txHash ‖ reasonCode(1) ‖ policyRuleHash(32))` matching the `ExclusionRecord` format from Section 5a.5

If the proof verifies, the challenge resolves with no slash.

Step 3: **`claimKWACensorshipSlash(challengeId)`**

If the operator fails to refute within the window, any party claims a 25% slash of the operator's stake, awarded to the challenger.

Monotonic KWA commitment. Once an operator commits a batch with non-zero `seenRoot` for a given chain, that chain is KWA-active. The contract records the activation slot. Subsequent batches omitting `seenRoot` are reportable via `reportKWADeactivation()`, triggering a 10% slash. KWA is a commitment, not an optional annotation. Operators cannot selectively disable it on inconvenient slots.

Liveness invariant. A valid batch where every KWA-seen tx is either included or has a justified exclusion record cannot be successfully challenged. Only genuine omissions (txs in `seenRoot` absent from both `txRoot` and `decisionRoot`) result in slash.

5a.9 Relationship to Forced Inclusion

KWA and L1 forced inclusion are complementary, not competing:

- **Forced inclusion** resolves censorship after the fact by allowing users to submit transactions directly to L1.
- **KWA** makes censorship *disputable* within the slot, at the kernel event boundary, with on-chain slashing consequences.

KWA changes the accountability model from "user notices after several blocks" to "operator is slashable from the next batch commitment."

5a.10 Policy Rule Registry

The `POLICY_REJECTED` exclusion code is a powerful operator instrument. It allows an operator to exclude an entire class of transactions from a batch without individual justification. Without a binding commitment mechanism, this power would be abusable: an operator could construct a policy after the fact that retroactively justifies any exclusion. The Policy Rule Registry closes this loophole.

Pre-registration requirement. Before an operator may use `POLICY_REJECTED` for a given rule hash, they must call `registerPolicyRule(ruleHash)` on the `DeosSequencer` contract. The transaction is recorded with:

```
policyRules[ruleHash] = PolicyRule {
    registeredAt: block.timestamp,
    effectiveFrom: block.timestamp + POLICY_DELAY,
    registeredBy:  msg.sender,
    revokedAt:     0
}
```

`POLICY_DELAY` is 2 hours (7200 seconds). A policy rule is not usable until `effectiveFrom` has passed, preventing same-block registration and use.

Validity conditions. `refuteKWACensorship()` calls `isPolicyEffective(ruleHash, operator, batchTimestamp)`, which returns `true` only when all four conditions hold:

Condition	Requirement
Non-zero hash	`ruleHash != bytes32(0)`
Registered	`rule.registeredAt != 0`
Operator binding	`rule.registeredBy == operator`
Temporal window	`rule.effectiveFrom ≤ batchTimestamp < rule.revokedAt` (or not yet revoked)

A zero `ruleHash` always fails. The contract explicitly rejects `POLICY_REJECTED` refutations with zero hash regardless of other conditions.

Prospective-only revocation. An operator may call `revokePolicyRule(ruleHash)` to deactivate a rule. Revocation sets `revokedAt = block.timestamp` and takes effect immediately for future batches. Critically, revocation is prospective only: `isPolicyEffective` returns `true` for any `batchTimestamp < revokedAt`, so past batches that legitimately used the rule remain valid. An operator cannot revoke a rule to retroactively invalidate their own prior refutations.

Why this prevents policy laundering. The 2-hour mandatory delay means the policy must be public and observable by participants *before* it can be applied. Any participant who disagrees with an operator's stated policy can stop submitting the affected transaction types, challenge an existing batch, or switch sequencers, all within the window before the policy becomes effective. The operator binding condition (`registeredBy == operator`) ensures that a policy registered by one operator cannot be used by another; policies cannot be borrowed or transferred.

Integration with `refuteKWACensorship()`. When an operator submits a `POLICY_REJECTED` refutation with `inTxRoot = false`:

```
require(policyRuleHash != bytes32(0),
    "POLICY_REJECTED requires non-zero policyRuleHash");
require(isPolicyEffective(policyRuleHash, batch.operator, batch.timestamp),
    "Policy rule was not registered and effective at batch time");
// verify decisionRoot Merkle inclusion proof
```

Both the non-zero hash check and the `isPolicyEffective` check must pass before the Merkle proof is even evaluated. A failed `isPolicyEffective` check produces a slashable outcome: the operator's refutation is rejected and `claimKWACensorshipSlash()` becomes callable.

---

6. VDF Blind Sequencing: Ordering Fairness by Cryptography

6.1 The Intra-Slot MEV Problem

The three-tier ZK proof system guarantees *execution correctness*: given a fixed ordered set of transactions, DEOS can prove it executed them faithfully. But it does not (yet) constrain *how* the sequencer chose to order those transactions within a slot.

This intra-slot ordering freedom is the root of **MEV** (Maximal Extractable Value). A sequencer that can see transaction contents before committing to their order can:

- **Front-run:** Insert its own transaction before a large trade to profit from price impact
- **Sandwich attack:** Bracket a user's swap with a buy-before / sell-after, extracting the price impact as profit
- **Selective censorship:** Ignore low-fee transactions that would reduce the value of a profitable bundle
- **Content-based reordering:** Choose the ordering that maximizes fee extraction at the expense of arrival fairness

All of these attacks require the sequencer to *see* transaction contents before *committing* to their order. If the sequencer commits to an ordering while blind to the contents, content-based manipulation becomes structurally impossible. This guarantee applies to transactions using the blind path; plain-path transactions remain at the operator's discretion, subject to economic slashing.

6.2 Verifiable Delay Functions (VDFs)

A **Verifiable Delay Function** (VDF) is a function that:

1. **Takes sequential time**: cannot be parallelized; computing output N requires first computing output N−1
2. **Produces a verifiable proof**: anyone can verify the result in O(1) time without redoing the work
3. **Has predictable timing**: the delay is measurable in wall-clock time

The intuition: a VDF is a clock that cannot be skipped. Just as you cannot know tomorrow's hash until you compute today's hash first, a VDF chain embeds *time* directly into its output.

DEOS VDF construction (Poseidon2 sequential hash chain):

```
state[0]   = epoch_seed
state[t+1] = Poseidon2(state[t], tick_input[t])

tick_input[t] = Poseidon2(t.to_le_bytes(), window_acc[t])
window_acc[t] = rolling hash of tx commitments arriving in window t
```

Key property: To compute `state[100]`, you must first compute `state[1]` through `state[99]`. No amount of parallel compute can skip steps. The chain's output is a cryptographic proof that *time has passed*.

6.3 Blind Sequencing Protocol

If users encrypt their transactions under a key that cannot be revealed until *after* the sequencer has committed to an ordering, the sequencer must order ciphertexts it cannot read. Content-based manipulation becomes structurally impossible.

DEOS Tessera implements this as a three-phase protocol:

Phase 1, Seal: The user generates a random 32-byte key, encrypts the transaction, and computes a commitment.

```
key         := VdfClient::random_key()          // kernel-supplied CSPRNG
ciphertext  := xor_keystream(key, plaintext)    // Poseidon2 XOR stream cipher
commitment  := BLAKE3(plaintext)                // posted publicly with ciphertext
```

Phase 2, Submit: The user sends `(commitment, ciphertext)` via `SYS_VDF_SUBMIT_ENCRYPTED`. The kernel:

- Assigns `arrival_tick` from the current VDF clock (irrevocable)
- Sets `reveal_tick = arrival_tick + MIN_REVEAL_DELAY` (50 ticks = 5 ms)
- Writes a `VdfBlindTxArrived` event to the kernel event ring (permanent ordering record)
- Returns a `BlindTxReceipt` to the user

Phase 3, Reveal: After `reveal_tick` is reached (5 ms later), the user posts their decryption key via `SYS_VDF_REVEAL_TX`. The kernel:

- Decrypts: `plaintext := xor_keystream(key, ciphertext)`
- Verifies: `BLAKE3(plaintext) == commitment` (rejects substitutions)
- Writes a `VdfTxRevealed` event to the event ring
- Returns the plaintext transaction bytes

The sequencer calls `SYS_VDF_DRAIN_READY` at slot boundaries to collect all revealed transactions, then inserts them into the execution queue in arrival_tick order (the order ciphertexts arrived, not the order keys were revealed). Content-based reordering is impossible.

6.4 Hybrid VDF Chain Architecture

The DEOS Tessera VDF system is a two-layer hybrid. The outer layer provides formal sequential hardness via a class-group VDF at epoch boundaries (every 32 slots, roughly 6.4 minutes). The inner layer provides ZK-efficient high-frequency ordering via a Poseidon2 hash chain seeded from the class-group output.

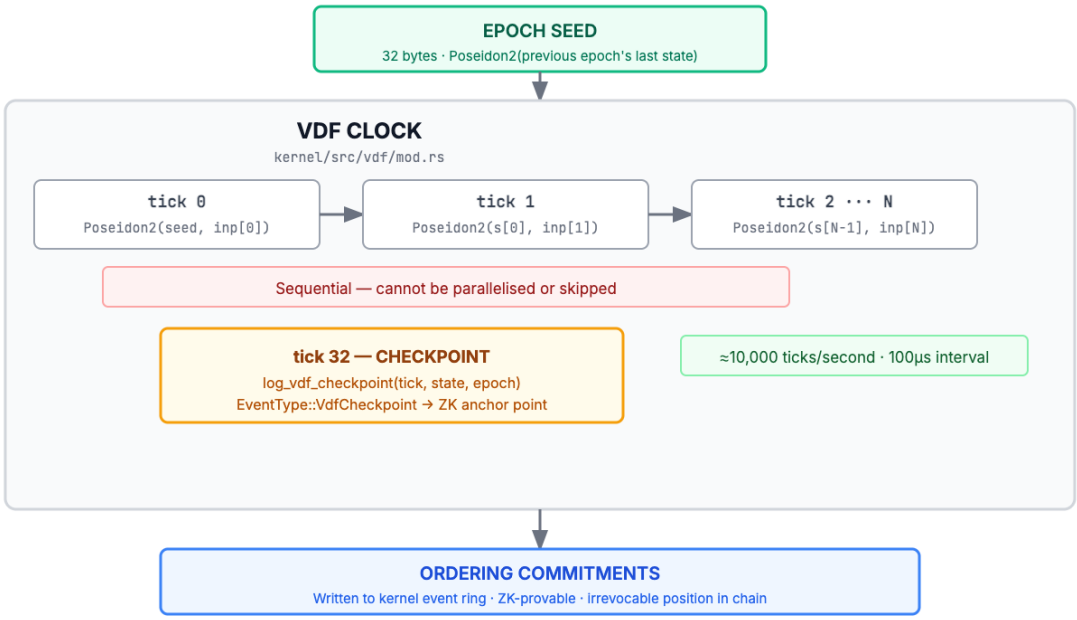


Figure 2: VDF Clock Architecture. Each tick is a sequential Poseidon2 hash; no step can be skipped. Every transaction's arrival_tick is an irrevocable position in this chain.

Runtime constants:

```
TICK_INTERVAL        = 100 µs   (10,000 ticks per second)
MIN_REVEAL_DELAY     =  50 ticks  ≈ 5 ms  (blind window)
CHECKPOINT_INTERVAL  =  32 ticks  ≈ 3.2 ms (ZK anchor points)
EPOCH_SLOTS          =  32 slots  ≈ 6.4 min (class-group VDF boundary)
VDF_T                =  2^20 squarings ≈ 15-30 s delay
DISCRIMINANT         =  -p secp256k1  (256-bit, no trusted setup)
```

6.5 Blind Sequencing Protocol Flow

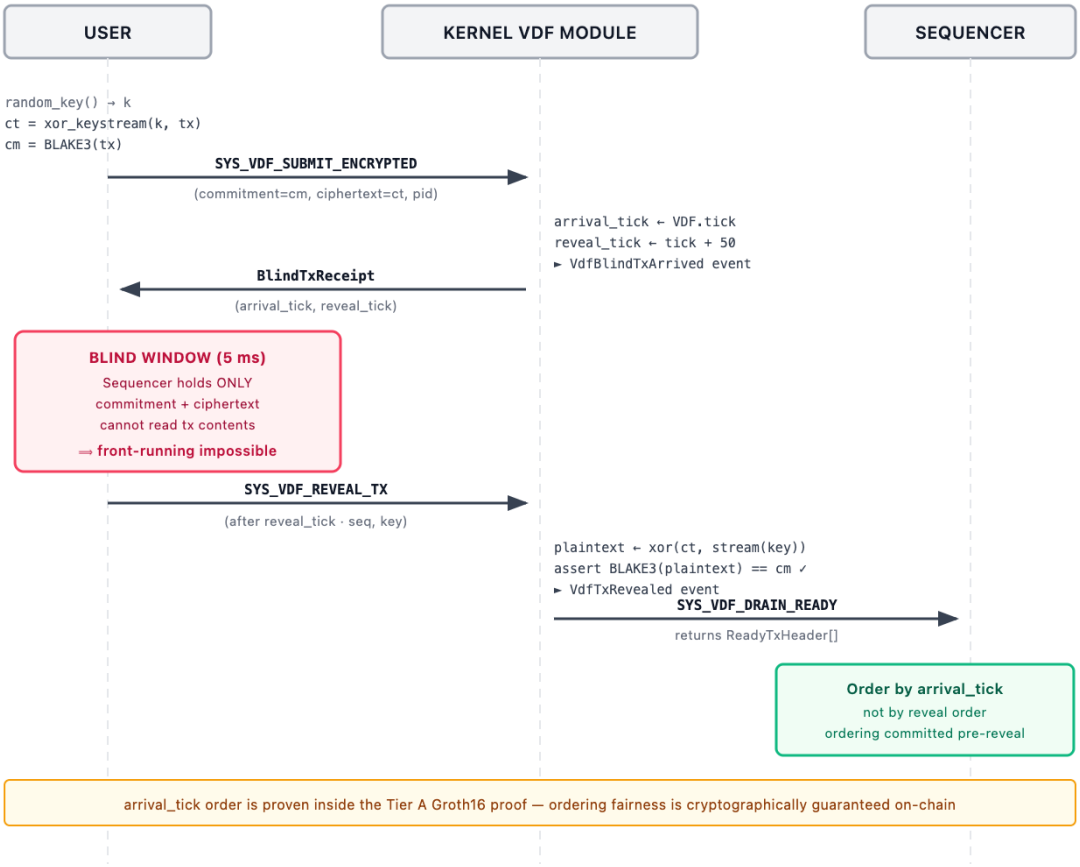


Figure 3: Blind Sequencing Protocol Flow. The sequencer orders encrypted ciphertexts, committing to an arrival_tick while blind to transaction contents. After the 5ms blind window, users reveal their keys; the sequencer executes in arrival_tick order. This ordering is proven inside the Tier A Groth16 proof.

6.6 Slot Timeline with VDF Ticks

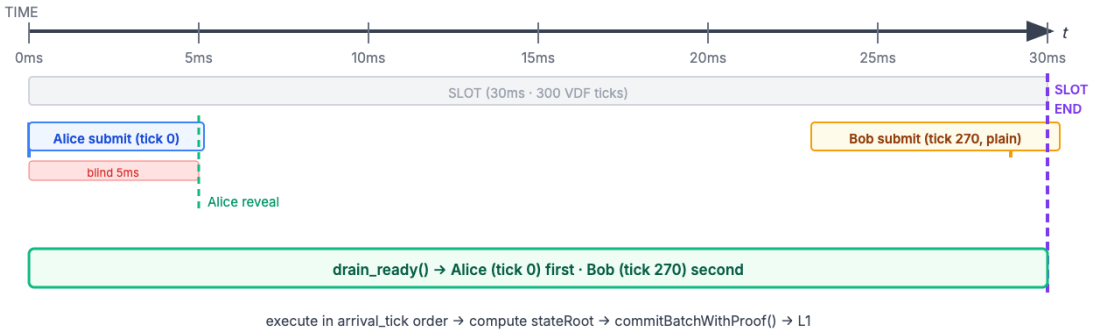


Figure 4: VDF Slot Timeline. Alice submits blind at t=0 (arrival_tick=0); Bob submits plain at t≈27ms (arrival_tick=270). At slot boundary, drain_ready() returns in arrival_tick order — Alice first, regardless of reveal timing.

6.7 ZK Integration: VDF Chain Inside SP1 Circuit

The DEOS Tessera VDF chain is built on Poseidon2, the same hash function used throughout the three-tier ZK system. This is not incidental. It means the VDF state chain can be verified *inside* the existing SP1 guest program at negligible extra cost, extending the Groth16 proof to cover ordering fairness in addition to execution correctness.

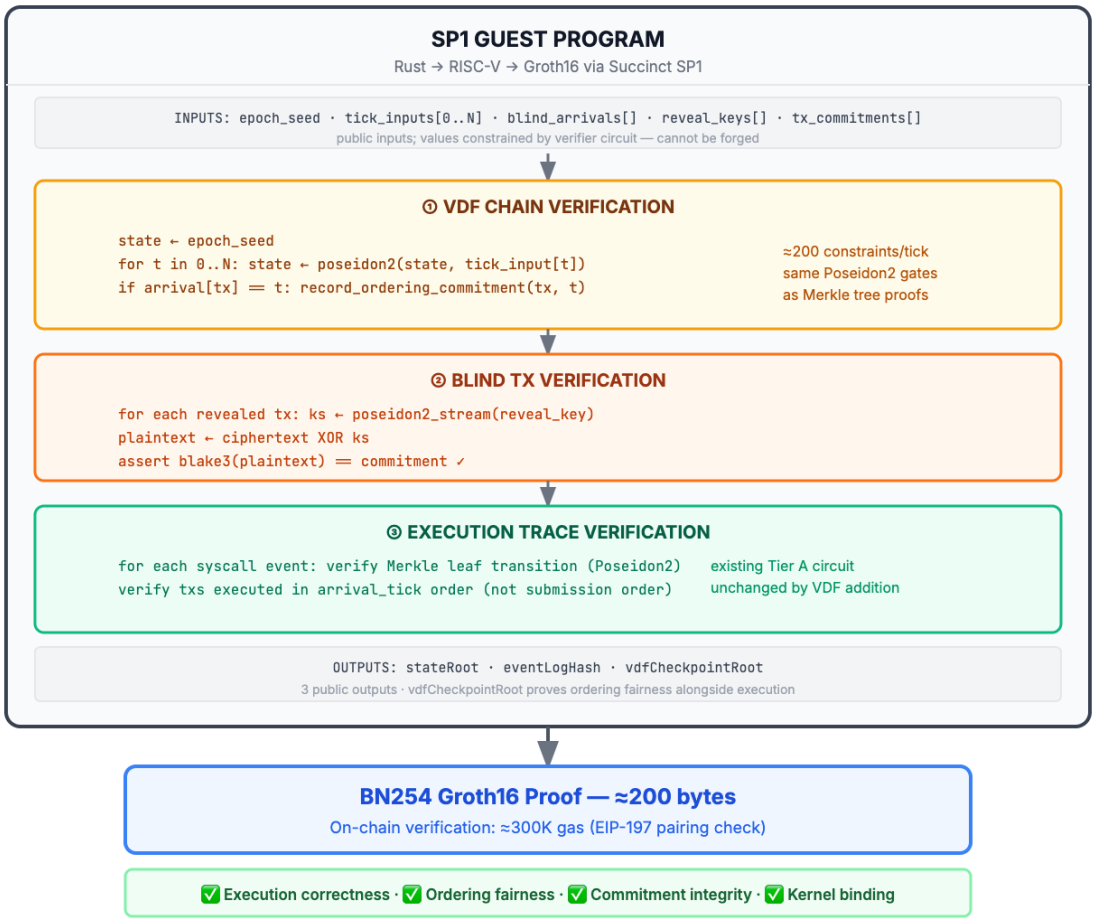


Figure 5: SP1 Guest Program Circuit. Three verification layers run inside a single RISC-V guest compiled to Groth16. VDF chain (①) and blind TX verification (②) add negligible overhead by reusing the same Poseidon2 gates as Merkle tree proofs (③).

Unique positioning: No other shared sequencer combines ordering fairness and execution correctness in a single on-chain verifiable proof. Espresso and Astria use BFT consensus for ordering (committee trust assumption, no execution proof). ZK rollups prove execution but not ordering. DEOS Tessera proves both in one roughly 200-byte Groth16 proof verifiable for roughly 300K gas.

6.8 Hybrid VDF Design: Class Groups + Poseidon2

Why Not RSA?

Earlier VDF proposals (RSA sequential squaring: `x^{2^t} mod N`) require a trusted setup ceremony to generate `N = p × q`. Whoever generates the modulus retains the factors and can compute any future VDF output instantly, defeating the time guarantee entirely. Class groups have no trapdoor: the discriminant `Δ` is public, and there is no known efficient algorithm for computing the group order of `Cl(Δ)`. No ceremony, no trusted party.

Why Class Groups for Epoch Seeds?

Class-group VDFs (Wesolowski 2018) provide formal sequential hardness: computing `g^{2^T}` in `Cl(Δ)` requires at least `T` sequential squarings unless the Strong RSA problem can be solved in `Cl(Δ)`, a hardness assumption at least as strong as integer factoring. The Wesolowski proof allows efficient verification: given `(g, y, π, ℓ)`, check `y == π^ℓ · g^r` in O(log ℓ), roughly 128 group operations. No re-running the computation.

DEOS discriminant: `Δ = −p_secp256k1` where `p = 2^256 − 2^32 − 977`. This is the secp256k1 base field prime, a "nothing up my sleeve" constant. Since `p ≡ 3 mod 4`, we have `Δ ≡ 1 mod 4`, making it a valid fundamental discriminant. At 256 bits it provides roughly 128-bit practical security; the production upgrade to a 2048-bit discriminant will match RSA-2048 hardness without trusted setup.

Why Poseidon2 for Intra-Epoch Ticks?

Class-group squaring costs approximately 30,000-50,000 R1CS constraints per step (large-field arithmetic over 256-bit integers). At 10,000 ticks/second and a 2,000-step batch, that is 60-100 billion constraints, completely impractical inside an SP1 circuit.

Poseidon2 costs roughly 200 constraints per step. A 2,000-step batch adds only roughly 400K constraints on top of the existing Merkle tree proofs. The tick chain is ZK-provable at negligible overhead. This is the fundamental reason for the hybrid: class groups provide formal hardness at epoch boundaries where ZK-proving is not required (Wesolowski proof goes to L1 directly), and Poseidon2 handles the high-frequency inner loop where ZK-proving is essential.

Implemented Construction (libdeos/src/class_group_vdf.rs)

```
Parameters:
  Δ = −p_secp256k1                              (256-bit discriminant, no trusted setup)
  T = 2^20 = 1,048,576 squarings               (≈15–30 s delay on commodity hardware)
  ℓ = H(serialize(g), T)                        (Fiat-Shamir prime — avoids two-pass)
  g = hash_to_form(prev_state_root)             (input form derived from last state root)

Evaluate+Prove (single-pass Wesolowski Algorithm 2):
  y  = g^{2^T}                                  (VDF output)
  π  = g^{⌊2^T/ℓ⌋}                              (quotient element, computed inline)
  proof_blob = serialize(g) ‖ ℓ.to_le_bytes() ‖ serialize(π)   (264 bytes)
  epoch_seed = blake3(serialize(y))             (32-byte epoch seed)

Verify (on-chain, O(log ℓ) ≈ 128 group operations):
  r   = 2^T mod ℓ                               (computed by verifier)
  lhs = compose(pow(π, ℓ), pow(g, r))
  assert lhs == y
```

The SQFCOMP squaring algorithm (Shanks) is used for class-group squaring: `λ = c · b⁻¹ mod a`, then `A = a²`, `B = b − 2aλ`, followed by Gauss reduction to canonical form `(−a < b ≤ a ≤ c)`. This is the same algorithm used in Chia's VDF competition entries and the reference implementations of Wesolowski/Pietrzak.

Epoch Seed Submission

Every 32 slots (roughly 6.4 minutes), the sequencer:

1. Calls `libdeos::class_group_vdf::compute_epoch_seed(prev_state_root)` to run T squarings inline (roughly 15-30 s, well within the 6.4-minute epoch window)
2. Calls `VdfClient::vdf_submit_epoch_seed(epoch, &seed, &proof_blob)` → `SYS_VDF_SUBMIT_EPOCH_SEED` (syscall 735)
3. Kernel updates the Poseidon2 tick chain's epoch seed via `crate::vdf::start_epoch(seed, epoch)`
4. `proof_blob` is included in the L1 batch calldata; the `DeosSequencer.sol` Wesolowski verifier checks it on-chain

Formal sequential hardness guarantee (epoch boundaries): Any party wishing to predict or pre-compute the epoch seed for epoch `E` must compute `g^{2^T}` in `Cl(Δ)`. By the Strong RSA assumption in class groups, this requires at least `T` sequential squarings, approximately 15-30 seconds of real computation. The Wesolowski proof confirms this work was done without re-running it.

Intra-epoch practical hardness: The Poseidon2 tick chain does not carry a classical sequential hardness proof. It relies on the practical assumption that no algorithm computes `state[t]` from `state[0]` faster than `t` sequential Poseidon2 evaluations. This is strongly believed given Poseidon2's algebraic structure and the absence of any known shortcut, but it is not a formal hardness reduction. Epoch seeds provide the formal anchor; ticks provide ZK-efficient ordering within that anchor.

6.9 Prior Art

VDF + blind sequencing combines two independent research threads, both predating any blockchain sequencer project:

Work	Year	Contribution
Rivest, Shamir, Wagner -- "Time-Lock Puzzles and Timed-Release Crypto"	1996	First formalized timelock construction (RSA squaring)
Boneh, Bonneau, Bünz, Fisch -- "Verifiable Delay Functions"	2018	Formal VDF definition, security proofs, candidate constructions
Justin Drake (ethresear.ch) -- "VDF-based RNG for Ethereum"	2018	First proposed VDF application to Ethereum ordering fairness
Shutter Network	2021	Threshold-encryption mempool for front-running protection
DEOS Tessera VDF module	2026	First class-group+Poseidon2 hybrid VDF sequencer; Wesolowski on-chain epoch seed verification without trusted setup

The VDF blind sequencing concept is not proprietary to any single project. DEOS's novel contribution is the hybrid VDF construction: class-group Wesolowski proofs for formal sequential hardness at epoch boundaries (on-chain verifiable, no trusted setup) combined with ZK-provable Poseidon2 tick ordering within epochs, a combination not present in any prior sequencer or VDF work.

6.10 Security Guarantees of VDF Blind Sequencing

Front-running blocked (blind-path transactions): When a user uses the blind path, the sequencer commits to transaction order at `arrival_tick` before the plaintext is visible (5 ms before `reveal_tick`). Any attempt to insert a front-running transaction after seeing the plaintext would require a lower `arrival_tick`, which the VDF clock cannot produce retroactively. The chain state at any prior tick is already fixed.

Sandwich attacks blocked (blind-path transactions): A sandwich requires placing attacker transactions at positions N−1 and N+1 around the victim at position N. With blind sequencing active, the attacker cannot see the victim's transaction content at arrival time, so targeted bracketing is impossible. The attacker would need to sandwich every arriving ciphertext, which is economically infeasible. Plain-path transactions do not carry this guarantee until mandatory blind mode is activated.

Commitment binding: `commitment = BLAKE3(plaintext)` prevents transaction substitution. If a user reveals a different transaction, `BLAKE3(new_tx) ≠ commitment` and the kernel rejects it at the syscall boundary. Neither the user nor the sequencer can swap a committed ciphertext for different content after the arrival_tick is fixed.

No key escrow: Unlike threshold encryption schemes (e.g., Shutter Network), there is no trusted committee holding decryption keys. Each user holds their own key and reveals it voluntarily after the blind window. The sequencer never has early decryption capability.

Censorship resistance preserved: Blind sequencing does not impair forced inclusion (§5.3). If a user's blind transaction is not included within 16 slots, the forced inclusion path applies to the ciphertext. The ciphertext arrival_tick commitment is the basis for proving censorship on-chain.

ZK-provable intra-epoch ordering: The `VdfCheckpoint` events logged every 32 ticks create ZK anchor points in the kernel event ring. The SP1 circuit verifies the Poseidon2 chain between checkpoints, making intra-epoch ordering as cryptographically strong as the execution proof.

Epoch seed formal hardness: Each 32-slot epoch begins with a class-group VDF epoch seed. The Wesolowski proof included in L1 calldata guarantees, under the Strong RSA assumption in class groups, that the sequencer spent at least T sequential squarings of real time computing the seed. No party can pre-compute or grind epoch seeds without this delay. This is a formal proof-of-work at epoch boundaries with no trusted setup.

---

7. Operator Economics

6.1 How Operators Earn Revenue

DEOS Tessera sequencer operators earn revenue from two sources:

Sequencing Fees: Each L2 chain that uses DEOS Tessera as its sequencer pays a per-batch or per-transaction fee. This fee is negotiated off-chain between the L2 team and the DEOS Tessera operator set, and flows to the operator designated for each slot. An operator winning 10% of all slots earns 10% of total sequencing revenue.

MEV (Bounded): Within a slot, the designated operator determines transaction ordering. That creates limited MEV capture opportunities where the operator can prioritize higher-paying transactions. However, this is bounded by the slashing mechanism: provable censorship (ignoring lower-fee transactions completely) triggers slashing, and ordering equivocation (contradicting a signed certificate) is slashable. The economic design discourages the most harmful MEV strategies while allowing operators to earn priority fee income.

6.2 Incentive Structure

Action	Economic Effect
Commit valid batches	+Sequencing fees; +reputation for L2 clients
Submit Tier A ZK proof	+Shorter dispute window (10 min vs 1 hour); lower counterparty risk
Earn committee certification	+Shortest dispute window (5 min); higher trust from L2 clients
Miss assigned slot	−10% stake; lost slot revenue
Invalid state root	−50% stake; reputation damage
Censor transactions	−25% stake per proven censorship event
Equivocate	−100% stake (total loss)

The unbonding period (7 days) ensures operators cannot exit to avoid penalties. Their stake remains at risk for a full week after requesting withdrawal. This aligns with the maximum dispute window and prevents the "stake and run" attack where an operator makes fraudulent batches then immediately withdraws.

6.3 Capital Efficiency

A 1 ETH minimum stake is sufficient to start earning sequencing revenue. Higher stake earns proportionally more slots. An operator staking 32 ETH (same as an Ethereum validator) would expect:

- Roughly 3.2% of all sequencing slots (at 32 ETH out of a 1,000 ETH total staked pool)

- Revenue proportional to the sequencing fee rate multiplied by slot share
- Lower counterparty risk for L2 clients (more to lose = more trustworthy)

Stake is not locked against productive use. It earns sequencing fees while serving as security collateral. The economics are similar to Ethereum validator staking, where staked ETH simultaneously secures the network and earns yield.

6.4 Illustrative Economic Model

The following uses rough estimates to illustrate the economic structure. Actual figures depend on adoption, governance-set fee rates, and hardware costs. All numbers are illustrative, not commitments.

Assumptions:

- Slot duration: 1-5 seconds (governance parameter; target for soft finality)
- Target connected chains: 3-10 L2s, each contributing 50-300 TPS
- Aggregate throughput: 200-2,000 TPS across all chains
- Sequencing fee: 0.001 —0.005 per transaction (10-100x below typical L2 gas fees, positioning sequencing as a commodity cost)
- Total operator pool: 1,000 ETH staked (50 operators, avg 20 ETH each)

Operator revenue at 500 TPS aggregate:

Metric	Value
Transactions per day	≈43M
Gross sequencing revenue (at $0.002/tx)	≈$86K/day
Operator with 10% stake share (100 ETH)	≈$8,600/day gross
Annual gross (10% operator)	≈$3.1M

Proving cost (Tier A, local 32-core server):

- Cloud equivalent: roughly $2-4/hour for a 32-vCPU instance
- Each proof covers one slot of transactions; at 1-second slots, continuous proving requires roughly 1 prover per slot in the pipeline (proofs run async, 2-5 min each, ~120-300 provers in parallel or SP1 Network outsourcing)
- SP1 Network pricing: currently roughly 0.10 —0.50 per proof; at 86,400 proofs/day roughly $8K$ —43K/day proving cost across all operators
- Proving cost represents the primary operational expense; higher throughput improves unit economics

Break-even analysis: An operator with 10 ETH stake, earning roughly 1% of slots at 500 TPS aggregate, grosses roughly $860/day before proving costs. With SP1 Network outsourcing at proportional cost, proving runs roughly $80 —430/day (1% of total). Net positive at current estimates; economics improve as TPS scales.

These figures are sensitive to adoption (TPS), fee rates, and proving cost trends. The economic model becomes substantially more favorable as ZK hardware accelerators reduce proving costs and as SP1 Network pricing scales down with volume.

---

8. Integration Guide: Connecting an L2 to DEOS Tessera

7.1 What Changes on the L2 Side

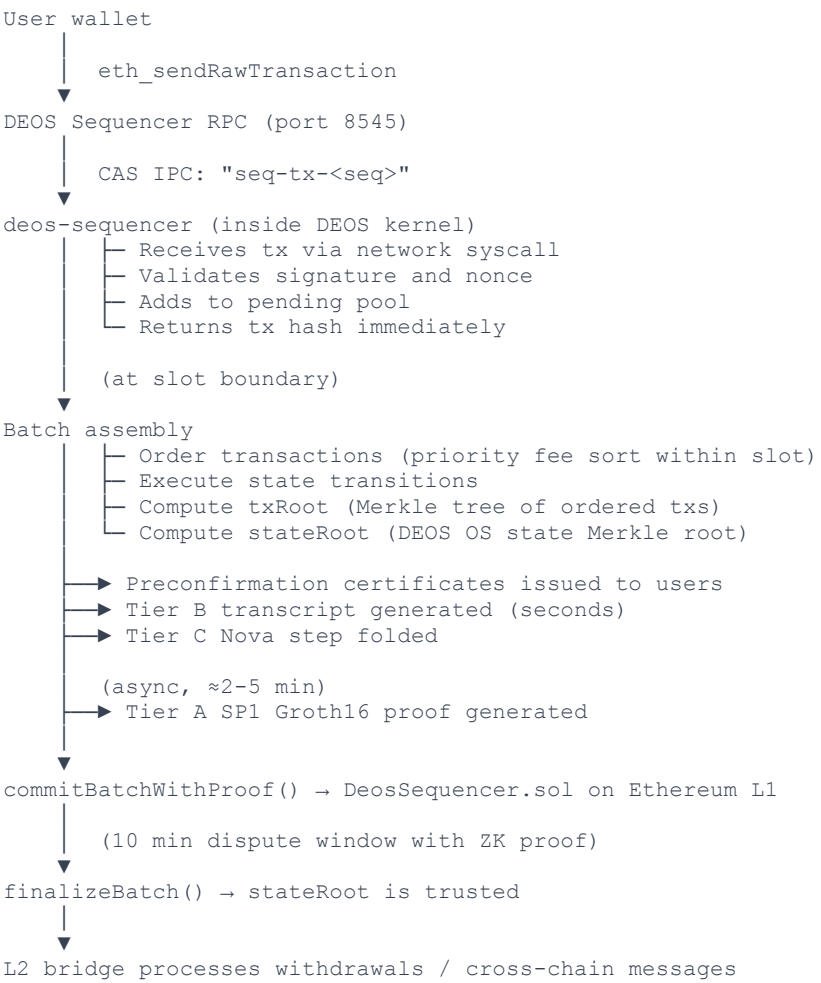
Integrating with DEOS Tessera as a shared sequencer requires three changes to an existing L2:

1. Sequencer Endpoint: The L2 node configuration gets updated to submit transactions to the DEOS Tessera sequencer's JSON-RPC endpoint instead of a local sequencer process. For OP Stack chains, this is the `sequencer_endpoint` in the rollup config. For Arbitrum Orbit chains, this is the `sequencer-url` in the node config.

2. State Root Source: The L2's bridge/rollup contract gets updated to read finalized state roots from `DeosSequencer.sol` instead of a chain-specific contract. The L2 maps its `chainId` to the relevant batches.

3. Dispute Contract: The L2's existing fraud proof or validity proof contract is extended to accept the `eventLogHash` from DEOS Tessera as the canonical batch reference, allowing disputes to use the DEOS replay mechanism.

7.2 Transaction Submission Flow



7.3 Verifying a Batch (Anyone Can Do This)

No special software is required to verify a DEOS Tessera batch beyond the DEOS OS itself, which is open source:

```
# 1. Get the event log URI from the on-chain batch
cast call $SEQUENCER_ADDR "getEventLogUri(bytes32)" $BATCH_ID

# 2. Download the event log
curl <eventLogUri> > batch.deos-log

# 3. Verify the log hash matches what's on-chain
dverify replay --log batch.deos-log --expected-hash <eventLogHash>

# 4. Replay and compute the state root
dverify replay --log batch.deos-log --compute-root
# Output: stateRoot = 0xabc123...

# 5. Compare with the on-chain commitment
cast call $SEQUENCER_ADDR "getTrustedStateRoot(bytes32)" $BATCH_ID
```

If the computed state root differs from the on-chain commitment, the batch is disputable during the dispute window.

7.4 OP Stack Integration (Planned)

For OP Stack L2s (Optimism, Base, Mode, Zora, etc.):

1. Replace `op-batcher` with DEOS sequencer, which produces the same `(txRoot, stateRoot)` commitments but posts them to `DeosSequencer.sol` in addition to the OP Stack rollup contract
2. The OP Stack fault dispute game is extended to accept DEOS event log replay as a valid execution trace
3. The `op-proposer` reads state roots from finalized DEOS batches instead of computing them locally

7.5 Arbitrum Orbit Integration (Planned)

For Arbitrum Orbit chains:

1. The Arbitrum sequencer feed (`sequencer.feed.url`) is replaced with DEOS's RPC endpoint
2. The `RollupCore.sol` is extended to accept state roots from finalized DEOS batches
3. Arbitrum's existing fraud proof (BOLD) can be used to challenge DEOS state roots during the 1-hour dispute window

---

9. Phase 4: Kernel-Bound Proof State

A key innovation that distinguishes DEOS from post-hoc proof systems: **the kernel emits ZK leaf snapshots at execution time**, binding proofs to actual hardware execution rather than reconstructing state from logs afterward.

8.1 The Gap in Traditional ZK Sequencers

Traditional ZK rollup provers work offline: collect the transactions, replay them in a virtual machine, generate a proof of that replay. The proof certifies the VM execution, not the actual sequencer hardware execution.

There's a subtle but critical gap here: an adversary could run the honest VM computation for the proof while running a different (malicious) computation on the actual sequencer hardware. The ZK proof would verify correctly, but the sequencer's actual behavior would differ from what the proof claims.

The trust boundary is the kernel binary. Phase 4 narrows this gap (but does not eliminate it). It ensures the ZK proof reflects what the running kernel actually computed. However, if the kernel binary itself is replaced before the sequencer begins operating, the proof faithfully records the modified kernel's behavior, which may differ from

the published DEOS source. The defenses against this are: (1) the DEOS kernel binary is content-addressed (BLAKE3 hash); operators can be required to publish their binary hash at registration, and any verifier can check it against the canonical build; (2) the deterministic replay guarantee means that anyone who suspects a modified kernel can replay the event log against the canonical binary. If the state roots diverge, that divergence is on-chain evidence of substitution. This is a meaningful improvement over post-hoc proof systems, but it is not a claim of hardware-level trust; the trust boundary remains the kernel binary.

8.2 DEOS's Solution: Live State Binding

During syscall execution, DEOS kernel handlers emit a pending leaf snapshot to a per-CPU ring at the exact moment the syscall executes:

```
// Inside sys_fork():
let old_leaf = live_state::get_leaf(TREE_PROCESS, child_pid);
let new_leaf = live_state::fork_process_hash(child_pid as u32, parent_pid);
live_state::set_leaf(TREE_PROCESS, child_pid, new_leaf);
live_state::set_pending(TREE_PROCESS, old_leaf, new_leaf);
```

The dispatch path consumes this snapshot when logging the syscall event:

```
let (leaf_old, leaf_new, leaf_tree) = live_state::take_pending();
events::log_syscall_with_leaf(syscall_num, args, result, tid, leaf_old, leaf_new, leaf_tre
e);
```

The `EventType::Syscall` struct carries `leaf_old`, `leaf_new`, and `leaf_tree` fields in the kernel's ring buffer. These values were computed at the moment the syscall executed. They cannot be forged after the fact.

8.3 Bridge Consumption

The ZK proof bridge reads these kernel-provided leaf values when constructing Merkle witnesses:

```
if event.leaf_tree != TREE_NONE {
    // Seed the live Merkle tree with the kernel-emitted leaf value
    tracker.seed_leaf(event.leaf_tree, key, event.leaf_old);
}
```

By seeding the proof from kernel-emitted values rather than reconstructing them independently, the bridge ensures that the ZK proof is cryptographically tied to the kernel's actual execution path. Any divergence between what the kernel executed and what the proof claims would require forging a Poseidon hash, which no known attack achieves.

8.4 Coverage

Four syscall families emit live leaf snapshots:

Syscall	Tree	Key	Transition
`fork()`	Process	`child_pid`	Empty → Running process entry
`pipe()`	FD	`pid<<32|fd`	Empty → Pipe read-end descriptor
`mmap()`	Memory	`addr/4096`	Empty → Mapped page (flags + vpn)
`exit()`	Process	`pid`	Running → Zombie process entry

`sys_exit()` is special: it logs directly inside the handler before the irreversible context switch, because the dispatch path never runs after process exit.

---

10. Technical Specifications

9.1 Contract Addresses (Sepolia Testnet)

Contract	Address
`DeosSequencer`	`0x9180c058304077b7aafa2a4ff42095287597033d` (Sepolia)
`Groth16Verifier`	`0x1826673ee12141b52d22e91cbfb3d0b46c436e60` (Sepolia)

9.2 Proof System Parameters

Parameter	Value
Proving system (Tier A)	SP1 v6.0.2, Groth16 over BN254
Trusted setup (Tier A)	Aztec Ignition Powers of Tau (214 participants)
Proving time (Tier A, local)	2-5 minutes (32-core server)
Proving time (Tier A, SP1 Network)	30-60 seconds
Hash function (in-circuit)	Poseidon2 (`t=12`, `RF=8`, `RP=22`, `α=7`)
Hash function (Tier B)	Poseidon2 over Goldilocks (`p=2^64-2^32+1`)
Verification time (Tier B)	<1 second (laptop)
Folding scheme (Tier C)	Nova IVC, uniform step circuit
Merkle tree depth	32 levels (sparse)
Constraints per Merkle path	≈11,328 (Poseidon2)
Constraints per Nova step	≈13,256
Tier A on-chain gas	≈300K gas (BN254 pairing)
Tier A proof size	≈200 bytes (3 BN254 curve points)
Tier C on-chain gas	≈1M-5M gas (pending EIP-2537)

9.3 Slot Parameters

Parameter	Value
Epoch length	32 slots
Unbonding period	7 days
Forced inclusion window	16 slots
Liveness window	4 slots
Dispute window (plain)	1 hour
Dispute window (ZK-proven)	10 minutes
Dispute window (certified)	5 minutes
Equivocation penalty	100% stake
Invalid root penalty	50% stake
Liveness penalty	10% stake
Censorship penalty	25% stake
Minimum operator stake	1 ETH

9.4 The DEOS Event Log Format

Each slot's event log is a sequence of 256-byte `KernelEvent` records containing:

- Global event ID (Lamport clock)

- Vector clock (16-CPU multi-core causality)
- TSC-based nanosecond timestamp
- CPU ID
- Event type: `Syscall`, `ContextSwitch`, `Network`, `PageFault`, `ProcessLifecycle`, etc.

For `Syscall` events: `syscall_num`, `args[6]`, `result`, `tid`, `leaf_old[32]`, `leaf_new[32]`, `leaf_tree`.

The log is content-addressed via BLAKE3 and stored in DEOS CAS at key `seq-events-<slot>`. The `eventLogHash` committed on-chain is the BLAKE3 hash of this log. Anyone who downloads the log can verify it matches the hash, then replay it to verify the state root.

9.5 Deterministic Replay

The replay is fully deterministic: all sources of non-determinism are recorded in the event log and re-injected during replay:

- **RDTSC traps:** TSC (timestamp counter) reads are trapped, the recorded value is injected
- **RDRAND traps:** Hardware random number generator reads are trapped, the recorded bytes are injected
- **Network I/O:** All network responses are logged with content hashes, replayed from CAS
- **Timer interrupts:** IRQ timing is recorded and re-fired at the same logical points

The computed state root after replay must match the committed on-chain state root. Any deviation is cryptographic evidence of fraud.

---

11. Comparison with Existing Solutions

Feature	Optimistic Rollup	ZK Rollup	Espresso/Astria	DEOS Tessera
Proves execution correctness	Fraud proof (7d)	Validity proof	BFT consensus	ZK proof (10 min)
Proves sequencer honesty	❌	❌	Social BFT	✅
Ordering fairness (anti-MEV)	❌	❌	BFT ordering	**VDF blind (blind-path txs); Groth16 integration planned**
Front-running protection	❌	❌	❌	✅ (blind-path transactions)
Ordering proof on-chain	❌	❌	❌	Kernel-enforced today; **Groth16 on roadmap**
Censorship resistance	Optional	Optional	Optional	Forced inclusion + slash
Liveness guarantee	❌	❌	BFT liveness	Liveness slashing
Equivocation protection	❌	❌	BFT safety	On-chain slash
Multi-chain	❌	❌	✅	✅
Trusted setup required	N/A	Yes (most)	No	One-time (Aztec Ignition)
VDF trapdoor risk	N/A	N/A	N/A	**None (Poseidon2)**
On-chain proof cost	N/A	≈1M gas	N/A	≈300K gas
Replay verifiability	Yes (slow)	No	No	Yes (seconds)
Preconfirmation certificates	Optional	Optional	Yes	Yes + slashable

Feature	Optimistic Rollup	ZK Rollup	Espresso/Astria	DEOS Tessera
OS-level execution binding	❌	❌	❌	✅ (Phase 4)
Built-in threat detection	❌	❌	❌	✅ (AiDR)

---

12. Security Analysis

11.1 What Is Cryptographically Guaranteed

- **Execution correctness:** If a Tier A proof verifies on-chain, the sequencer's execution was correct. The RISC-V circuit enforces the DEOS state machine.
- **State binding:** Phase 4 kernel integration binds the ZK proof to actual hardware execution via live leaf snapshots. Forging a proof would require inverting Poseidon.
- **Equivocation impossibility:** Two valid proofs for the same `(slot, chainId)` with different state roots would require finding a Poseidon collision. Equivocation is deterred by slashing for the probabilistic case (two committed batches without proofs).
- **Non-censorship:** Forced inclusion with on-chain slashing provides a hard time-bounded liveness guarantee for individual transactions (16 slots maximum delay, provably).

11.2 What Requires Honest Operators

- **Ordering fairness within a slot (mitigated by VDF blind sequencing, §6):** The VDF blind path eliminates content-based MEV (front-running, sandwiching) for all blind transactions. Operators commit to a ciphertext ordering before content is visible. Operators retain the option to process plain (non-blind) transactions without the blind delay. The blind path is available to any user but not currently mandatory. Making blind sequencing mandatory for all intra-slot transactions is a planned governance option.
- **DA availability:** The event log must remain downloadable for the duration of the dispute window. Blob commitments (EIP-4844, roughly 18 days availability) and the `eventLogUri` provide strong practical DA guarantees. A fully unavailable event log that prevents dispute submission would require a censorship attack on both Ethereum blobs and the event log storage simultaneously.

11.3 Trust Minimization vs. Prior Art

ZK rollup sequencers prove that *output is valid given input*, but they do not prove what the sequencer decided to include as input or in what order. DEOS adds the missing layer: proof that the sequencer's execution, from receiving transactions to committing batches, was faithful, observable, and replayable.

Shared sequencer committees (Espresso, Astria) reduce the single-operator problem but replace it with a committee trust assumption. DEOS eliminates the committee trust assumption for execution correctness: the ZK proof is valid regardless of how many operators are colluding.

11.4 OS-Level Tamper Detection (AiDR)

DEOS Tessera runs on a kernel with built-in anomaly detection (AiDR). From a security analysis perspective, the relevant property is: disabling anomaly detection requires modifying the kernel binary, which changes the kernel's BLAKE3 content hash. A sequencer node running modified kernel code produces event logs that diverge from those an honest node would produce for the same workload. That discrepancy is detectable by any party replaying the event log against the canonical kernel binary. Operator infrastructure tampering is detectable through the same replay mechanism used for state root disputes, without requiring trust in the operator's self-reporting. See Appendix B for implementation details.

13. Failure Modes and Adversarial Analysis

The following describes how the protocol behaves under attack, and where honest assumptions are required.

Malicious sequencer submitting an invalid state root: The operator is slashed when any party downloads the event log, replays it through DEOS, and submits a dispute with the correct state root during the dispute window. The replay is deterministic, so both parties running the same log reach the same result. A valid-looking false Tier A Groth16 proof would require breaking SP1's soundness assumption.

VDF epoch seed grinding: An operator whose execution determines the epoch seed (derived from the previous epoch's last VDF state) could grind it to bias slot assignment in future epochs. Planned mitigation: incorporate external entropy (e.g., Ethereum RANDAO) into the epoch seed computation. Not yet implemented.

Stake concentration: If a single entity accumulates enough stake to dominate slot assignment (e.g., >50% of staked ETH), they control most batch commitments. Economic penalties deter individual bad acts, but a well-capitalized adversary could afford targeted slashing. Permissionless operator registration and decentralization of the operator set are roadmap items intended to bound this risk.

DA withholding: An operator commits a batch but withholds the event log, preventing dispute submission. EIP-4844 blob commitments directly mitigate this: the `blobHash` is committed on-chain, and Ethereum guarantees blob propagation for roughly 18 days. Withholding would require simultaneously censoring the Ethereum blob network, a separate and substantially harder attack.

Forced inclusion timing attack: The 16-slot forced inclusion window is enforced on-chain. An adversary who fills Ethereum blocks with competing transactions could delay forced inclusion transaction processing, effectively extending the censorship window. This is an economic attack (requires sustained gas expenditure), not a free one.

Poseidon2 cryptanalysis advance: A significant algebraic cryptanalysis breakthrough could weaken Poseidon2. Upgrading the hash function would break compatibility with existing proofs and commitments and would require a migration. This risk is shared by most ZK systems deployed today.

Trusted setup compromise (Groth16 only): If the Aztec Ignition ceremony was fully compromised, an attacker could generate a false Tier A proof for any execution trace. Tier B (no trusted setup) and replay-based dispute would still function correctly; only the 10-minute dispute window shrinks to the 1-hour fallback.

Blind path adoption: If most users submit plain-path transactions, VDF ordering fairness protects only the minority using blind mode. MEV extraction remains possible against plain-path transactions. Making blind mode mandatory across the operator set is a governance decision that has not yet been made.

---

14. Roadmap

The table below separates what is implemented in the current codebase from what is planned. Performance claims in §10 Technical Specifications are measured from the running system on commodity hardware (32-core server for Tier A, standard laptop for Tier B), except where noted as estimated for planned features.

Milestone	Description	Status
L1 contract deployment	`DeosSequencer.sol` + `Groth16Verifier` on Sepolia	✅ Complete
SP1 guest ELF build automation	`build.rs` + `sp1-build` -- no manual steps	✅ Complete
Tier B Goldilocks transcript	Poseidon2 transparent proof	✅ Complete

Milestone	Description	Status
Tier C Nova IVC	Recursive epoch folding	✅ Complete
Phase 4 kernel binding	Live syscall leaf snapshots	✅ Complete
VDF kernel module	Poseidon2 sequential hash clock (`kernel/src/vdf/`)	✅ Complete
Blind sequencing syscalls	`SYS_VDF_SUBMIT_ENCRYPTED` / `REVEAL` / `DRAIN` (730-734)	✅ Complete
Userspace VDF client	`libdeos::vdf::VdfClient` + `BlindTx`	✅ Complete
Sequencer blind integration	`deos-sequencer` drains and orders by `arrival_tick`	✅ Complete
VDF inside SP1 circuit	Poseidon2 chain proven in Groth16 proof	Planned
Class-group VDF epoch seeds	Formal sequential hardness (no trapdoor) for epoch seed generation	Planned
Mandatory blind mode	Policy option: all txs must use blind path	Planned
Mainnet deployment	Production operator set, real stake	Planned
EIP-2537 Nova verifier	Cheap on-chain Tier C (≈200K gas)	Pending EIP
Decentralized operator set	Permissionless operator registration	Planned
OP Stack adapter	Drop-in replacement for op-batcher	Planned
Arbitrum Orbit adapter	Replace sequencer feed with DEOS	Planned
SP1 Network integration	Sub-60s Tier A proofs via Succinct	Planned

---

15. Conclusion

The Ethereum ecosystem has made extraordinary progress on execution correctness. ZK proofs now verify that L2 state transitions are valid. But the sequencer layer remains a trusted black box. Users accept that the sequencer *could* reorder, censor, or lie, and simply hope it does not.

DEOS Tessera closes this gap by making the operating system itself the proof boundary:

1. **Cryptographic proof of sequencer execution**, covering execution faithfulness, not just output validity
2. **Deterministic replay** where any party can reconstruct the exact state from the event log in seconds
3. **On-chain dispute resolution** so incorrect state roots are slashable in 10 minutes with a ZK proof
4. **Censorship resistance** through forced inclusion with automatic economic penalties
5. **VDF blind sequencing** that blocks content-based MEV (front-running, sandwich attacks) for blind-path transactions; kernel-level ordering enforcement today, with Groth16 proof integration on the roadmap
6. **OS-level AiDR** for threat detection built into the kernel, detectable if tampered
7. **Multi-chain shared sequencing** where one operator set sequences multiple L2s atomically

No other sequencer combines kernel-level execution provability, VDF-enforced ordering fairness, and on-chain slashing in a unified system. DEOS Tessera reduces the trust placed in the sequencer from a social guarantee to a combination of cryptographic proofs, auditable execution records, and aligned economic incentives.

Appendix A: Glossary

Term	Definition
AiDR	AI Detection and Response: real-time threat classification built into the DEOS kernel
Arrival tick	The VDF clock value when a blind transaction's ciphertext was accepted; the irrevocable ordering commitment
Blind sequencing	A protocol where users encrypt transactions before submission, forcing the sequencer to order ciphertexts it cannot read
Blind window	The `MIN_REVEAL_DELAY` period (50 ticks = 5 ms) during which the sequencer holds a ciphertext before the user may reveal the decryption key
Batch	A compressed group of ordered L2 transactions committed to Ethereum L1
BFT consensus	Byzantine Fault Tolerant consensus: a committee voting protocol that tolerates up to 1/3 malicious participants
BN254	A pairing-friendly elliptic curve used in Groth16 and supported by Ethereum precompiles
BLAKE3	A fast cryptographic hash function used for content-addressed storage in DEOS
Blob (EIP-4844)	Cheap temporary Ethereum storage (128KB, roughly 18 days) used for L2 batch data
CAS	Content-Addressed Storage: data stored by the hash of its contents, not by name
DA	Data Availability: ensuring transaction data is downloadable for independent verification
Equivocation	Signing two conflicting statements about the same slot; the most severe sequencer offense
Finality	The point at which a transaction is cryptographically irreversible
Fraud proof	Evidence submitted on-chain proving that a posted state root is incorrect
Goldilocks	A prime field `p = 2^64 - 2^32 + 1` that fits in a `u64`, enabling fast arithmetic
Groth16	A ZK proof scheme producing constant-size (roughly 200 byte) proofs, requires trusted setup
Hard finality	Irreversible settlement on Ethereum L1
IVC	Incremental Verifiable Computation: proving a sequence of steps without producing N separate proofs
L1	Ethereum mainnet (Layer 1), the base settlement layer
L2	Layer 2: an off-chain execution environment that posts summaries to L1
Lamport clock	A logical timestamp that tracks causal ordering of events across distributed systems
MEV	Maximal Extractable Value: profit from reordering, inserting, or censoring transactions
Merkle tree	A hash tree where the root commits to all leaves; membership proofs are O(log N)
Nova	A ZK folding scheme for IVC; accumulates steps without trusted setup
Operator	A staked participant who runs a DEOS Tessera node and commits batches to L1
Poseidon / Poseidon2	ZK-friendly algebraic hash functions with roughly 200 constraints per call (vs 50K for SHA-256 in a circuit)
Preconfirmation	A signed receipt from the sequencer guaranteeing transaction inclusion before L1 finality
R1CS	Rank-1 Constraint System: the format ZK proofs use to express computations
Sequencer	The server that accepts, orders, executes, and batches L2 transactions
Shared sequencer	A neutral sequencing service used by multiple L2s simultaneously
Slash	Confiscate an operator's staked ETH as a penalty for provable misbehavior
Soft finality	A fast, economically-backed confirmation from the sequencer, before L1 settlement
Sparse Merkle tree	A Merkle tree over a large key space where most leaves are empty
SP1	Succinct's RISC-V zkVM; compiles arbitrary Rust programs into ZK proofs
SRS	Structured Reference String: the cryptographic parameters produced by a trusted setup
State root	A 32-byte Merkle hash committing to the entire L2 state after executing a batch

Term	Definition
Syscall	A request from a user program to the OS kernel (e.g., read file, create process, send data)
Trusted setup	A one-time ceremony generating ZK parameters; safe if at least one participant was honest
TSC	Timestamp Counter: a CPU register counting clock cycles, used for nanosecond timing
Validity proof	A ZK proof that a state root is correct, used in ZK Rollups for instant finality
VDF	Verifiable Delay Function: a sequential computation whose output proves that time has passed, verifiable in O(1)
VDF tick	One step of the DEOS VDF hash chain, executing every 100 µs
zkVM	A virtual machine whose execution can be proven in zero knowledge

---

DEOS Tessera is built on roughly 243K lines of Rust (kernel + userspace), deployed on Sepolia testnet, with proof generation running on commodity hardware via the Succinct SP1 toolchain.

Appendix B: AiDR -- AI Detection and Response

DEOS Tessera runs on a hardened OS with built-in AI Detection and Response (AiDR). This is not an add-on security layer. It is the kernel itself.

Architecture

Neural security policy: A 64-dimensional observation vector encodes the current process's syscall behavior (frequency distribution, memory access patterns, network activity, privilege level). A REINFORCE-trained feedforward network classifies this vector in real time at each syscall boundary, producing a threat score and an action recommendation (monitor / escalate / kill).

Rogue process detection: Processes exhibiting anomalous patterns (unusual memory access outside their VMA, unexpected outbound network connections, privilege escalation attempts, syscall sequences not seen during training) trigger automatic suspect escalation. Suspect status persists across reboots via CAS.

DMS-1 on-device inference: DEOS's own model standard (1.25M parameters, sensory encoder + curiosity-driven learner). No external LLM dependency. DMS-1 continuously learns from sequencer execution patterns, improving anomaly detection over time. Weights persist in CAS.

Why It Matters for Sequencers

An attacker who compromises the sequencer host OS can disable traditional intrusion detection software. With DEOS, disabling AiDR requires modifying the DEOS kernel binary. That changes the kernel's BLAKE3 content hash, which is immediately visible in the kernel event log and detectable by any replaying party.

A compromised sequencer node **self-reports its compromise** through the event log. Post-incident forensics are as deterministic as the sequencer execution itself: boot DEOS, replay the event log, and reconstruct the exact anomaly that triggered the security alert, with the same cryptographic guarantees as any other batch verification.

Integration with ZK Proof Pipeline

AiDR events are logged as standard `KernelEvent` records alongside sequencer syscall events. They appear in the event log hash committed on-chain. An auditor replaying a batch will see security alerts exactly as the kernel issued them, in causal order with the sequencer operations they flagged.

Appendix C: Reviewer FAQ

C.1 Why prove OS syscalls instead of VM instructions?

Popular zkVMs (SP1, RISC Zero, Jolt) prove that a specific RISC-V program executed correctly on given inputs. That is the right tool for proving *computation*, but it is not the right tool for proving *ordering*.

A sequencer's fairness guarantee is fundamentally about causal precedence: transaction A must demonstrably have arrived at and been committed by the sequencer *before* transaction B. That precedence is established at the OS event layer: when the kernel's TCP receive path handed the bytes to the sequencer process, when the scheduler dispatched the sequencer thread, when the `write()` syscall committed A to the pending batch. None of these events are visible inside a RISC-V zkVM; they are OS-level events that precede and enclose any user-space computation.

Proving VM instructions answers "did this computation produce the correct output?" Proving syscalls answers "in what order did these real-world events occur, and on what kernel did they occur?" Tessera needs the second answer. The two are complementary: Tier A Groth16 proofs (SP1) cover computation correctness inside the sequencer process; the kernel event log (with its Lamport-clock causal chain) covers the ordering of those syscalls in real time.

A secondary motivation: syscall-level proofs capture the full resource environment (memory allocations, network I/O, scheduler decisions) that affects ordering. A VM-instruction proof of the sequencer binary would prove that the binary behaved correctly *given* an ordering, but cannot prove that the ordering itself was not manipulated by the host OS.

C.2 Why not use threshold encryption instead of VDF blind sequencing?

Threshold encryption schemes (Shutter Network, Ferveo, SUAVE-style commit-reveal) encrypt transaction content under a committee public key. The committee reveals the decryption key only after the ordering is finalized, preventing content-based front-running.

Tessera's VDF approach and threshold encryption address the same adversary (a sequencer that reorders based on tx content) but with different trust and liveness properties:

Property	Threshold Encryption	Tessera VDF Blind Window
Liveness dependency	Honest-majority committee online at decryption time	None: VDF is a solo computation
Latency	Round-trip to committee for each batch	Blind window adds 5 ms; no committee latency
Trust assumption	Honest-majority of keyholders	Sequencer cannot compute future VDF outputs
Content concealment	Full concealment until decryption	Concealment until arrival tick + 50 ticks
Verifiability	Committee attestation	VDF output verifiable by any party in O(1)
Failure mode	Key leakage or committee collusion	Sequential hardness assumption broken

The liveness risk is the practical concern: if a threshold decryption committee is offline or colluding, encrypted transactions are permanently undecryptable. Tessera's blind window requires no committee; the VDF sequence is entirely a property of the sequencer's own computation, and its output is publicly verifiable.

The honest limitation: threshold encryption provides *full* content concealment until after ordering, whereas Tessera's blind window provides concealment only for the duration of the window (currently 5 ms). For transactions submitted with blind-path intent, this is sufficient to prevent block-level reordering. For adversaries with sub-millisecond latency advantages, the approaches have different residual exposure. Tessera's roadmap includes hybrid support: blind-path transactions could optionally be combined with a threshold decryption layer for defense-in-depth, using Tessera's VDF as the ordering anchor and a lightweight committee solely for content concealment.

C.3 Why not rely on PBS / proposer-builder separation?

Proposer-builder separation (MEV-Boost, SUAVE, ePBS) decouples the validator's role (attesting to the block) from the builder's role (constructing ordered transaction sets). The goal is to prevent validators from capturing MEV directly and to create a competitive builder market.

PBS is a market-structure intervention, not a cryptographic ordering guarantee. It changes *who* captures MEV (shifting it from validators to a specialized builder market) but does not eliminate reordering within a builder's block. Sophisticated builders front-run, sandwich, and backrun within their constructed blocks; the MEV is extracted by builders rather than validators. The transaction sender's adversary shifts from validator to builder, but the adversary remains.

For L2 sequencers specifically, PBS in its Ethereum L1 form does not apply: L2 sequencers typically have unilateral ordering authority over L2 transactions, with no proposer/builder separation. The MEV-Boost ecosystem is an L1 construction.

Tessera's contribution is orthogonal to PBS: it provides a cryptographic proof that a given ordering *was* the ordering produced by the OS at the time of receipt, regardless of who constructed the block. PBS and Tessera are composable. A Tessera sequencer node could participate in a PBS-style builder market, offering provably-fair ordering as a differentiator. In that configuration, Tessera's causal proofs serve as the builder's commitment to its ordering policy, verifiable by any party examining the on-chain event log hash.

C.4 Why not run the sequencer inside a TEE (SGX / SEV-SNP)?

TEE-based sequencers (e.g., proposals using Intel SGX or AMD SEV-SNP) use hardware attestation to prove that a specific binary is executing in a protected enclave, with memory encrypted against host-OS inspection.

The TEE model has several weaknesses that Tessera's design avoids:

Side-channel attack surface. SGX has been broken by Spectre, Meltdown, Foreshadow, SGAxe, LVI, CacheBleed, PlundervVolt, and others. Enclave memory encryption does not eliminate microarchitectural side channels. SEV-SNP has fewer published breaks but is younger and less studied. Tessera's security does not rely on microarchitectural isolation.

Attestation is not ordering proof. Hardware attestation proves "binary X is running inside enclave Y." It does not prove "transactions arrived in order Z." A malicious operator could run an attested binary that reorders internally before the enclave's I/O boundary; the TEE would attest correctly, and the reordering would be undetectable. Tessera's kernel event log proves the ordering itself, not merely that a binary ran.

Hardware vendor trust. SGX and SEV-SNP require trusting Intel and AMD, respectively, including their key hierarchies, microcode, and supply chains. Nation-state actors may possess undisclosed vulnerabilities in vendor root keys. Tessera's proofs are mathematical, verifiable by any party with the public verifying key and the proof, with no hardware vendor in the trust chain.

No post-hoc auditability. Once an SGX enclave is torn down, its execution cannot be replayed. Tessera's event log supports deterministic replay indefinitely; any party can re-verify any historical batch without special hardware.

Tessera's approach is not inherently incompatible with TEEs: running DEOS inside an SEV-SNP VM would add a hardware attestation layer on top of Tessera's cryptographic proofs, providing defense-in-depth. But Tessera's security properties do not *require* TEE guarantees, and the absence of a hardware trust dependency is a deliberate design choice.

C.5 What happens if most users submit plain transactions?

This is the right question, and the honest answer is that plain transactions receive a weaker protection profile than blind-path transactions.

What Tessera provides for plain transactions:

- *Verifiable ordering record.* Every plain transaction's arrival is logged as a `KernelEvent` with a Lamport timestamp derived from the kernel's TSC-trapped tick counter. Operators cannot retroactively claim a transaction arrived earlier or later than the kernel recorded. Any ordering dispute can be adjudicated by replaying the event log.
- *VDF time anchor.* Even plain transactions are timestamped relative to the VDF tick clock, which is not under operator control. An operator cannot claim a transaction arrived in tick 1000 if the VDF chain places it in tick 1020.
- *On-chain batch commitment.* The Merkle root of every batch is committed to L1. Omitting a transaction from a committed batch is immediately detectable by anyone who observed it in the mempool.

What Tessera does not provide for plain transactions:

Content-based front-running (observing a large DEX trade and inserting a sandwich in the same batch) remains possible for plain-path transactions, because their content is visible to the sequencer at arrival. The blind-path protection (commitment before content is processed) applies only to transactions explicitly submitted via the blind channel.

Why this is acceptable in practice:

MEV extraction requires targeting. Sandwich attacks are only profitable when the target transaction moves price significantly. The plain-path user who submits a small transfer or a routine interaction is not a meaningful MEV target; verifiable ordering is the relevant protection. The plain-path user who submits a large AMM trade can opt into the blind path for that transaction. Tessera's design makes the opt-in cost low (same gas, 5 ms added latency) and the protection strong for that category of transaction.

Network effects also matter: as more MEV-sensitive transactions migrate to the blind path, the MEV opportunity in the plain pool decreases, improving the experience for plain-path users without requiring them to change behavior.

C.6 How does this interact with rollup fault proofs?

Rollup fault proof systems (Optimism's Cannon, Arbitrum's BOLD, zkSync's proof system) allow challengers to dispute invalid L2 state transitions by re-executing the disputed computation on-chain or submitting a ZK proof of correct execution.

Tessera and L2 fault proofs operate at different but compatible layers:

Layer	Fault Proof System	Tessera
What is proven	State transition correctness (given this batch, is this state root valid?)	Ordering correctness (was this batch assembled from transactions in this causal order?)
Dispute mechanism	On-chain re-execution or ZK state proof	On-chain Groth16 ordering proof + event log replay
Trust assumption	L2 execution correctness	Sequencer ordering honesty
Relationship	Downstream (consumes batch)	Upstream (produces batch)

A rollup that uses Tessera as its sequencer receives a Tessera-produced batch with an attached Merkle ordering commitment and a Groth16 proof (on-chain or available on-demand). The L2's fault proof system then takes this batch as input and proves execution correctness as usual. The ordering dispute and the execution dispute are handled by separate, non-overlapping proof systems.

Tight integration path (planned): An L2 could modify its dispute resolution contract to accept Tessera's batch Merkle root as a pinned input, so that a fault proof challenger cannot dispute the ordering of a batch that Tessera has already proven. This separates "did the sequencer order honestly?" (Tessera's domain) from "did the L2 execute correctly given that order?" (the fault proof system's domain), and allows each system to focus on its respective guarantee.

Current limitation: In the present design, Tessera's ordering proofs are independent of the L2's execution proof system. Tight integration requires the L2 to adopt Tessera's batch format as a first-class input to its dispute game, which is an L2-side protocol change. This is on Tessera's integration roadmap but is not required for the base sequencing service.