Overview

This RFC will share an update on the latest block production design and highlight key deviations from the previously-discussed Fernet + Sidecar systems.

The primary aim is to share the direction in which we're building the block production system and get feedback from the community on outstanding design questions.

The RFC will be open until October 11th during which we'll respond to comments and engage in discussion with the community.

Section 1: Block Production Workflow

In this section, we describe the tx lifecycle: from the moment it is authorized by a user, to being included in a block proposal by the sequencer, which gets appended to the Pending Chain on L1.

Note: We use the terms "sequencer" and "proposer" interchangeably throughout this RFC. In both cases, the sequencer/proposer is the entity responsible for building and proposing L2 blocks.

sequenceDiagram participant PXE participant L2 Node participant L2 P2P participant L2 Proposer participant L2 Committee participant L1 Rollup Contracts

```
L2 Proposer->>L1 Rollup Contracts: Watches for new L2PendingBlockProcessed events
L2 Proposer->>L1 Rollup Contracts: Download TxEffects and apply them to get to same state root as on L1

alt If state root does not match
   L2 Proposer->>L2 Proposer: Node is faulty or committee is bad. Scream and yell.
else If state root does match
   L2 Proposer ->> L2 Proposer: Proceed
end

PXE ->> L2 Node: Submit TxObjects + ClientIVC Proofs
L2 Node ->> L2 Node: Verify proofs and store TxObject
L2 Node ->> L2 P2P: Broadcast TxObject + ClientIVC Proofs
L2 P2P ->> L2 Proposer: Broadcast TxObject + Proofs
L2 Proposer ->> L2 Proposer: Execute all TxObjects, construct proposal
L2 Proposer ->> L2 Committee: Send proposal containing the header, state root + list of TxHashes
L2 Committee ->> L2 Committee: Check if they've seen all TxHash's, request TxObjects + proofs for unseen TxHash's, verify versioned hashes
L2 Committee ->> L2 Committee: Execute the block.
L2 Committee ->> L2 P2P: Sign proposal if the proposed state root matches the result of their execution.
L2 P2P ->> L2 Proposer: Aggregate signed proposals.
L2 Proposer ->> L1 Rollup Contracts: Post TxEffects to L1 DA.
L2 Proposer ->> L1 Rollup Contracts: Supply header, state root and signatures as call data.
L1 Rollup Contracts ->> L1 Rollup Contracts: Check signatures, check that versioned hash commitment matches what was published to the DA.
L1 Rollup Contracts ->> L1 Rollup Contracts: Add block to Pending Chain, emit L2PendingBlockProcessed event.
```

Transaction propagation

1. Users send transactions execution requests to their PXE

The PXE simulates all application circuits that correspond to private calls, and generatesprivate kernel proofs client-side. Any public execution requests are enqueued for execution by the sequencer later.

The private tx state diffs, (i.e. note hashes, nullifiers, logs etc., which we call privateTxEffects) and enqueued public execution requests are packaged into what we call a TxObject by the PXE and then broadcast to a node along with the ClientIVC proof generated during client-side simulation.

1. Nodes validate TxObjects, store a TxHash and then broadcast to more Nodes

A node which receives a transaction from a user's PXE or another node, must first verify the private kernel circuit proof sent with the TxObject

. Once they verify the proof, they store the TxObject

including a TxHash

, and then broadcast the TxObject

and the private kernel proof to the p2p layer.

No execution is necessary at this step.

Epoch initialization

1. Compute the committee

At the beginning of an epoch, the L1 contract samples a seed using pseudo-randomness supplied from block.prevrandao

.

In the beginning of epoch N+1

, we use the seed computed in a previous epoch (typically epoch N

) to select a committee of size TARGET_COMMITTEE_SIZE

, sampling without replacement from the validator set using theSwap-or-not algorithm. We use Swap-or-not because:

1. There is a precedence for using it (Ethereum)
2. The cost for computing the committee size does not depend on the size of the validator set i.e. O(k)

where k

= TARGET_COMMITTEE_SIZE

The committee (and order) is fixed for the duration of the Epoch.

1. Computing Proposers

At the beginning of each epoch, with a fixed epoch

, sampleSeed

, and validatorSetSize

, proposers can figure out the specific slot in the current epoch for which they've been assigned proposer duties.

Difference from Fernet

In the previous Fernet proposal, sequencers sample a new block.prevrandao

value every slot to determine their odds of being selected as proposer. In the current proposal, proposers don't have to sample anything. They query the L1 contract to figure out the slots for which they've been assigned proposer duties.

Proposer Lookahead

Currently we don't have an implemented committee or proposer lookahead even though the seed (randomness) is sampled in the previous epoch.

This is because even though the randomness for epoch N+1

is available from the start of epoch N

, the validator set might change at the beginning of epoch N+1

. This affects the committee computation, and the order of validators in the committee

array. Thus also affecting proposer election.

The plan is to implement snapshots of the validator set so we could achieve at least a full epoch (i.e. 32 slots) of proposer and attester lookahead.

Proposer builds block

1. Proposers execute transactions and compute side effects and a state root

For each slot, the selected sequencer must collect enough TxObjects

from the p2p layer (or from its private stash) to satisfy minTxPerBlock

. The sequencer executes each transaction they want to include to generate a list of state diffs (per tx) and a final state root (per block proposal).

1. Proposers send the proposal to the committee to collect attestations

A proposal containing the header, a list of TxHash, and a state root is sent to every committee member.

1. The committee verifies the proposal

Each committee member which receives a proposal must:

1. Check that they've seen the TxHash

before i.e. they have verified the proof for the corresponding tx. An honest node would request from the p2p layer the TxObject

and associated proof for a TxHash

that it has not seen before.

1. Execute all the TxObject

objects included in the proposal.

1. Verify the KZG versioned hashes included in the header.
2. Compare their resulting state root to that proposed by the sequencer.

If the check passes, an honest committee member would then sign over the proposal and send it back to the sequencer.

Sequencer publishes a block

A sequencer would then collect and aggregate signatures from the committee. If they're able to get the required quorum, then the sequencer:

1. Posts the TxEffects

i.e. block body to blob DA

1. Posts signatures, the block header (which contains a commitment to the TxEffects), and a state root to calldata.

The L1 Rollup contract will perform some checks including:

1. The KZG versioned hash in the header matches what has been published to blob DA
2. The correct proposer proposed the block.
3. Verify global variables such as slot number and block.timestamp

etc.

1. Verify signatures of the L2 attestation committee.

Next sequencer comes online

Now that a block has been appended to the Pending Chain, the next sequencer in line has been anticipating their turn (proposer lookahead) and is ideally online and prepared to begin building blocks.

Before the next proposer begins accumulating txs from the p2p layer, they must first "catch up" to the Pending Chain. To do so, they download the TxEffects

from L1 DA. They apply all TxEffects

and compute a state root. If it matches the Pending Chain state root as published to L1, then they proceed to the block building workflow as above.

Section 2: Proving Workflow

Blocks appended to the Pending Chain are not considered final and they may in-fact reorg under different scenarios. To finalize a block, it must be proven and added to the tip of the Proven Chain. Blocks added to the Proven Chain are considered final from the perspective of the rollup and cannot be re-orged (without an L1 reorg).

Provers must produce proofs for an entire epoch. A proof for a strict subset of an epoch will not be accepted on the L1 rollup

contract.

Proof claim phase

The proving workflow starts from the moment an epoch=N

ends (i.e. 32 slots have elapsed since the start of the epoch).

Proposers in the first C=13

(see FAQ for why C=13

) slots of epoch=N+1

fetch quotes from provers who commit to producing a rollup proof for the entire epoch=N

. The commitment to produce a proof is in the form of a bond posted by the proposer on behalf of the prover.

For example, in the first slot of epoch=N+1

, only the first proposer has the right to post the bond. If they have not posted a bond by the time slot 2 starts, that right is transferred to the slot 2 proposer and so on until we cycle through the first C

proposers.

If no bond is secured within the first C

slots, then the Pending Chain is pruned to the last proven block and we restart trying to grow the Pending Chain again.

Why require a bond?

A bond holds a prover accountable for producing a epoch proof. It also prevents multiple provers from competing to produce the proof, consequently wasting compute resources.

Proof submission phase

If a proving bond is secured within the first C

slots of epoch=N+1

, the prover will have until the end of the epoch to produce a valid proof that is verified on the L1 and accepted into the Proven Chain. If a proof for epoch=N

lands anytime within epoch=N+1

and is verified by the L1 contract, then epoch=N

is added to the Proven Chain and is considered finalized.

If a bond for epoch=N

is posted in the first C

slots of epoch=N+1

but the proof is not accepted by the L1 during the same epoch=N+1

, then the bond is completely slashed, the Pending Chain is pruned to the last proven block, and we restart trying to grow the Pending Chain again.

[

Proving Phases

1920×964 67.7 KB

](https://europe1.discourse-cdn.com/flex013/uploads/aztec/original/2X/3/3017d03626b0ead2cc966c3a4286216112c0b4b6.jpeg)

Proving coordination

In the Aztec rollup, proposers coordinate out-of-protocol with provers to: post proving bonds, generate the final rollup proofs, verify them on L1 and advance the Proven Chain.

A coordination mechanism is necessary to ensure that:

1. Proposers are able to obtain proofs for epochs, considering they will likely not have proving infrastructure.
2. Provers don't have to "race" each other to submit proofs, as this would be a waste of resources.

"Quotes" are bids for the right to prove the preceding epoch.

Prior to submitting a quote, the prover deposits funds into an L1 escrow contract. The escrow contract implements a withdrawal delay such that a proposer who receives a quote from the prover can be confident that the escrow contract will have the funds required to post the performance bond.

The protocol enshrines the quote and the escrow contract but coordination of quotes is left out-of-protocol.

To that end, the node will support two optional mechanisms that provers can use to submit quotes:

- a p2p topic at /aztec/epoch-proof-quotes/0.1.0
- a json-RPC method at node_submitEpochBid

Both mechanisms expect quotes of the following form:

struct EpochProofQuote { // Signature over the quote details below, and identifying the rollup address on L1. // Must be signed by the L1 address that previously deposited funds in escrow. Signature signature; address proverAddress; // address of the prover uint256 epochToProve; // The epoch the prover is committing to prove. uint256 validUntilSlot; // provers can submit quotes which are valid for multiple slots uint256 bondAmount; // the amount the prover is willing to stake as their bond uint32 basisPointFee; // % of the proving rewards to be collected by prover. }

A proposer can accept any quote they receive by calling the function claimEpochProofRight

on the rollup contract. This causes the rollup contract to direct the escrow to place the bondAmount

from the prover "at stake". The rollup also designates the provider of the bond as the address that will share in rewards as a result of the submission of a proof to the L1 contract.

The prover is expected to deliver the proof to the L1. When it does so, the rollup contract will direct the escrow to unstake the bond.

It is important to note that a well-resourced sequencer can vertically integrate and elect to choose themselves as the designated prover, as long as they are able to post the bond.

Section 3: Liveness & Censorship Resistance

The rollup must provide liveness guarantees and censorship resistance. In the event that a dishonest committee colludes to stop ledger growth and prevent users from getting their transactions included, we need two mechanisms to achieve the aforementioned properties.

Based fallback

In the case participants of the attestation layer are being non-cooperative or repeatedly fail to advance the Pending Chain, we introduce a mechanism by which the state can advance without involvement from the committee.

After T_{(\text{fallback, enter})}

L2 slots where the Proven Chain has not advanced, based fallback mode is activated. In this mode, blocks can be published directly to the L1 without needing to involve the L2 committee i.e. you skip attestations.

The block validity requirements during based fallback mode are slightly different:

1. No need to obtain a quorum of attestations from the committee.
2. Anyone can submit a valid proposal.
3. The corresponding proof must come along with the proposal.
4. Relax the constraint that the slot of the block must be exactly the current slot.

We need 1) because assumably, the committee is non-cooperative or is failing to propose blocks to the Pending Chain. We need 2) because it gives us 1/N trust assumptions. We need 3) because otherwise there's no guarantee that the data required to eventually prove the block has been made available. We need 4) because block proving times could be significantly longer than slot times (36s).

We get out of based fallback mode under any of the following two conditions:

1. T_{(\text{fallback, exit, activity})}

time has passed without anyone advancing the Proven Chain.

1. T_{(\text{fallback, exit})}

total time has passed since we entered based fallback mode.

With respect to T_{(\text{fallback, enter})}

, it must be at least twice greater than the Epoch length otherwise based fallback could be triggered before a valid proof could land on L1. In practice, we think it should be larger in order to reserve based fallback for cases where multiple committees are massively censoring i.e. refusing to include a transaction from the forced-inclusion queue.

Forced inclusion queue

While based fallback is for ledger growth, the forced inclusion queue is for censorship resistance.

Simply put, the forced inclusion queue is a list of transactions that must be included by a proposer in a certain ordering before a certain time. Each transaction in the queue is in the form of:

struct ForceInclusionTx { bytes32 txHash; // hash of the associated TxObject uint256 includedByEpoch; // inclusion date bool included; // flag to indicate whether this transaction has been included or not }

To accomplish this, we extend the checks performed when blocks are proposed to make sure that transactions from the forced inclusion queue are included at the top of an upcoming block proposal.

This means that given transactions in the forced queue that are due (i.e. currentEpoch

= ForceInclusionTx.includedByEpoch

), a block proposal can only contain transactions from outside of the queue if they have included all due forced transactions.

[

Forced Inclusion Queue

3120×1728 297 KB

](https://europe1.discourse-cdn.com/flex013/uploads/aztec/original/2X/2/2730daae9affbc0df20a0f554ff60a437daf11c9.png)

In the above diagram, only the following block proposals are valid:

- [tx1]
- [tx1, tx2]
- [tx1, tx2, tx3]
- [tx1, tx2, tx3, tx4]
- [tx1, tx2, tx3, any other combination of transactions]

Any other permutation is considered invalid and won't be accepted.

This means that a censoring committee can propose blocks containing only a small number of transactions from the forced queue and nothing else in order to progress the chain (albeit slowly) without triggering based fallback mode. To get around this, the solution is to mandate a minimum number of transactions from the forced queue to be included for a block proposal to be deemed valid.

Optimizations

Lazy sequencing

We've built our sequencer client to post state diffs + a state root to the L1 to provide data ordering and availability. Recently there's been discussions to move to posting TxObjects

instead, which is the execution request object void of any state diffs or state roots.

This effectively decouples execution from ordering on the Aztec rollup. Proposers don't execute. They just pick transactions from the p2p layer, order and include them in a block proposal. The committee does not execute either. They only check that they have indeed verified the private kernel proof for all txs included in the block. The Pending Chain can advance fast.

This is sometimes called "lazy sequencing" and the benefits are:

1. Increased throughput (i.e. TPS) because sequencers, and more importantly committee members, no longer need to execute to verify the proposal's validity.

2. Reduced hardware requirements for participating in the attestation committee.

It is not without demerits. Anyone transacting who needs to know the state must ask for it from a node that has executed. This has privacy and UX implications.

Optimistic signature verification

Right now the signatures from the L2 committee are posted to calldata and verified by the rollup contract, increasing L1 overhead. An alternative is to not have the L1 contract verify signatures, and opt for optimistic signature verification.

We would save on L1 overhead but we would be reliant on someone verifying the signatures in the short period of time between the block arriving to the Pending Chain and being added to the Proven Chain.

FAQ

- **What are the Major changes since Fernet + Sidecar?**

- L2 Attestation Committee

In the new design, we introduce a new validator network where committee members sign off on proposals to enable faster build ahead. Thus proposers build a "Pending Chain", which later gets added to a "Proven Chain" once the associated proof has been verified on the L1.

The resulting effect is soft finality on Aztec L2 is achieved much quicker - currently 3 L1 slots or ~ 36s.

If the corresponding proof for an epoch of the Pending Chain (i.e. 32 slots) does not land within the allotted time, the Pending Chain is pruned to the tip of the Proven Chain and we restart growing the Pending Chain.

Leader election and computing the committee are described in theInitializing an Epoch section.

1. Prover Timeliness & Coordination

Proofs for an epoch N

must land within epoch N+1

or else the Pending Chain is pruned. Previously if a proof was missed, we trigger race mode and allow anyone to supply the proof. This was changed in favor of a "Based Fallback" mode which is triggered only after a number of unsuccessful tries to prove a Pending Chain epoch.

Previously the sequencer who submits the proposal must also secure a performance bond. We remove this requirement in favor of the first C

proposers from the next epoch. More details in theProof claim phase.

Prover coordination is still out-of-protocol but we add an optional mechanism where provers can submit quotes to proposers via the p2p layer or a json-RPC endpoint.

- **Why does the sequencer post the bond (and not the prover)?**

Such a setup would require an additional coordination step: provers would need to submit quotes (as they do now), but then proposers would need to indicate which quotes they were willing to accept.

Further, a prover could choose not to post the bond even if the proposer accepted their quote. Thus proposers may want to hedge their risks by approving multiple quotes from multiple provers who would then race to submit the bond to the L1 contract.

This wastes gas and could lead to worse outcomes (the winning prover is not necessarily the prover who submitted the best bid).

- **Why choose C**

to be 13 in the proof claim phase?

The value of C

should be chosen to minimize the chance that a censoring committee controls the first C

slots and therefore does not post the bond leading to a re-org of the Pending Chain. A higher value of C

could mean less proving time available so ideally we pick the smallest C

such that the probability of an attack is below a certain threshold.

The hypergeometric distribution describes the probability of k

successes, in n

draws from a population of size N

.

Assuming a total validator set of 10,000 validators, and with a maximum \frac{1}{3}

of those being malicious, we picked the smallest committee size K

where the probability of drawing more than \frac{2}{3}K

malicious validators is less than 1e^{-6}

.

That committee size is K=48

.

Then we picked the smallest C

where the probability of drawing C

malicious validators in a row from a population of K=48

validators, at least \frac{2}{3}

of them honest, was less than 1e^{-6}

.

That value of C

is 13

- **Does the proving mechanism encourage faster proving?**

The longest time to finality in the happy path is a transaction landing in block 1 of epoch N

and then a proof for epoch N

landing in the last block (i.e. slot 32) of epoch N+1

. This is approximately 38 mins which is quite long.

The current proving mechanism does not incentivize proposers to chose provers who are fast, just provers who complete proofs on time and for a competitive price. Incentivizing faster proving might lead to faster provers winning more often and therefore be able to undercut competitors. In other words, this is a potential centralization vector.

We’re interested in ideas for designs that 1) don’t encourage racing 2) decentralized 3) and can encourage provers to start proving early.

Reference Links

1. Original Spartan + Sidecar Request For Comments
2. Further reading on Lazy Sequencing from Celestia

Disclaimer