Optimistic pool - a universal co-processor for Ethereum mainnet

Scaling through co-processors.

This post is devoted to the particular instantiation of the idea of the co-processor: a universal contract that saves users gas by doing some computations off-chain and validating them more efficiently on-chain.

This is an idea similar to roll-up, however, co-processors typically do not hold user's tokens, and instead amplify the performance of their main execution environment. These approaches, though, typically do break atomic composability.

There are some examples either already existing in the wild, or being built right now, so this idea is not entirely novel (though, different kinds of co-processors are a bit hard to categorize yet):

1. WAX(formerly BLS wallet), being developed by PSE, intends to aggregate user's transactions to save on signature validation cost.
2. Axiom's co-processor is an example of an universal zk-coprocessor providing Ethereum state proofs.
3. In the future, intent-based systems for swaps, like CowSwap and others can batch user's actions. They can even hold user assets in a rollup and perform an aggregated swap on mainnet. This is not yet done, but looks like a natural route of evolution (considering that isolated execution contexts fracture the liquidity).

Similarly to rollups, such solutions can roughly be categorized as being zk / optimistic, and by relative degree of centralization required for their operation.

In our (my and @curryrasul 's) work, generously funded by Nouns ⌐◨-◨ DAO, we needed a low-cost solution for checking zk-proofs in a private voting system. As decentralized enough zk-recursion is still a bit too hard, and the time delays are acceptable, we have decided to go optimistic route - with hope of switching to zk-recursive approach when it matures enough.

The system is quite universal, and a lot of applications can, potentially, plug into it, improving performance for everyone - as an example, modern versions of Tornado.Cash, like Privacy Pools, and various smart contract wallets can also benefit. It is currently under construction, but we are hoping to release a first version in the coming days.

We would like to thank Nouns ⌐◨-◨ DAO for funding this research,@gluk64 for useful suggestions on validator queue / avoiding introduction of a new token, and to PSE for supporting us in general.

Optimistic pool overview

A simplified exposition:

An optimistic pool is a contract allowing to:

1. Register new claim kinds
2. types of claims that can be checked by this optimistic pool. Claims typically should be immutable, or they are not guaranteed to be checked correctly, and checking them natively in the EVM should fit in a block gas limit. These restrictions must be enforced by a developer integrating a new claim.
3. Deposit a new claim

into a contract. After surviving there for some time (the challenge period), it becomes finalized

, and the contract integrating such claim can treat it as valid.

What's inside of a claim kind?

Claim kind is a contract address, and this contract is assumed to have a particular ABI: It should support function checkClaim(uint256 claim_hash, uint256[] advice) -> bool

, which checks the claim given its hash, and an additional advice (typically, an actual claim will be encoded as a keccak Merkle tree).

Also, claim kind should have a function depositClaim

implemented, as it is a contract which is authorized to deposit claims of its kind. The calldata of the depositClaim

should be enough to reconstruct the advice required to check it (or, post-EIP4844, blob-carrying transactions can be used

instead).

It should be noted that some ClaimKind-s might be adversarial - for example, their checker functions could output different results depending on who is calling them or some other external properties. There will be an option to not check adversarial claim kinds.

In this simplified exposition, user would need to deposit a collateral greater than the gas cost of validating the claim. This is inconvenient, so, actually, a bit more involved process is happening.

There is a special (permissionless) role called blesser

. Blessers are organized in a queue (which we will describe in details later), and they process claims in batches.

When new batch of claims is formed (which happens once every few hours), the blesser sends a “blessing”, which is a sequence of trits (0,1,2) - with 0 meaning that the claim is “cursed” (incorrect), 1 that claim is “blessed” (correct) and 2 that claim is “ignored”.

This sequence is serialized using base-3 encoding, which means that one uint256 fits in a blessing for a batch of at most 161 = floor(log_3(2^256)) claims (actually a bit less, because we will also put some additional data in a few leading bits).

The blessing can be challenged by anyone in one of the following conditions:

1. There is a claim which is blessed

, but its checker function returns False.

1. There is a claim which is cursed

, but its checker function returns True.

1. There is a claim which is ignored

, and there is a claim of the same kind

which is not ignored

.

Third requirement ensures that the blesser can not censor claims selectively - they must either check every claim of the same kind, or ignore this kind completely (this might be necessary if the checker function of a kind is adversarial).

Blesser priority queue mechanics

To ensure orderly processing and minimal amount of gas wars, we introduce a blesser queue. It is a queue, and in order to participate in it, user must provide BOND_VALUE_QUEUE collateral. This is a protocol level constant.

Each claim batch goes through the following phases:

1. Formation
2. Blessing phase
3. Challenge phase

During the blessing phase (which takes 2 hours), first hour only the 1st blesser can bless the batch. Next 20 minutes, 2nd blesser, next 20 minutes 3rd one. In the last 20 minutes, anyone (even not from a queue) can bless the batch.

In order to bless, any blesser (either from a queue or not) must provide a deposit of size BOND_VALUE_BLESSER - and this is not a protocol constant, but an adjustable value.

In a case that a blessing is defeated during the challenge period, their blessing bond is given to the challenger.

In a case that 1st blesser have failed to provide a blessing in first hour of the blessing period, their queue bond is burned (this is done to disincentivize spamming the queue and not showing up).

As this contract must not have any governance, we suggest the following automatic scheme to adjust the various protocol values:

BOND_VALUE_BLESSER = 300*10*6 * avg_gas_cost

where avg_gas_price is computed over the previous 12 hours.

MAX_TIP_FACTOR

- a parameter. On the registration of the blesser, it locks in the TIP_FACTOR

for the claim batch, ensuring that TIP_FACTOR <= MAX_TIP_FACTOR

; and the fee for the claim batch determined as TIP_FACTOR * BOND_VALUE_BLESSER.

if QUEUE_CURRENT_LENGTH > 20 : MAX_TIP_FACTOR *= 0.98

if QUEUE_CURRENT_LENGTH < 5: MAX_TIP_FACTOR *= 1.02

If a queue is empty, MAX_TIP_FACTOR is set to a relatively high 0.0003 (1/3000), and if there is an empty blessing (of an empty ClaimBatch), the MAX_TIP_FACTOR *= 0.5 (and up to the relatively small minimum of 0.00003 (1/30000)).

The scheme described above has a particular (while unlikely) failure mode scenario. It looks as follows:

The gas price suddenly spikes ~30 times and stays there (different claim kinds are affected at different ranges, this assumes the particularly huge claim requiring 10M gas to check). Our gas price oracle fails to adjust, which makes challenging claims economically irrational.

Therefore, we suggest the following simple defense mechanism: 10% of all the fees collected for a particular ClaimKind are put into a small automatic treasury (separate for each ClaimKind) and are effectively burned. This treasury can be used to fully compensate the gas cost of the successful challenge transaction in the second half of the challenge period

.

This ensures that for relatively popular ClaimKind-s this occurrence is unlikely to lead to the submission of the wrong proof. It potentially opens the door for validators looting the treasury in extreme scenarios (by blessing wrong proofs and then challenging themselves), but this requires a large-scale collusion between validators (as they would need to ensure beforehand that they would be able to challenge themselves and compensate gas cost) and is likely very limited in scale (as this is limited to the priority fee).

Choosing time periods

This seems to be largely ad-hoc for most optimistic rollups (we, at least, couldn't find any rigorous analysis on why fraud proofs require a week for Arbitrum). One of the relatively advantageous properties of our design, compared to normal challenge games, is the fact that our design is essentially 1-round.

That makes it easier to estimate requirements:

Imagining our adversary controls 66% of Ethereum stake, their probability to control 81 consecutive block is less than 2^{-128}, which is a bit less than 17 minutes. The more scary scenarios, therefore, come not from validator censorship itself, but some sorts of system degradation / extreme congestion. We decided that arbitrarily increasing this period and setting CHALLENGE_PERIOD = 6 hours

is likely enough to thwart such treats (and even potentially coordinate social recovery in case an unforeseen issue arises).

BLESSING_PERIOD = 2 hours

, it could be smaller, but this is convenient for blessers (who have lesser probability of missing their turn and losing their stake).

BATCH_FORMATION_PERIOD

is adjusted dynamically using the following rule: if the previous batch is of size < 100, it grows by 10%, and if it is >155, it shrinks by 10%. As an additional requirement, can never be more than 8 hours

.

Blesser payout mechanic

For an individual ClaimBatch, all blessers that have survived the challenge period are ordered by the time of their blessing. An individual blesser #k

is entitled to fee from all non-ignored Claim-s that were ignored by the previous blessers.

We devise a lazy gas-optimized solution for this problem.

1. Every blesser must also, as a part of the blessing, send total amount of non-ignored Claim-s. If it is incorrect, it can be challenged normally during the challenge period. This ensures that we know the fee that the first blesser is entitled to.

2. Every other blesser must deposit withdrawal claim into an optimistic pool as a special ClaimKind, in which the claim is a bitstring declaring to which Claim-s this blesser is trying to withdraw the tip for. It can be challenged by providing the following witness: a Claim that this user is trying to withdraw the tip for, and an earlier unchallenged blessing that also did not ignore this Claim.

Comparison with optimistic roll-ups / possible applications / conclusion.

Compared to normal optimistic roll-ups, our system enjoys a high degree of decentralization - it has decentralized fraud proofs from get-go. This is possible due to the fact that instead of considering multi-round games, we consider a parallel composition of 1-round games; which drastically reduces the requirements for collateral.

As possible applications we would like to list

1. On-chain private voting (and zk-proof checker in general).
2. Tornado-cash style contracts using zk-proofs (like Privacy Pools).
3. Smart contract wallets with complex custom logic.
4. Optimistic aggregation of signatures.

We are hoping that in future we will be able to build zero knowledge proving pools achieving similar degree of decentralization, and even more robust and safe performance.