OVERLOAD: THE OMNICHAIN RESTAKING LAYER

`6e6b5fb9371e92c64a49aa135bff2bef4e1fb0eb`

Overload Labs.

`@overloadfinance`

June 13, 2024

Abstract

As blockchains have proliferated and the total assets available onchain has increased, the opportunity to providing security and validation has become increasingly favourable. Both blockchains that are close to genesis and applications that have strong validation requirements could make use of such a value proposition, and it is a feature that "restaking" can fulfill.

As the restaking field has progressed, we at Overload Labs have devised a new approach in how restaking can be architected, furthering the scope of the restaking paradigm. We introduce the Omnichain Restaking Layer, where any asset from any chain, can be utilized to secure Actively Validated Services (AVSs).

In this paper, we will present and detail the Omnichain Restaking Layer and the validator approach to restaking. There are numerous different configurations which one can implement when designing such a primitive, and we will discuss the various trade offs in each approach.

1 Introduction

1.1 Prior Work

With the innovation of the restaking primitive, introduced by the seminal work of Eigen Labs Team [1], it has become possible to utilize assets for validation and bootstrapping of Proof-of-Stake (PoS) based consensus blockchains.

The Ethereum network as of today, is being secured by staked Ether and validator nodes. The core idea presented by the Eigen Labs team is how both native Ether and liquid staked derivatives (LSDs) such as e.g. stETH can be further staked to secure networks besides Ethereum itself, effectively bootstrapping the Proof-of-Stake validator set and total value-at-stake. As an L1 blockchain, the more value-at-stake securing your chain, the better, as it increases the difficulty of performing an attack on the consensus.

Overload presents a different a different approach from EigenLayer, described in this paper. EigenLayer takes on an operator-centric model to restaking, reducing flexibility in restaking congifurability, although on the other hand provides a simplified approach to slashing of operator and user stakes.

The gossip layer and algorithm for reaching consensus on AVSs also differ. EigenLayer mainly relies on peer-to-peer networking to propagate attestations in the network. This relies on a centralized "aggregator" (the name stems from the role of aggregating BLS signatures) to gather individual signatures from all the operators restaking to the AVS. For reaching consensus, the aggregator has the task of thereafter posting the signatures onchain on Ethereum Mainnet for validation using the Ethereum Virtual Machine (EVM). Given the EVM does not support `bls12-381` onchain [2], AVSs instead uses the `alt_bn128` curve (now called `bn254`) in the operator-centric model. Note that the Ethereum network coupled with the EVM will ensure that bytecode is executed correctly. Hence, BLS is not a strict requirement for reaching consensus and instead is an gas optimization for validation.

1.2 Background

Overload aims to provide an architectural generalization of the restaking primitive, presenting a minimal and simplified model that is loosely dependent on the fact that blockchains are expected to scale to order of magnitudes of higher transactions per second (TPS).

In this paper, we will modularize all the parts of the restaking primitive, describing changes in how restaking can be thought from a different point-of-view. For example, the gossip layer/networking layer can be both offchain or onchain, and the same applies for the algorithm for reaching consensus - which is a spectrum from centralized to decentralized.

We will present both a multichain implementation of The Omnichain Restaking Layer as well as formalizing previous work drafted at Overload Labs, with the initial reasoning behind it.

2 Mechanism

2.1 Restaking Model

As of now, there are two primary approaches to implementing a restaking primitive between asset holders and node providers - the operator model utilized by projects such as EigenLayer, and the validator model presented by Overload.

2.1.1 Operator Model

The operator model works such that asset holders decide which operator they restake to, but do not decide which AVSs should be validated, which is instead controlled by the operator.

Let n be the total number of available AVSs that can be restaked to. Then, to cover all possible configurations of restaking combinations, you would need $2^n - 1$ different operators, excluding the operator case where no AVSs are being validated.

We believe this tradeoff to be sufficient in a controlled environment, but would be deemed insufficient for restaking needs for larger n. As an example, if $n = 256$, then the amount of operators required would grow astronomically and it would be more difficult to provide a tailored experience for the end user. A solution would be for users to solo stake, or for there to be a service that spawn solo staking operators, where a new operator is spun up and configured for a single individual's needs.

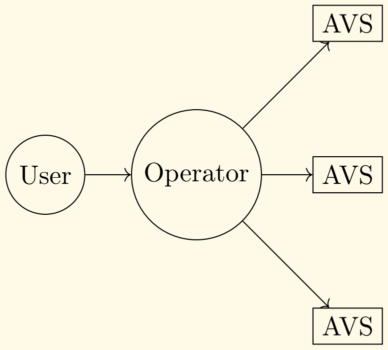

Figure 1: The Operator Model

2.1.2 Slashing

A feature that is enabled and simplified by operators is slashing. Assume that an operator commits an infraction and will be slashed that the restaked assets are pooled into an ERC-4626 vault, then slashing the operator would be equivalent to slashing all restakers.

This is largely possible because the operator model that ensures that all restakers are restaking to the same AVSs, and thus additional checks are not required for slashing.

2.1.3 Validator Model

In the validator model, a user restakes directly to a validator for each AVS they want to provide cryptoeconomic security to, and the combinatory problem is side-stepped. A user can restake to AVS in any configuration of their choosing.

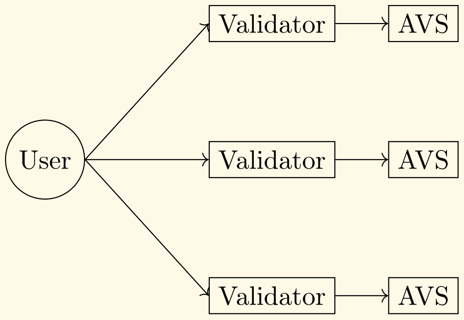

Figure 2: The Validator Model

2.1.4 Inability to Slash

Let V_i be a validator for an AVS, n_i be the total number of users restaked to the validator, and let r_{ij} be the number of restaked balances for the j-th user in the i-th validator.

To slash a validator, the validator would then need to check all the restaked balances for each token, for each user. Since the number of restaked balances for each user r_{ij} is variable and independent of n_i, we sum the total number of restake objects across all users to determine the overall complexity.

Therefore, the operation of slashing a validator scales with the total number of restake objects, which can be expressed as:

$$O\left(\sum_{j=1}^{n_i} r_{ij}\right).$$

Given the potential variability and the large number of possible balances, performing this operation onchain would result in prohibitive gas consumption. Consequently, we mitigate this issue by avoiding slashing through onchain consensus in the case of Overload.

2.2 Accounting

2.2.1 ERC-6909 Accounting

Upon depositiing tokens into Overload, ERC-6909 tokens [3] are credited to the user where the token id is `uint256(uint160(token))`. The operation is safe as an `address` type in Solidity takes up `160 bits`, while `uint256` is `256 bits`. We note that not all `uint256` ids are valid token addresses, and `revert` in the case of an invalid `id`.

2.2.2 Non-compliant ERC-20

Overload supports fee-on-tranfer ERC-20 tokens, but not rebasing ERC-20 tokens. We suggest wrapping rebase tokens to be non-rebasing before depositing them into Overload.

2.2.3 Bonding

Let u be the ERC-6909 token balance for a user and let b be the bonded balance of the token balance. Then, $u + b$ should always be equal to the principal ERC-20 token balance.

The u unbonded balance can be withdrawn from Overload at anytime without delay, while the bonded balance b needs to be unstaked from (c, v) pair(s) until the balance becomes unbonded (subject to potential withdrawal delays).

Let c be a consensus address, v be a validator address and let (c, v) as a consensus-validator pair. Then, the u balance is decreased and the bonded balance b is increased when tokens are restaked to as a (c, v) pair. When tokens are unstaked from a consensus-validator pair, bonded tokens can be unbonded and returns to being ERC-6909 balance.

In the table below, we will describe the process of bonding and unbonding token balances, where $\mathcal{M}$ is $max(d_0, ..., d_n)$ and d_i is a restaked balance to a (c, v) pair, including the latest restaked balance, and define δ as $\mathcal{M} - b$.

Function	Amount	Context	Unbonded	Bonded
`delegate`	$r > \mathcal{M}$	-	$u := u - \delta$	$b := r$
`delegate`	$r <= \mathcal{M}$	-	-	
`undelegating`	$r \leq \mathcal{M}$	With delay	-	
`undelegating`	$r = \mathcal{M}$ and $\exists! i\ (d_i = r)$	No delay	$u := u + \delta$	$b := max(\{d_0, \ldots, d_n\} \setminus \{d_i\})$
`undelegating`	$r < \mathcal{M}$	-	-	-
`undelegate`	$r = \mathcal{M}$ and $\exists! i\ (d_i = r)$	-	$u := u + \delta$	$b := max(\{d_0, \ldots, d_n\} \setminus \{d_i\})$
`undelegate`	$r < \mathcal{M}$	-	-	-

Figure 3: Table of Token Bonding

2.3 State

When tokens are restaked to a (c, v) pair, they can transition through different states upon invoking state transition functions. The states possible are None, Delegation and Undelegation—with the state transitions being `delegate`, `redelegate`, `undelegating` and `undelegate`.

State	Input	Next State
-	`delegate`	Delegation
Delegation	`redelegate`	Delegation
Delegation	`undelegating`	Undelegation
Undelegation	`undelegate`	-

$$\begin{tikzcd} S1 \arrow[r, "a"] & S2 \arrow[l, bend left, "c"] \arrow[r, bend right, "d"'] \arrow[loop above, "b"] & S3 \arrow[ll, bend left=50, "e"] \end{tikzcd}$$

S1: None
S2: Delegation
S3: Undelegation
a: `delegate`
b: `redelegate`
c: `undelegating`
d: `undelegating_delay`
e: `undelegate`

Figure 4: State transitions

2.3.1 Jail

Let v be a validator, c be a consensus contract, t be the jailtime and cd be the cooldown. Then, a consensus contract can jail a validator from calling `undelegating` for t seconds.

After t has passed, the validator enters a cooldown period of cd seconds, where it can neither be jailed and should not be able to rejoin the validator set for the (c, v) pair. The cooldown period both serves as a time to fix potential issues with the validator node as well as preventing an attack vector in continuous jailing, which can soft-lock assets inside Overload. Assuming a consensus contract c keeps calling `jail` on a validator at any opportunity that it can, then a validator v has cd seconds to call *undelegating* inbetween every `jail` call.

The purpose of jailing mainly serves as an alternative to slashing validator stakes. As the inactivity leak is the weakest possible offence in PoS consensus, we think jailing of validators as a sufficient punishment. AVSs can implement a stricter liveness requirement as well to increase the purging of inactive nodes to prevent potential halting of validation.

2.4 Gossip

The networking between validators and a sequencer can be generalized and modularized. For example, the networking does not have to strictly be peer-to-peer between the validators and the sequencer, and can instead be performed through other networking technologies.

2.4.1 Peer-to-peer

For example, EigenLayer utilzes peer-to-peer communication to send BLS signatures from operators to the aggregator [8]. Implementing p2p networking comes with varying potential complications, such as ensuring signature propagation or preventing DDoS attacks.

Sending signatures peer-to-peer also implies that infractions such as double signing needs to be taken into account, as there's no single source of truth to keep track of what an operator has and has not propagated in the network.

The advantage of being peer-to-peer is that operators do not need pay gas fees when validating as signatures are sent peer-to-peer to the aggregator. It is then the aggregator's job to verify the signatures offchain and eventually post them onchain in an aggregated BLS verification.

2.4.2 Data Availability

Data availability (DA) networks can be viewed as an intermediate solution to a gossiping layer. In this setup, operators or validators post their attestations on the DA network, allowing the aggregator or sequencer to gather these attestations directly from the chain.

While we acknowledge that DA networks provide strong guarantees, their lack of an EVM limits their ability to handle infractions like double signing. Therefore, we propose that an L2 solution would be more effective than a DA network in this context.

2.4.3 L2

Using an L2 for gossiping addresses both data propagation as well as prevention of double signing in the network.

Operators/validators and aggregators/sequencers both read-write directly with an L2, and there would not be any additional infrastructure required for the setup. The gossiping and general throughput is effectively capped by the performance of the settlement L2 itself.

2.4.3.1 Scaling with finality and TPS

If the AVS being validated demands faster finality, relying on an L2 might not suffice as the settlement chain for the AVS. Conversely, if the settlement chain offers single-slot finality with high transaction per second (TPS) throughput, it would enhance the performance of a restaking primitive that settles on that chain.

2.5 Hooks

In this section, we describe how hooks are critical for a permissionless restaking primitive, enabling any developer to build AVSs on top.

Hooks in Overload are similar to Uniswap V4 hooks [5], but they differ in certain details to achieve a simpler and more developer-friendly implementation.

2.5.1 Overview

Hooks are function callbacks. When a user calls `delegate`, for example, there are two hooks involved: the `before` and `after` hooks for the function. These two callbacks enable flexible composability when integrating with Overload.

For the hook calls, we implement them using the `abi.encodeCall` function at the code level instead of `abi.encodeWithSelector`. This approach helps avoid potential type or parameter errors that could occur.

2.5.2 ERC-165 Interface

For callback detection, we use the ERC-165 standard [4], unlike Uniswap, which employs permission flags in hook addresses to decide whether a callback should be triggered. Our choice is primarily driven by simplicity, rather than gas optimization. Although permission flags reduce gas costs, they significantly increase deployment complexity.

On L2 blockchains, gas costs have been significantly reduced through the development of EIP-4844 and DA solutions, which store calldata more inexpensively. As such, we consider using ERC-165 to be a good choice, as it is an established standard commonly used in other EIPs and ERCs.

We maintain the pattern of having hooks return the correct `bytes4` method identifier to determine whether a hook succeeded or not—and where any other value indicates a failed call.

2.5.3 Strict Mode

Strict mode is a `bool` parameter for all transition functions `delegate`, `redelegate`, `undelegating` and `undelegate`.

When `strict` is `true`, any failed hooks calls will propagate up—and when `strict` is `false`, failed hook call are ignored and code execution would continue.

Strict mode combined with a gas budget prevents soft-locking of assets when users restake to an unknown or unaudited consensus contract. Even if all hook calls are forcefully reverted, users can still unstake and withdraw their assets from Overload.

To prevent users from bricking or misaccounting token balances within a consensus contract, this can be effectively managed by requiring `strict` to be set to `true` for the `delegate` function, while making it optional for all other transition functions. The gas budget will ensure that `redelegate`, `undelegating`, and `undelegate` functions behave correctly, thus maintaining proper operation and accounting.

2.5.4 Gas Budget

Let g be the gas budget. Then, for each hook call, the max possible gas to be consumed is g, where anything above g is considered undefined behavior. By default, the gas budget is $2^{20} = 1,048,576$, though it can be adjusted to a lower value by the consensus contract.

3 Omnichain

In this section, we introduce the idea of an Omnichain Restaking Layer, where any asset from any chain can be used to secure AVSs.

3.1 Messaging

A central component to omnichain working in practice is the requirement of a sufficiently decentralized messaging protocol integrated with the consensus contracts.

When a user restakes to a specified consensus contract using `delegate`, both `beforeDelegate` and `afterDelegate` will be called. Within these hooks, the consensus contract can invoke the LayerZero endpoints [7], which send crosschain messages about restaked balances to the settlement chain. We distinguish between a messaging consensus contract and the settlement consensus contract—where the former is a subcensus contract and the latter a consensus contract. Subcensus contracts ensure that restaking parameters are relayed correctly, while the consensus contract ensures the settlement of validated chunks.

Subcensus contracts are rarely the limiting factor for high-performance validation; instead, the settlement chain is more crucial. While there might be delays in messaging protocols, such as LayerZero, these are negligible compared to the delays exhibited by native L2 bridges.

3.2 The Restaking Rollup

As the workload for onchain consensus settlement increases, launching a separate L2 could be advantageous for a restaking primitive. On a separate L2 blockchain, AVSs would not need to compete with other existing applications for blockspace and would benefit from a higher TPS and lower gas fees.

Given an omnichain restaking layer and sufficiently decentralized crosschain messaging platform, ecosystem composability for the settlement chain does not matter. Restaked balances from various chains would be mirrored on the restaking L2, and attestation power would be reflected accordingly. Hence, a restaking layer does not require an ecosystem for proliferation, as it would strictly be a settlement chain where only AVS operators and sequencers directly perform read-write operations.

When validity proofs and ZK-Rollups become predominant in the industry, a restaking rollup would also benefit from their technical advantages.

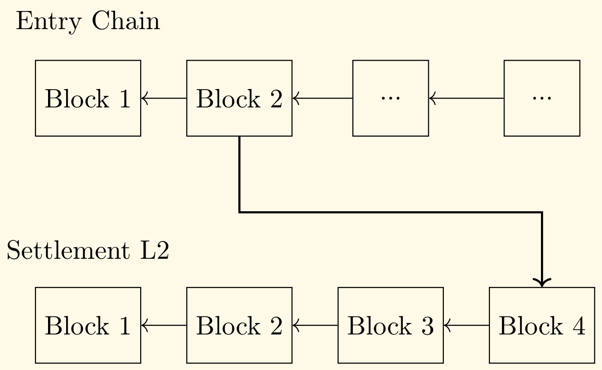

Figure 5: Sending a message to the Restaking Rollup

3.3 Checkpointing

When a state transition is called, function parameters are sent to the consensus contract. Inside the consensus contract, balances are checkpointed according to `block.number`, providing accurate accounting at the time of validation for attesting chunks.

To look up the historical balance of an address, we implement a binary search function, which has an overall time complexity of $O(\log n)$.

3.4 Chunk

A chunk is an onchain or offchain data structure that validators need to attest. The specific implementation of a chunk is determined by the developer.

If the chunks are to be posted onchain, ensuring this is the sequencer's responsibility. The term "sequencer" is borrowed from L2 terminology, reflecting the similarity in roles.

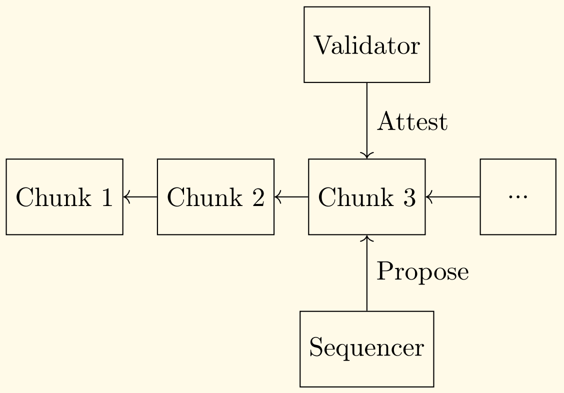

Figure 6: Chain of Chunks

3.5 Attestation

By moving attstations onchain, there's a single source of truth on whether a validator has attested on a chunk or not, and whether they accepted or rejected the chunk.

3.5.1 Bitmap

On each attestation from a validator, a common implementation for keeping track of attestations is through a `uint256` bitmap. By saving each attestation as a single bit inside the `mapping(uint256 => uint256)` mapping, we can utilize a `solady` library called `LibBitmap.sol` [6] to check the liveness of a validator.

For example, by calling `LibBitmap.popCount(bitmap, start, amount)`, we can check the liveness for `amount` of chunks starting from `start`.

Let s be the start bit, a be the amount of bits to check, and p be the `popCount` function. Then, we can check the liveness requirement by calculating:

$$\frac{p(s, a)}{a}.$$

References

[1] EigenLayer Team. *EigenLayer: The Restaking Collective*, `https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf`, Accessed: 2024-06-12.

[2] Alex Vlasov, Kelly Olson, Alex Stokes, and Antonio Sanso. *EIP-2537: Precompile for BLS12-381 curve operations*, `https://eips.ethereum.org/EIPS/eip-2537`, Accessed: 2024-06-12.

[3] JT Riley, Dillon, Sara, Vectorized, and Neodaoist. *ERC-6909: Minimal Multi-Token Interface*, `https://eips.ethereum.org/EIPS/eip-6909`, Accessed: 2024-06-12.

[4] Christian Reitwießner, Nick Johnson, Fabian Vogelsteller, Jordi Baylina, Konrad Feldmeier, and William Entriken. *ERC-165: Standard Interface Detection*, `https://eips.ethereum.org/EIPS/eip-165`, Accessed: 2024-06-12.

[5] Hayden Adams, Moody Salem, Noah Zinsmeister, Sara Reynolds, Austin Adams, Will Pote, Mark Toda, Alice Henshaw, Emily Williams, and Dan Robinson. *Uniswap v4 Core [Draft]*, `https://github.com/Uniswap/v4-core/blob/main/docs/whitepaper-v4.pdf`, Accessed: 2024-06-12.

[6] Vectorized, outdoteth, estarriolvetch, and Raiden1411. *LibBitmap.sol*, `https://github.com/Vectorized/solady/blob/main/src/utils/LibBitmap.sol`, Commit: `d699161248fdb571a35fe12f4bd3077032f33806`.

[7] Ryan Zarick, Bryan Pellegrino, Isaac Zhang, Thomas Kim, and Caleb Banister. *LayerZero*, `https://layerzero.network/publications/LayerZero_Whitepaper_V2.1.0.pdf`, Accessed: 2024-06-12.

[8] Eigen Labs Tean. *Operator Overview*, `https://docs.eigenlayer.xyz/eigenda/operator-guides/overview`, Accessed: 2024-06-12.