Abstract

Following the stateless client idea, we implement an efficient on-chain dynamic Merkle tree with

- on-chain inclusive verification;
- on-chain append / in-place update;
- O(1) cost in storage space;
- O(1) storage write cost for update / append operations.

Background

Static Merkle tree is widely used on-chain to verify the membership of a large group with a very low storage cost, e.g., Uniswap on-chain airdrop. Instead of uploading the airdrop info (address, amount) of all users on-chain, which can be extremely costly, the airdrop can significantly save the cost by

- storing the root hash of the tree on-chain
- verifying/distributing the reward for a user upon user's collection with an off-chain computed proof.

Further, the on-chain dynamic Merkle tree is gaining interest. Ernst & Young (EY) has developed an on-chain append-only dynamic Merkle tree (https://github.com/EYBlockchain/timber). It saves the storage cost of the tree by only storing "frontier" nodes instead of all the nodes of the tree, however, the write cost of append operation is O(log2(N)), which may consume considerable gas on EVM.

Basic Idea

Similar to the existing static Merkle tree, which uses proof to verify inclusion, the basic idea of the on-chain dynamic tree is to reuse the proof to update the root hash of the tree after inclusion verification. The steps of a tree update are:

1. Given leafIndex

, oldLeafHash

, newLeafHash

, oldRootHash

, proof

1. Calculate rootHash

with oldLeafHash

and proof

. If the calculated rootHash != oldRoothHash

, inclusion verification failed; otherwise continue

1. Calculate newRootHash

with newLeafHash

and proof

, where proof

is reused and newRootHash

will be the root hash of the tree after the update.

Note that only newRootHash

is written to blockchain so that the cost of space and write is O(1).

Applications

Merklized ERC20

The ERC20 standard can be modified to Merklize the tree of (account, balance). Any mint/burn/transfer operations will require Merkle proof. The applications of the Merklized ERC20 maybe

- On-chain voting - a governance proposal voting can cheaply take a snapshot of the ERC20 and count the vote on-chain based on snapshot instead of maintaining all history of ERC20 balance change (Compound) or off-chain snapshot.
- Remote liquidity mining - a contract on a remote chain performs airdrop/liquidity-mining of the local ERC20 users, where the ERC20 snapshots are periodically forwarded to another chain via decentralized oracle.

Example code can be found here:

github.com

QuarkChain/DynamicMerkleTree/blob/abe6c7ee8f2fef105649943d5e329e5f5e697f8d/contracts/MerklizedERC20.sol

//SPDX-License-Identifier: MIT pragma solidity ^0.8.0;

import "hardhat/console.sol";

import "@openzeppelin/contracts/token/ERC20/IERC20.sol"; import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol"; import "@openzeppelin/contracts/utils/Context.sol";

import "./DynamicMerkleTree.sol";

contract MerklizedERC20 is Context, IERC20, IERC20Metadata { mapping(address => uint256) private _balances;

```
mapping(address => uint256) private _indices1;

uint256 private _totalSupply;

string private _name;
string private _symbol;
```

This file has been truncated. show original