Off-chain Node Operators keys storage

At Lido protocol, the cost of adding BLS key pairs by Validators and depositing ETH is around 180000-190000 gas per key. With gas price 50 gwei and ETH price ~$2700 it is ~$25 per key, so Lido spends ~$780 per staked 1000 ETH. These costs can be optimized by taking BLS key pairs off-chain.

TLDR: checkout the Merkle tree section and go directly to Results where you can see that Merkle 4096 looks like the optimal setup making the deposit process ~4.7 times cheaper.

Lido stake delegation process consists of several steps:

1. Node Operators generate locally a bunch of BLS key pairs and calculates deposit_data_root using withdrawal_credentials provided by Lido.
2. Node Operators uploads this DepositData to the smart contract and awaiting approval.
3. The protocol governance checks the public key uniqueness and that every deposit_data_root matches Lido withdrawal_credentials.
4. After key approval deposit bot (or anybody who collect a quorum of 2/3 of the signatures of the deposit security committee members) could start delegation of the 32 buffered ether chunk to the next validator because the contract has all data prepared on-chain. If there are no unused and approved key+signature pairs, no ether gets delegated.

I've created a prototype to compare the current "Naive" approach with different off-chain storage strategies. In the best case adding and depositing one BLS key pair costs 40753 gas. With gas price 50 gwei and ETH price ~$2700 it is ~$6 or ~$166 per staked 1000 ETH, which is ~ 4.7 times less than the on-chain storage approach. But remember that these results captured on a simplified prototype, actual protocol results could be slightly differ.

Off-chain keys storage should fulfill three conditions:

1. Only approved keys can be used for depositing.
2. Keys reuse should be prohibited. It can be implemented by storing a counter for used keys inside a smart contract and indexing all keys. Along with each key, you should store an integer index of the key.
3. Any number of keys can be used per one depositBufferedEther

call. This requirement allows implementing automatic funds balancing between Node Operators in a round-robin fashion.

Possible optimizations

The main idea of all optimization approaches is taking key pairs off-chain and passing them to depositBufferedEther

method. The smart contract only verifies that these keys were approved by DAO and hadn't been used before.

Simple batch

The most straightforward approach is to concatenate keys into one string and store the hash of this string inside a smart contract. However, it allows depositing only a fixed amount of keys stored in a batch. There is a workaround, but it still requires sending all batched keys inside a transaction. Such workaround limits the batch size reducing the efficiency of such a solution.

Merkle tree

We can generate a Merkle tree from several keys and store only Merkle root on-chain. Merkle tree allows using any amount of keys which it contains per one depositBufferedEther

call.

To prohibit keys reuse, the keys inside a tree have their index and, there is a counter inside the smart contract that increments and concatenated to the key before verification.

The keys are often used in a batch. Because we added strict order to the keys we can optimize the Merkle proof algorithm by reusing leafs and nodes of the Merkle tree. We only need to know the index of the first leaves in the Merkle tree slice.

Here is the "verify Merkle slice" implementation:

Python (simplified)

Python (memory-optimized)

Solidity

Let’s take a look at the example. Assume we have a tree storing 16 keys, and we already used the first five keys.

[

merkle1

2416×808 61.4 KB

](https://europe1.discourse-cdn.com/business20/uploads/lido/original/1X/26d724f708e60e1dc4e0b2238a0b7420abee59ca.png)

For every single key, we need a proof consisting of 1 leaf hash and 3 node hashes, so for five keys, it would be 4*5 = 20 hashes. But because we know that these five keys are going in a row, we can calculate and reuse some hashes (blue) and reduce the overall proof size to 4 (yellow) hashes.

[

merkle2

2342×834 66.5 KB

](https://europe1.discourse-cdn.com/business20/uploads/lido/original/2X/9/9cf91463d174aeb47e43d81f9425da7b63ad2474.png)

In the best case, if the slice’s length is a power of two, proof length can be even smaller.

Merkle + Batch

It was interesting to find out can we save even more by combining prior approaches and using batches as Merkle tree leaves, but such a solution shows worse results than just a bigger Merkle tree and it also has some limitations on the number of keys per one depositBufferedEther

call.

That’s why I am not recommending using this approach.

Results

The table below contains the gas cost for adding and depositing one key for every approach with different tree/batch sizes. Rows represent approaches, and columns show the number of keys used pre one depositBufferedEther

call.

[

merkle3

1920×730 247 KB

](https://europe1.discourse-cdn.com/business20/uploads/lido/original/2X/5/5d6d8685dd26dda6ecea69268df1149d73445d55.jpeg)

Full test results

It seems that the cheapest way is storing 4096 keys inside the Merkle tree and deposit them in batches of 128 keys. Merkle trees bigger than 4096 differ negligibly in terms of cost per key. It can make more sense to have only one Merkle tree per one Node operator. One Merkle root per Node operator scheme is easier to implement and manage, and it still can be cost-effective and requires fewer actions from DAO.

I would be happy to work on this and make a pull request intolido-dao repo but I need some input from lido DAO, community and developers first.