Strengthening Cryptobridges: Assessing Hashing Accuracy in Ethereum with a Focus on Merkle Patricia Trie

jistro.eth and ariutokintumi.eth

Last update: January 8, 2024

ABSTRACT

This document explores Merkle Trees, Ethereum block structures, transaction data, and the implementation of Merkle Patricia Tries, emphasizing successful pre-merge verification but noting challenges in post-merge scenarios.

Keywords: Merkle tree, PATRICIA Trie, Ethereum, Blockchain

CONTENTS

1 MERKLE TREE

Before delving into the topic, it is important to understand the concept of a Merkle Tree. A Merkle Tree is a binary tree of hash values where each leaf node represents a single piece of data or the hash of a piece of data. It is commonly employed to efficiently verify the integrity of large datasets. Invented by Ralph Merkle in 1979, it is an integral part of all blockchains. [1]

The most common hashing algorithm in a Merkle tree is SHA-256 (Secure Hash Algorithm 256), which is part of the SHA-2 algorithm family, producing a fixed-length, 256-bit (32-byte) hash value. The key features of SHA-256 are [2]:

- **Message Length**: The length of the readable text before it is encrypted should be less than 264 bits.
- **Digest Length**: The hash digest length should be 256 bits.
- **Irreversibility**: All SHA-256 outputs are irreversible by design. For each input, you have exactly one output, but not the other way around. Multiple inputs produce the same output. The output has a fixed size, but the input doesn't have size restrictions.

As seen in Figure 1, the tree has three parts: Leaves, Nodes/Branches, and a Root. For the use of every leaf, it represents a transaction. This gives us the transactions hashed to later concatenate with another hash. This process is done n number of times until we get the Merkle root.

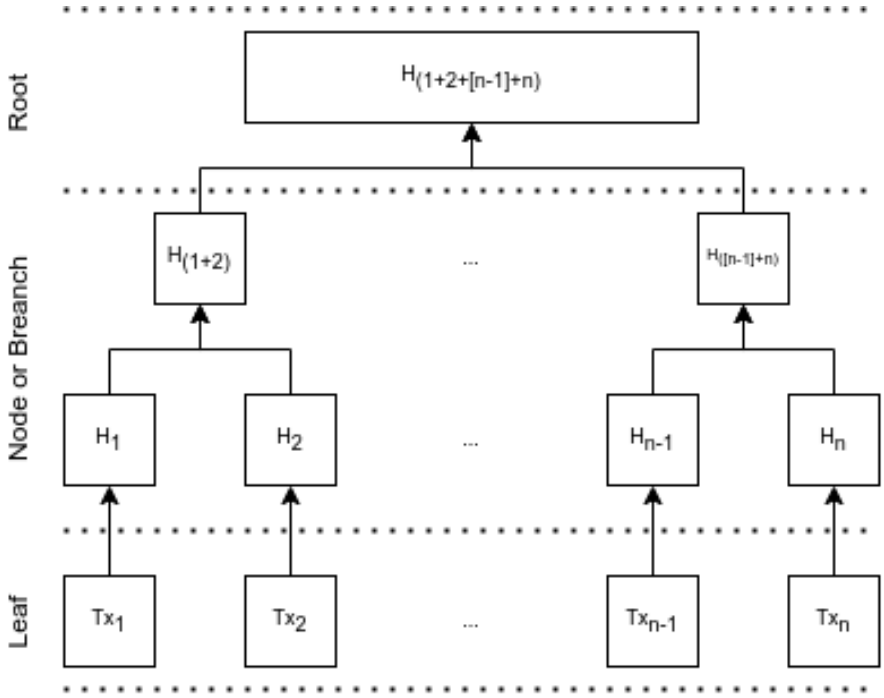


Figure 1. lustration of Merkle Tree

To verify if a transaction really happened in a block (as illustrated in Figure 1), for example, Transaction 1, we only need to verify with H_{1+2}, and by combining $H_{[n-1]+n}$ we get the Merkle root. If some data is not correlated with the hash, the root will be different than the result. This is what we call a Merkle proof. However, not only can we prove a leaf, but we can also prove a node or even the root itself.

2 THE ETHEREUM BLOCK

The structure of an Ethereum (ETH) block undergoes a change in "The Merge." As the name implies, "The Merge" involves combining the old blocks (execution layer) with the new Beacon Chain blocks (consensus layer), which operates on Proof of Stake (PoS)

Under PoS, an Ethereum block is comprised of three main parts [3]:

2.1 Administration

Is the the metadata of the block

- `slot`: The position of the block in the beacon chain blockchain (eg the 100,000 block is in slot 100000)
- `proposer_index`: The proposer who created this block
- `parent_root`: The root hash of a Merkle tree of the previous block
- `state_root`: The root hash of a Merkle tree which stores the state of the Beacon Chain (BeaconState)
- `randao_reveal`: Protocol-verified randomness, generated between all block proposers during an epoch. Randomness is critical to the Beacon Chain; security depends on being able to unpredictably and uniformly select block proposers and committee members.
- `graffiti`: An optional 32-byte field in which block proposers can put anything they want. Often used by mining pools to log their blocks.

- `signature`: The signature the block proposer creates to take responsibility (add to blockchain and collect reward if good, get slashed if bad). Created by combining the BeaconState, BeaconBlock and the proposer's private key.

2.2 Consensus

Is the layer coordinating the Beacon Chain providing the cryptographic security of PoS

- `deposit_root`: The root hash of a Merkle tree which stores the ETH deposits into the staking contract (required to become a validator)
- `deposit_count`: Amount of ETH in the staking contract
- `block_hash`: Hash of the block
- `deposits`: Amount of validator deposits which have been included in this block by the block proposer. Interestingly, the only non-0 value I could find was in the genesis block
- `voluntary_exits`: Withdrawals from the staking contract
- `attestations`: A list of all the signatures that attested to this block. Ethereum PoS elects a proposer who is charged with building (or selecting) a block and proposing it to the network. Attesters review the block and, if it's valid, sign it with their keys.
- `proposer_slashings`, `attester_slashings`: Validators that have performed a hostile action against the network. The network confiscates a portion of their staked ETH and ejects them from the validator set.
- `sync_aggregate`: Contains the aggregate BLS signature for the active sync committee. A sync committee is a group of 512 validators, randomly assigned once every 256 epochs (approximately 27 hours). This committee creates the signatures needed for an efficient light client.
- `sync_committee_bits` An efficient representation of committee participation.
- `sync_committee_signature`: The signature the sync committee creates to take responsibility for the block/epoch.

2.3 Execution

Is the data of the block, (almost) exactly mirroring PoW blocks.

- `baseFeePerGas`: Building on EIP-1559, the proposal establishes a minimum cost, known as the base fee, for each unit of gas in Ethereum transactions. This base fee is entirely burned with every block creation, and its value adjusts based on the fullness of the preceding block, allowing for a maximum change of 12.5% per block. Additionally, EIP-1559 introduces the concept of difficulty, which approximates the time a miner should spend calculating a hash function before discovering a block. $\frac{Network\ hash\ rate}{difficulty} = average\ block\ time$ It's important to note that these dynamics are no longer applicable under Proof of Stake, signaling a broader shift in the Ethereum consensus mechanism [4].
- `extraData`: An (optional) 32-byte field in which block proposers can put anything they want. Often used by mining pools to log their blocks. Similar to the graffiti in the consensus layer
- `gasLimit`: Total gas available to the block
- `gasUsed`: Gas used by the block
- `hash`: Hash of the block
- `logsBloom`: A Bloom filter is a probabilistic structure that allows a user to filter through each element in the block. minimizes the number of queries a client needs to make.
- `Miner`: The Ethereum address of the miner who successfully created this block
- `mixHash`: Intermediate value calculated from nonce, used for validation
- `nonce`: Extra data miners add to a block before hashing. A block is created when this hash matches a specific value; mining is the process of attempting to find this value by altering the nonce.
- `number`: Ethereum block number. Similar to slot in the consensus layer
- `parentHash`: The root hash of a Merkle tree of the previous block
- `receiptsRoot`: The root hash of a Merkle tree which stores the receipts created by the transactions in a block. A receipt includes: block number, block hash, associated contracts, gas used, the stateRoot at the time (before) transaction, etc.
- `size`: Size of the block in bytes
- `stateRoot`: The root hash of a Merkle tree which stores the entire state of the Ethereum Virtual Machine (EVM), account balances, contract storage, contract code, etc

- `timestamp` the date/time when the block was created, as reported by the block proposer
- `totalDifficulty`: The cumulative value of the difficult required to build the chain up until this block No longer applicable under Proof of Stake
- `transactionsRoot`: The root hash of a Merkle tree which stores the transactions contained within the block.
- `transactions`: An ordered list of all the transactions executed within the block.

2.4 Transaction data

Inside `transactions` written in blocks there are three main sections [5]:

2.4.1 Metadata

The metadata details of a transaction

- `blockHash`: The unique signature created by hashing the block the transaction is contained in.
- `blockNumber`: A unique, sequential number that identifies the block's position in the blockchain.
- `chainId`: Applying EIP-155 for simple replay attack protection, this field signifies whether the transaction is on the Ethereum chain or other EVM-compatible chains [6].
- `from`: Address (wallet or smart contract) the transaction is being sent from
- `gas`: Units of gas used by the transaction
- `gasPrice`: Amount paid (in WEI) per unit of gas for this transaction
- `hash`: Hash of the transaction
- `maxFeePerGas`: Maximum amount (WEI per gas) the user who created the transaction is willing to pay. Inclusive of base fee and priority fee
- `maxPriorityFeePerGas`: Maximum amount (WEI per gas) above the base fee the user who created the transaction is willing to pay. This fee will be paid directly to the miner/validator as a tip to incentive inclusion.
- `nonce`: Number of transactions sent from a given address. Once imprinted on a block, the wallet's nonce is increased. Protects against replay attacks
- `r, s, v`: Three values that form the signature of the user who created the transaction. They can be used to verify that the user authorized the transaction before it was executed in the EVM.
- `to`: Address (wallet or smart contract) the transaction is being sent to
- `transactionIndex`: Position of transaction within block
- `type`: There are two types: a new contract (0x0) and all others (0x2).
- `value`: Amount of ETH being transferred.

2.4.2 Cache

This section pertains to the **`accessList`**, which comprises a list of addresses and keys that the transaction expects to utilize. While the transaction retains the ability to utilize resources outside this list, it incurs a higher gas cost.

The introduction of the `accessList` was facilitated by EIP-2929, allowing clients to fetch and cache data for use during the transaction.

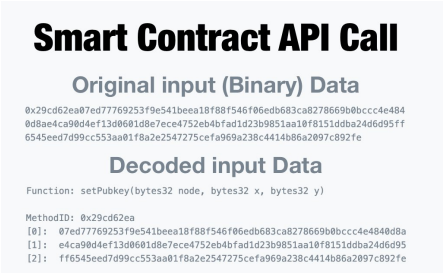
EIP-2929 brought about several key changes, including increased gas costs for state access opcodes, the introduction of `accessed_addresses` and `accessed_storage_keys` sets, and the concept of the `AccessList`. These changes collectively address historical issues related to opcode underpricing and potential attack vectors [7]. Following EIP-2929, EIP-2930 was introduced, adding a new transaction type that includes an access list, a list of addresses and storage keys that the transaction plans to access. While accesses outside this list remain possible, they incur higher costs [8].

As of now, there is approximately a 10% discount for using addresses and keys in the `AccessList`. However, it's crucial to note that this discount may increase in the future as Ethereum evolves to support light clients.

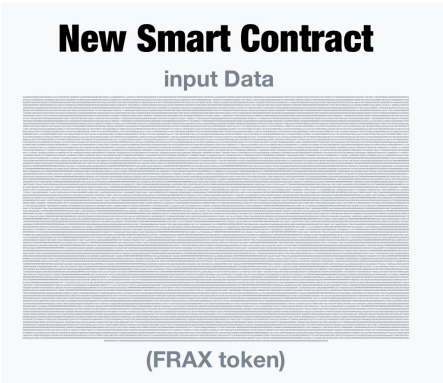
2.4.3 Data

This section delves into the transaction payload, encompassing either smart contract code or an API call. The data payload transmitted by the transaction serves diverse purposes and can be harnessed in three distinct ways:

1. ETH transfer: Nill payload (`0x00`).



(a) Input data in binary



(b) Input data in human-readable form

Figure 2. Input data [5]

2. Smart contract API call: Contains the name of the function and its parameters.

3. New smart contract: Includes the code of the smart contract.

While the data in the input field is recorded in binary (Figure 2b), it can be translated back into a human-readable form (Figure 2a).

It's crucial to note that the input field exists on-chain but is not part of the EVM state. Instead, it provides data for the contract to use during the transaction and is not tracked by Ethereum or used in consensus.

The EVM can only utilize the data supplied in that specific transaction; it does not have the ability to look back.

This property proves particularly useful for applications aiming to write historical data to the Ethereum blockchain (e.g., for manual retrieval later) but do not necessitate direct EVM access.

3 DEVELOPMENT

3.1 Merkle tree implementation

In the development process, we leverage acquired knowledge to construct the `blockHash` utilizing Merkle trees and Ethereum node providers in the prescribed order for accurate hashing [9]. The sequence of elements used includes:

1. `parentHash`
2. `sha3Uncles`
3. `miner`
4. `stateRoot`
5. `transactionsRoot`
6. `receiptsRoot`
7. `logsBloom`
8. `difficulty`
9. `number`
10. `gasLimit`
11. `gasUsed`
12. `timestamp`
13. `extraData`
14. `mixHash`
15. `nonce`

Despite employing this approach to gather block information, it has been determined that this method does not yield the correct `blockHash`. This is because Ethereum uses another type of tree structure, namely a **Merkle Patricia Tree**. This tree is a combination of a Merkle tree and a Patricia Trie.

3.2 PATRICIA Trie

Developed by Donald R. Morrison and published in October 1968, a Practical Algorithm to Retrieve Information Coded in Alphanumeric (PATRICIA) Trie is a special case of Radix Trie that follows three defined rules [10, 11]:

- Each node has two children, and all internal nodes of the tree have at least one child.
- A node is only split into a prefix with two children (each child forming a branch) if two words share the same prefix.
- Every word ending will be represented with a value within the node different than null.

Let's see a simple and graphical example based on the David Eisler video [11]:
We have this list of text in this order:

1. ROMANE
2. ROMANUS
3. ROMULUS
4. RUBENS
5. RUBER
6. RUBICON
7. RUBICONDUS
8. RUB

First, with "ROMANE," we group the characters into a single edge. In this case, we put the null terminator on the edge to indicate that it's a complete word. Next, we add "ROMANUS." Since "ROMAN" is common between both words, we branch off with "US," and the null terminator in the "E" with a null. Continuing the same process, we add "RUB." As seen in Figure 3, "RUB" has all the branches in common, so it is already in the tree. We add an edge with a null terminator to label "RUB" as a word.

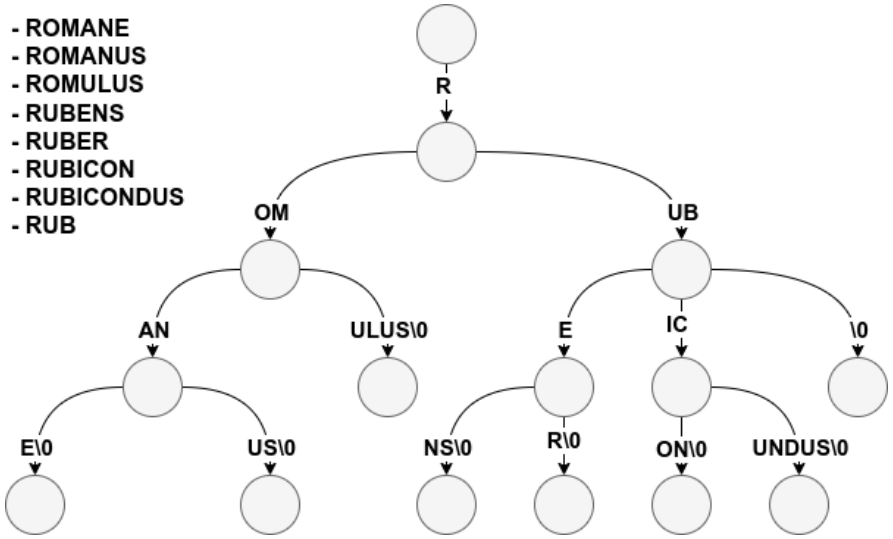


Figure 3. Patricia trie tree example from [11]

3.3 Merkle PATRICIA Trie

Merkle Patricia Trie (MPT) serves as a fundamental data structure within the Ethereum network, specifically designed for efficient state storage and retrieval. Much like the Merkle tree, every node in the MPT obtains a hash value through the sha3 hash of its contents. It's important to note that the keys and values stored in the MPT do not directly mirror the key-value pairs of the Ethereum state. Instead, the value stored in the storage represents the content of the MPT node, with the key being the hash of this particular node.

Key-values of the Ethereum state are represented as paths on the MPT, and nibbles are used as the unit to distinguish key values. Each node in the MPT can have up to 16 branches, with a branch node being an array of 17 items composed of 1 node value and 16 branches.

A node without a child is termed a leaf node, comprising two essential elements: its path and value. For instance, in the scenario outlined by Kim [12], if the key `0xBEA` contains 1000 and the key `0xBEE` contains 2000, a branch node with the `0xBE` path would be created. Underneath this node, two leaf nodes with paths `0xA` and `0xE` would be attached.

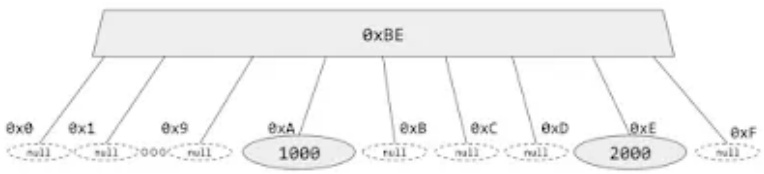


Figure 4. Example of MPT from [12]

The MPT encompasses three node types: leaf nodes, branch nodes, and extension nodes. A leaf node, consisting of a path and value, contrasts with a branch node, which manifests as an array of items. Extension nodes, serving as optimized nodes, compress branch nodes containing only one child. To distinguish between leaf and extension nodes, the MPT introduces a prefix to the path. For even-numbered nibbles in leaf nodes, `0x20` is added, while odd-numbered nibbles receive `0x3`. In extension nodes, `0x00` is appended for even-numbered nibbles, and `0x1` for odd-numbered nibbles, ensuring consistent byte representation of paths [12].

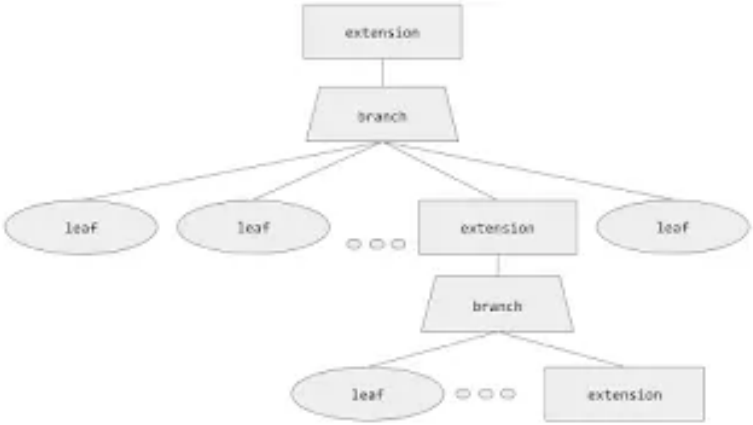


Figure 5. Structure of MPT from [12]

3.4 Merkle PATRICIA Trie implementation

Upon employing the correct implementation using in the examples from Gabriel Rocheleau [13] with the JavaScript library `ethereumjs/rlp` and `ethereum-cryptography/keccak`, we have successfully verified the accuracy of transaction and block hashes pre-merge. In this illustration, we utilized a legacy Ethereum transaction with the hash `0x39fbe5446a0be5cbf62c2b623b33bc3a93cf9ec04439218c45b5eafdea736de2`. For the block reference, we chose block `400000`. The results, as depicted in Figure 6, confirm the correctness of the hashing process.

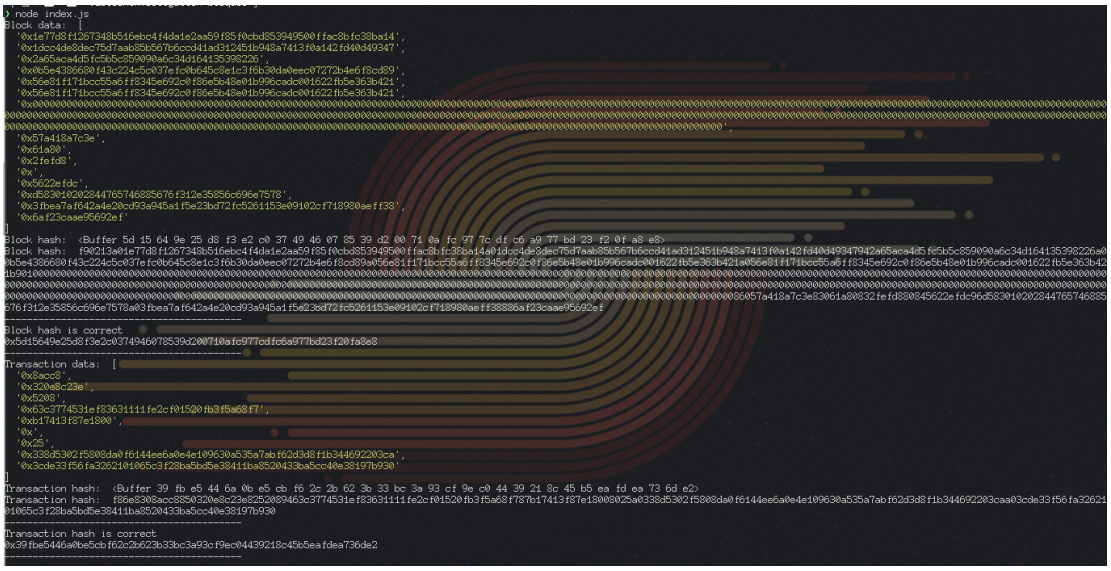


Figure 6. Output of the hashing process using pre-merge data

However, it's noteworthy that using transactions post-merge, such as smart contract creations or inter-

actions, and post-merge blocks provides an improper output. For example, we utilized the transaction hash `0x36855e78a4056dee5eee89ab8b56e89df27135195d23ed96b532f8bb8db99e78`, which corresponds to a type 2 transaction (interaction with a smart contract), and the block containing that transaction (`18903867`) as seen in Figure 7

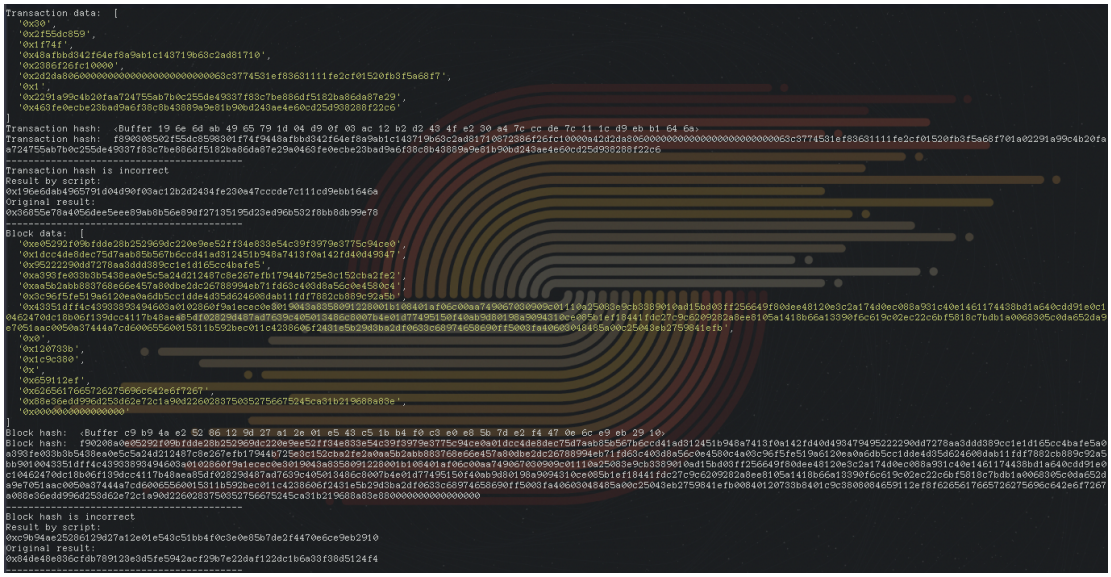


Figure 7. Output of the hashing process using post-merge data with the same process

This improper output happens because the data post-merge change so the hashing method has a different input [14] further investigation about how transaction and block hashing is made post-merge is required to get a correct output.

All scripts can be find in https://github.com/Roll-a-Mate/Research/tree/main/0002-Strengthening%20Cryptobridges/code

REFERENCES

[1] K. Kumar, "An Intro to Merkel Tree: What is it and How Does it Work? | HackerNoon," Jul. 2023. [Online]. Available: https://hackernoon.com/an-intro-to-merkel-tree-what-is-it-and-how-does-it-work

[2] D. Gitlan, "What Is SHA-256 Algorithm & How It Works," Apr. 2023. [Online]. Available: https://www.ssldragon.com/blog/sha-256-algorithm/

[3] R. Kirshner and H. Salomon, "Ethereum Block," Jan. 2023. [Online]. Available: https://inevitableeth.com/home/ethereum/blockchain/block

[4] V. Buterin, E. Conner, R. Dudley, M. Slipper, I. Norden, and A. Bakhta, "EIP-1559: Fee market change for ETH 1.0 chain," Apr. 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1559

[5] R. Kirshner and H. Salomon, "Ethereum Transaction," Jan. 2023. [Online]. Available: https://inevitableeth.com/home/ethereum/blockchain/transaction

[6] V. Buterin, "EIP-155: Simple replay attack protection," Oct. 2016. [Online]. Available: https://eips.ethereum.org/EIPS/eip-155

[7] V. Buterin and M. Swende, "EIP-2929: Gas cost increases for state access opcodes," Sep. 2020. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2929

[8] ——, "EIP-2930: Optional access lists," Aug. 2020. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2930

[9] AnuragP, "Answer to "Block header hash calculation"," Feb. 2019. [Online]. Available: https://ethereum.stackexchange.com/a/67334

[10] B. Osiek, "PATRICIA Trie's Nuts and Bolts," Feb. 2020. [Online]. Available: https://medium.com/@brunoosiek/patricia-tries-nuts-and-bolts-170a317b3ab6

[11] D. Eisler, "Trie and Patricia Trie Overview," 2013. [Online]. Available: https://www.youtube.com/watch?v=jXAHLqQthKw

[12] K. Kim, "Modified Merkle Patricia Trie — How Ethereum saves a state," Aug. 2018. [Online]. Available: https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd

[13] G. Rocheleau, "Ethereum's Merkle Patricia Trees - An Interactive JavaScript Tutorial," Aug. 2022. [Online]. Available: https://github.com/gabrocheleau/merkle-patricia-trees-examples

[14] eth, "Answer to "How are Ethereum 2 block hashes computed?"," Nov. 2023. [Online]. Available: https://ethereum.stackexchange.com/a/156158