TL;DR:

We introduce Commitment-Satisfaction Committees

(CSC), an in-protocol solution to Protocol-Enforced Proposer Commitments (PEPC).

Thanks to Mike Neuder for developing Payload-timeliness committees and to Barnabé Monnot for the idea of using them to enforce proposer commitments.

Thanks to Terence, Thomas, Trace, Xyn, William X, Stokes, Francesco, Sam Hart, and Mike and Barnabé for their generous feedback.

[

Illustration of the Commitment-Satisfaction Committee discussing a vote on a block.

1024×1024 89.5 KB

](https://ethresear.ch/uploads/default/original/2X/9/95ed76f4efc7bc7203e20ee0a1b2db7663e493f0.jpeg)

This document solves one of the Ethereum Foundation's Robust Incentives Group (RIG) Open Problems: ROP-4: "Rewrite" vanilla IP-PBS in PEPC framework.

What is

this solution?

The Commitment-Satisfaction Committee

(CSC) aims to address some of the shortcomings of out-of-protocol solutions by introducing changes to the Ethereum protocol. In summary, for a given slot, a subcommittee of the attesters is constructed that enforces the proposer's commitments on the block.

Since PEPC would come after ePBS, we extend the Payload-timeliness Committee (PTC) to enforce proposer commitments. This is because PTC members already check for payload validity, and, in a PEPC world, a payload's validity is also contingent on whether the payload satisfies the proposer's commitments.

What are Payload-timeliness Committees

?

The Payload-timeliness Committee, introduced by Mike Neuder, is a design for ePBS that operates by having a dedicated committee of attesters responsible for ensuring that valid execution payloads

are released in a timely manner. Concretely, the PTC, which requires block-slot voting, decides whether a full

or an empty

block should be considered for a given slot, where the weight of the vote is considered in the next slot. The Commitment-Satisfaction Committee builds on this idea by generalizing these committees to additionally check for proposer commitments.

What is the motivation

behind this solution?

In addition to the previous reasons, this in-protocol solution allows us to leverage Ethereum's attesters to enforce commitments. This means that we don't need to bootstrap a new trust network with a corresponding amount of value at stake, as would be the case with, for example, PEPC-DVT.

This solution additionally allows the proposer to make credible commitments up to the same slot in which they are to take effect. This is important, because if commitments were introduced out of protocol, we may need to wait for them to be finalized to have certainty over whether they will be enforced.

Operational Mechanics

Commitments

A proposer commitment is represented by a relation, which can be defined extensionally by the list of the blocks that satisfy it or intensionally by a formula that we refer to as the characteristic function of the commitment. This article considers the second approach, where the function allows a third party to evaluate whether a block, passed as input, satisfies its corresponding commitment.

This function can be implemented by arbitrary logic in the EVM, and it would be offunction type, which means it can be broken down into an address and a selector. By means of these two variables, we store a “pointer” to this function in the container we use to represent a commitment in the consensus layer:

class Commitment(Container): address: ExecutionAddress selector: Bytes4

These commitments are included in a new container, specifically made for transporting proposer commitments on the network:

class ProposerCommitments(Container): data: ProposerCommitmentsData signature: BLSSignature

where ProposerCommitmentsData

aggregates the commitments for the slot.

class ProposerCommitmentsData(Container): slot: Slot commitments: List[Commitment, MAX_PROPOSER_COMMITMENTS_PER_SLOT]

The signature for an instance of ProposerCommitments

is verified by obtaining the public key of the proposer for the slot ProposerCommitments.ProposerCommitmentsData.slot

from the Beacon chain and using it to verify the signature ProposerCommitments.signature

.

How does the proposer make

commitments?

The proposer’s block includes a builder bid and the ProposerCommitments

that the full

block must satisfy. The CSC could also demand that the instance of ProposerCommitments

includes commitments made by the proposer earlier, possibly stored in an EVM contract, such asEmily’s.

The proposer has to propagate ProposerCommitments

at an earlier point, however. Specifically, the builder needs to be aware of these commitments by the time they build the block and provide a header to the proposer. Otherwise, the builder risks building a block that doesn’t satisfy commitments and is rejected by the CSC. We refer to this period as the proposer-commitments propagation phase

.

Payloads

In the payload, the builder includes a commitment to the instance of ProposerCommitments

they saw, which the builder ensures the block also satisfies. The builder checks so by iterating through ProposerCommitments.ProposerCommitmentsData.commitments

. For each, they call the characteristic function, passing as input the full block that would include the payload.

The trade-off of this approach is that, as pointed out byThomas and Trace, the amount of latency introduced is linear with the number of commitments in ProposerCommitments

. We address this in the section that discusses ways to control gas griefing.

Ultimately, the CSC checks that the ProposerCommitments

included by the proposer in the block coincides with the ProposerCommitments committed to by the builder in the payload. This condition ensures that the builder’s payload is never included in a block for a different instance of ProposerCommitments

than the one the builder ensured that the payload satisfied.

When is the payload released

?

Similarly to the PTC model, the builder releases the payload in the same phase as the attestation aggregation. The builder does not require a full view of the attestations to release the payload, and the release is conditional on the builder having not seen an equivocation of the proposer. An equivocation would correspond to the proposer having doubly-proposed or included in the block a ProposerCommitments

that isn't the one committed to in the payload. If the builder fails to release the payload in time (and the block turns out empty

), the payment still goes through, since the proposer fulfilled their end of the contract.

[

Timeline

3527×1557 354 KB

](https://ethresear.ch/uploads/default/original/2X/0/0403ee534181f32c573b6455e8657dd4bc658e92.png)

Commitment-Satisfaction Committee (CSC)

What

does the CSC vote on?

The CSC is made up of the first index in each slot committee, and, in addition to voting on the timeliness of the payload, they vote on whether the full block satisfies the commitments in the block's ProposerCommitments

. They earn full attestations for correct votes and are punished for incorrect ones, and their regular block attestations are ignored.

In the block, the proposer must include the previous slot's CSC attestations aggregated. As pointed out by Terence with respect to the PTC design, there's no reward for including incorrect CSC attestations.

When

does the CSC vote?

The CSC votes in the same phase as the PTC would, which we refer to as the commitment-satisfaction vote propagation phase

.

How

does the CSC vote?

CSC members first check that the instance of ProposerCommitments

included in the block is the one the builder committed to in the payload. Subsequently, they iterate through the commitments in ProposerCommitments

, ensuring that the block satisfies them all.

For checking whether a commitment is satisfied, CSC members leverage their local execution client, using the client's JSON RPC eth_call

to evaluate the commitment's characteristic function on the block.

If a check fails or a commitment is not satisfied, CSC members vote in favor of an empty

block for the slot. Otherwise, they vote for the full block.

Analysis

Properties

On top of the ones described in the PTC article, these properties are considered:

1. Honest proposer commitment safety:

the block made canonical is for the instance of ProposerCommitments

included in the block by the proposer.

1. Honest builder commitment safety:

the ExecutionPayload

made canonical includes a commitment to the same instance of ProposerCommitments

included in the block by the proposer. If the builder releases an ExecutionPayload

with a different commitment, the CSC votes in favor of an empty

block for the slot.

1. Outcome safety

: a block that violates a commitment in the instance of ProposerCommitments

it includes is not made canonical. If such a block is proposed, the CSC votes in favor of an empty

slot.

Non-properties

1. Honest-builder payload safety:

the builder can’t be certain that the block made canonical will be for the same ProposerCommitments

they committed to in the ExecutionPayload

, because there are no guarantees that the ExecutionPayload

will be included or that the block proposed will be for the same instance of ProposerCommitments

committed to in the payload. In the case of the latter, a different payload, which commits to the block’s instance of ProposerCommitments

, would be considered.

Behavior of the CSC

We now find the bounds over which the CSC behaves honestly, leveraging the same mathematical framework as in the PTC article. Concretely, we determine how much of the vote weight needs to be controlled by honest members for the CSC to behave in such a manner collectively.

Consider a slot N

where the CSC votes with weight W

. Additionally, consider a proposer boost of 0.4W

. Let C

represent the share of the CSC vote controlled by honest actors.

Honest Proposer Scenario

For an honest proposer in slot N+1

, we have that the following relation holds for the block in slot N+1

to not build on a block that violates commitments in slot N

:

weight(N+1, missing)>weight(N+1)

$C W+0.4W>(1-C)W$

$C>\frac{1-0.4}{2}=0.3$

Thus, at least 30% of the CSC’s weight must be controlled by honest actors.

Dishonest Proposer Scenario

If the proposer in slot $N+1$

is dishonest, we have instead the following relation:

$C W>(1-C)W+0.4W$

$C>\frac{1+0.4}{2}=0.7$

In this case, at least 70% of the weight of the CSC needs to be controlled by honest actors.

Comparison with PTC

[Barnabé](#) created this really useful table that compares ePBS with PTC to PEPC with CSC:

	ePBS with PTC	PEPC with CSC
Proposer commitments	Revealed execution payload hashes to h	Generalized commitments are satisfied by the execution payload
Which data does the builder know?	Parent root of the block they are building on	Parent root of the block they are building on + applicable proposer commitments
When do builders know the data?	Before proposer slot starts, they assume the proposer will propose on top of the same head they observe	Before proposer slot starts, builder must also receive applicable commitments from the proposer, before they are written to the chain
What if builder doesn’t know the right data?	Proposer doesn’t have a valid block to propose	Proposer doesn’t have a valid block to propose
How can the proposer equivocate?	Proposer makes two commit-blocks, committing to different exec payload deliverables	Proposer makes two commit-blocks, committing to different exec payload deliverables, or proposer broadcasts two different ProposerCommitments during the proposer-commitments propagation phase
What if the proposer equivocates?	Builder doesn’t have to release their payload	Builder doesn’t have to release their payload
What if builder doesn’t release payload/releases an invalid payload?	PTC votes for empty, builder still pays proposer	CSC votes for empty, builder still pays proposer

Potential Risks

Splitting

The same splitting scenarios discussed for the PTC apply to the CSC. No new splitting vectors are introduced because the instance of ProposerCommitments

the network makes canonical is propagated by

the block.

Gas Griefing

There’s a risk of a gas griefing vector given that the commitments’ functions implement arbitrary logic. These functions could consume excessive amounts of gas and grief CSC members. Additionally, since the builder has to check commitments to ensure the full block will satisfy them, the amount of time for checking these commitments should be minimal to introduce the least latency.

To control this risk, a limit for gas usage should be put for checking the entire set of commitments in ProposerCommitments

, regardless of the number of commitments it aggregates. If the limit is surpassed, we consider the check as having failed. This bounds the worst-case scenario of gas consumption. This limit would likely have to be standardized and could draw some inspiration from the EVM’s block gas limit. At the very least, it would have to strike a balance between allowing enough space for sophisticated commitments to be evaluated and not creating a significant computational burden for CSC members or the builder.

A gas-metering mechanism that subtracts from the proposer’s balance directly (so that it doesn’t use up the block’s gas) could also be considered as a means of making proposers internalize the social cost of their commitments. Since commitments are evaluated through the execution client by the CSC, the gas metering could be achieved by computing the difference at the start and the end in the values from the opcode gasLeft()

.

Future Directions

- Commitment satisfaction as a block validity condition

: As pointed out independently by Mike Neuder and Alex Stokes, it might make sense to consider having all validators instead of a committee check for commitments. The check would be embedded as a validity condition on the block, which may be preferable because it neither introduces an honest majority assumption on a committee nor a new phase, as happens under the CSC model. This implementation of sequencer commitments could serve as an example of block validity conditional on commitment satisfaction.

- Commitment verification efficiency

: Parallelization could be used for checking the commitments in ProposerCommitments

, since these checks are independent of each other and can be done in parallel.

- Aggregate phase for CSC votes

: If the committee’s size grows too large, an additional phase for aggregating their votes may need to be introduced.

- Real-world Applications and Use Cases

: Identifying and developing real-world applications and use cases enabled by the CSC model would be fundamental in proving the case for this solution.