Analyzing the Rationality of using Different Merkle Tree Constructions in Blockchain-based Accounting Systems*

Volodymyr Dubinin[1,†], Yaroslav Panasenko[1,†], Serhii Volynets[1,†], Viktoriia Zhebka[2,*,†] and Dmytro Nishchemenko[2,†]

[1] *Distributed Lab, Kharkiv, Ukraine*
[2] *State University of Information and Communication Technologies, 7 Solomyanska str., 03110 Kyiv, Ukraine*

Abstract

This paper provides a comprehensive overview of Merkle trees and their numerous extensions, which are fundamental data structures for ensuring data integrity and authenticity. Beginning with the foundational principles of k-ary Merkle trees, including their construction, membership proof generation, and verification processes, the article systematically explores a wide range of advanced variants. Key extensions such as Sparse Merkle Trees (SMT), Indexed Merkle Trees (IMT), Verkle Trees (VT), and Radix Merkle Trees (RMT) are detailed, alongside specialized implementations like the Merkle Patricia Trie (MPT) and Jellyfish Merkle Tree (JMT). The survey also investigates various optimization techniques aimed at improving storage efficiency, reducing membership proof size, and modifying the underlying logic. The paper concludes with a comparative analysis of these structures, evaluating their algorithmic complexities, trade-offs, and suitability for different applications, thereby serving as a guide for selecting the optimal Merkle-based construction for specific use cases like blockchain, cloud storage, and digital signatures.

Keywords

Merkle tree, data integrity, cryptographic hash function, blockchain, data auditing

1. Introduction

A Merkle tree was first proposed in [1] by Ralph Merkle for Digital Signatures. However, now it is used in a wide range of applications, allowing the creation of data authenticity proofs. A Merkle tree is a data structure constructed from hashes of various data blocks arranged in layers in a tree. The first layer consists of leaves, which are the blocks containing the direct hash of the data blocks. Then to get the next layer of the tree, those blocks are concatenated and hashed in pairs. The procedure repeats with the obtained hashes, till a single hash is yielded. Merkle tree can be used for efficient proofs of data inclusion in the following cases:

- One can aggregate the quorum of public keys into one root value with the ability to prove the membership of the particular public key later [2].
- One can check whether a blockchain transaction is included in the block or not [3, 4] (if the appropriate accounting system presumes building transaction trees).
- One can confirm the authenticity and the integrity of outsourced data or data block without the local copy of data files [3, 5–7].
- Users can be identified and verified on the IoT in the blockchain network [8, 9].
- One can check whether the data is properly stored in a cloud drive [6, 10, 11].

*DECaT'2025: Digital Economy Concepts and Technologies, April 4, 2025, Kyiv, Ukraine
* Corresponding author.
† These authors contributed equally.
dubinin@distributedlab.com (V. Dubinin); yp@distributedlab.com (Y. Panasenko); serhii023@gmail.com (S. Volynets); viktoria_zhebka@ukr.net (V. Zhebka); dima.nishchemenko@gmail.com (D. Nishchemenko)
0009-0006-3648-9057 (V. Dubinin); 0009-0004-3588-8617 (Y. Panasenko); 0009-0004-9373-0519 (S. Volynets); 0000-0003-4051-1190 (V. Zhebka); 0009-0006-1781-4109 (D. Nishchemenko)

For the most of the mentioned usages, security is necessary. The tree's digest computation function must be irreversible and collision-resistant to make the forging of a tree node by the owner or third-party adversary improbable, preventing money and/or privacy loss [12].

This paper aims to give a comprehensive overview of existing constructions of a Merkle tree and its modifications for different usages.

2. Preliminaries

2.1. Trees

The structure of a Merkle tree is based on the structure of a regular tree, so it is better to start with general definitions and properties of a tree to feel confident before going to a Merkle tree.

Definition 2.1 (Graph). *A* graph *G* is a tuple *G = (V, E)*, where *V* is a set whose elements are called nodes, and $E = V \times V$ is a set of pairs *(v1, v2)* of nodes, whose elements are called edges. The graph G is unordered, if all the edges have no direction, i.e. the existence of edge *(v1, v2)* implies the existence of edge *(v2, v1)*, otherwise, the graph *G* is called an ordered graph. For the node, *v1* the directed edge *(v1, v2)* is called the outgoing edge, and for a node *v2*, the edge is called the ingoing edge. The graph has an order annotated as |G|, which is the number of nodes in a graph, i.e. $|G| = |V|$. For the node v in an undirected tree, the order of the node v is a number of edges *(v, u)*, where $u \in V$. If the tree is ordered, then the order of a node is a number of all edges *(v, u)* and *(w, v)*, where $u, w \in V$.

Definition 2.2 (Tree). A tree T is a connected and acyclic graph, i.e. there is an undirected path between any pair of vertices and there is no non-empty path from any node to itself without repeated edges. The leaf in a tree is a node with only one edge, i.e. node of order 1.

Definition 2.3 (Perfect k-ary tree). A perfect k-ary tree is a tree of some height h with kh leaves, with non-leaf nodes, which have exactly k child nodes each.

If we have an ordered graph G, we can define the source node v as a node with only outgoing edges, and the sink node as a node with only ingoing nodes. For a directed tree T, a sink node is a leaf node, and the source node is a root node. There exists only one root in a directed tree. The height of the tree T is the maximum length of a path, i.e. the maximum number of edges, from the root to any leaf.

The next thing worth mentioning is that there is a hierarchy, sometimes described in levels. We will say that the root is located on the *0*-th level. Then, we have inner nodes and at the end of each branch, we have leaves. If the tree is perfect, then leaves are located on the last layer. One can use definitions of the parent node and child node to describe hierarchical relationships between two adjacent nodes *u* and *v*. Let the node u be located on a *n*-th level. If the node v is located on level *(n + 1)*, then the node *v* is a child node of *u* and *u* is a parent node of *v*. It is well known that the root has no parents, and leaves have no children.

2.2. Digest functions

A Merkle tree is constructed from data digests. These digests must be small enough to be stored efficiently while also providing security for the tree's construction. Cryptographic hash functions are excellent candidates for this purpose.

Definition 2.4 (Cryptographic Hash Function). For a function to be considered a cryptographic hash function $H: 0,1^{*} \rightarrow 0,1^{n}$, it must satisfy three fundamental security properties that are critical to the integrity of data structures such as Merkle trees.

- Preimage Resistance: This property guarantees that the function is one-sided. For any given hash output *y*, it must be computationally infeasible to find any input message *x′* for which $H(x') = y$.

- Second-Preimage Resistance: For a given input value x, it must be computationally infeasible to find another, different input value x' where $x \neq x'$ that generates the same hash, i.e. $H(x) = H(x')$.
- Collision Resistance: This is the strongest property that makes it computationally infeasible to find any pair of distinct inputs x and x' that result in an identical hash output $H(x) = = H(x')$.

In the context of Merkle trees, collision resistance is of paramount importance [10]. Although resistance to finding a second intro is also an important aspect of security, it is known that collision resistance formally implies second intro resistance for hash functions [13]. The robustness of these properties is critical because they serve as the primary defense mechanism against attackers attempting to falsify nodes and compromise the integrity of the tree.

The hash function can be applied for any string. However, to compute the hash of a set of strings s_i, $i = \overline{1, k}$ we compute the hash of the concatenation of that strings, i.e. $hash = H(s_1 || s_2 || ... || s_k)$ or simply $hash = H(s_1, s_2, ..., s_k)$ or even $hash = H(s_i | i = \overline{1, k})$. It is important not to forget that the order of concatenation affects the final result.

Hash functions aren't the only method for computing digests. An alternative approach involves using vector commitment schemes [10]. Informally, vector commitment schemes [14] can be thought of as a digital sealed envelope. When a party (S) wants to commit to a message m, she places it in the "envelope". Later, S can open the envelope to publicly reveal the message she committed to. In their most basic form, commitment schemes must satisfy two key properties. A commitment must be:

1. Hiding: The commitment should not reveal any information about the message it contains. Specifically, an observer should not be able to distinguish whether a commitment was created for a message m or a different message m', where $m \neq m'$.
2. Binding: The commitment mechanism must prevent the sender (S) from changing her mind about the committed message m after the fact.

More precisely, the binding property requires an efficiently verifiable opening procedure. This allows anyone to quickly check that the opened message is the one S originally committed to. Thus, a commitment scheme typically involves two phases:

1. Commit Phase: A sender (S) creates a commitment (C) for a message (m) using a specific algorithm.
2. Decommit Phase: The sender (S) reveals m and "convinces" a receiver (R) that C is the valid commitment to m.

A single commitment can be created for a vector m that contains several messages. Then, during the decommit phase, S can open just one element of the vector m at a time. A commitment scheme is considered non-interactive if each phase requires only a single message from S to R.

3. Merkle tree

3.1. Definitions and examples

In general, the structure of the Merkle tree (sometimes named the Merkle hash tree) is similar to an ordinary tree. Authors and researchers define a Merkle tree differently. That is why the definition of the Merkle tree varies from article to article. Moreover, some Merkle tree definitions were different enough to have separate names.

Here, we define a simple and general Merkle tree using [15] as a reference.

Definition 3.1 (k-ary Merkle tree). An k-ary Merkle tree $T = \langle V, E \rangle$ is a construction that has leaf nodes, internal nodes and a root, which are as follows:

1. Leaf nodes contain the direct hash value of data blocks.
2. Internal nodes contain concatenation of k hash values of its k children.
3. Root node is an internal node that has no parents.

The k-ary Merkle tree $T = \langle V, E \rangle$ is a tree with the next properties:

1. It is a rooted tree.
2. All the leaves are located on the same level.
3. Each inner node has exactly k children.
4. Duplicated vertices are allowed to meet restrictions.
5. The set of children nodes is ordered set, so digest can be calculated correctly.

Fig. 1 provides an example of a perfect binary Merkle tree. In contrast, Fig. 2 displays a non-perfect Merkle tree, illustrating how duplicated nodes are used to calculate the final root hash.

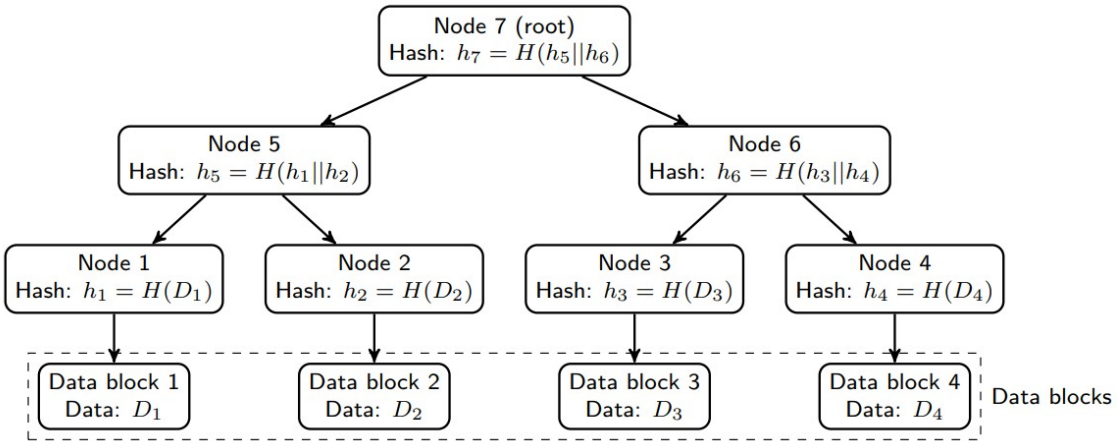

Figure 1: Perfect binary Merkle tree [16]

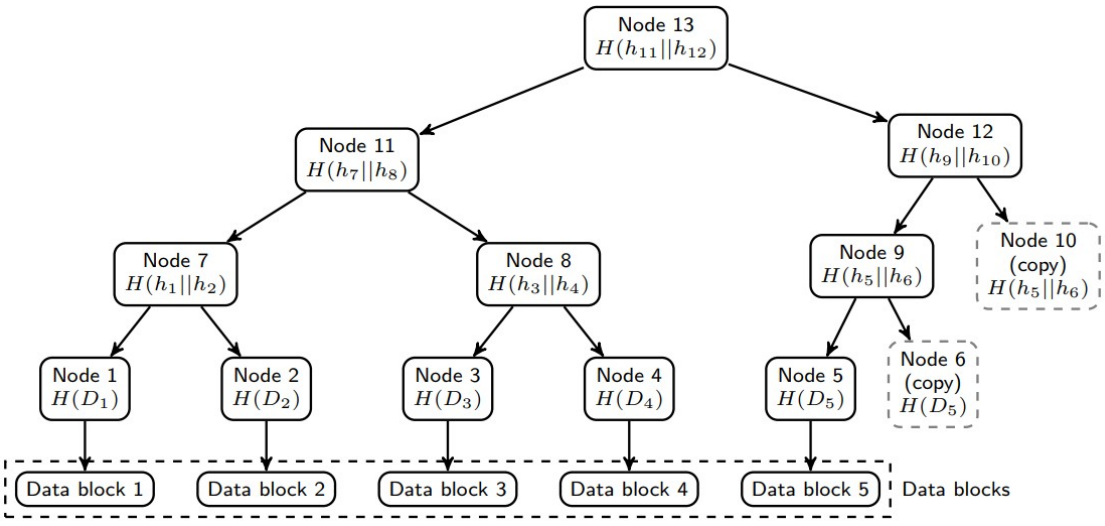

Figure 2: Non-perfect Merkle tree (MT) for 5 transactions [17, 18]. Here D_i is i^{th} data block and h_i is a hash of i^{th} node

In the Figs. 1 and 2, we can see the main idea of the Merkle tree. Let's assume that we compute the digest in a root. Because the digest in the root node depends on all the digest in leaves, any slightest change in any leaf digest changes the digest in a root. So, if we have the digest in a root, then we can check if any change is done. One can use it to prove that the element is in a tree, i.e. there is a leaf node with the element digest in it.

Here is an example of proving data block 2 membership for the tree in Fig. 1. As the hash of the searched data block is stored in node 2, we can give the proof consisting of two hash values: $h_1 = H(D_1)$ from node 1 and $h_6 = H(h_3||h_4)$ from node 6. To check the proof, we first calculate the hash in node 2 as $h_2 = H(D_2)$ using data D_2 that we know. It is also an option to store the hash h_2 and not have the data D_2. Then we calculate $h_5 = H(h_1||h_2)$, the hash value of the concatenation of node 1 and node 2 hashes, which gives us the hash of node 5, and then calculate $h_7 = H(h_5||h_6)$, the hash of the concatenation of node 5 and node 6, which provides us with the hash value in a root. To decide if the data is in the tree, we compare the computed hash in the root node h_7 with the real hash in the root h_{root} (h_{root} is the hash we computed previously or get from the trusted source). If the computed hash in the root matches the real hash in the root, i.e. $h_7 = h_{root}$, the hash of the data is in the tree, otherwise, it is not. However, with negligible probability, it can match without the data inclusion, and the verification procedure will give a false positive result.

If a malicious Prover wants to prove data block 2 inclusion if it is not in the tree, then the Prover must find such hashes of block 1 and block 6, that $H(H(h_1||h_2)||h_6) = h_{root}$. So the Prover needs to find a preimage of h_{root} or $H(h_1||h_2)$ of the hash function H, but we assumed before that the H is preimage-resistant. Another way for a malicious Prover to forge the node is to find a second preimage D_2' for D_2, such that $H(D_2') = H(D_2)$. Then the Prover can claim that the data D_2' is in the tree, however it is not. The malicious creator of a Merkle tree can forge the node from the start by searching any collision D_2' and D_2'' with the same hash, and then claim different statements: the data D_2' is in a tree and the data D_2'' is in a tree. Such manipulation of an adversary is a reason of using the cryptographic hash functions that have appropriate security properties.

The Binary Merkle Tree (BMT) is one of the most commonly used structures among k-ary Merkle trees. There are other named k-ary trees. For example, Ethereum's blockchain utilizes a Patricia-Merkle tree with up to 16 child nodes [19, 20]. The next example is the Jellyfish Merkle Tree proposed in article [21], which is also a tree of order 16.

The node of a Merkle tree contains references to its children and the digest. The main part of a Merkle tree that we need to remember is the hash in the root. However, sometimes it's also important to store hashes of leaves. If the memory isn't strictly limited, and it is possible to store all the nodes, it might be reasonable to append other fields. For example, nodes can have an additional field parent, which is undefined for the root node. For even more convenience, every node can have a boolean flag that states whether the node is the left child. For the root, the flag is undefined. Those fields can aid in constructing algorithms to make them more readable and fast. In our approach, we assume that all the fields are present unless otherwise specified.

Using notation in [2, 22], we can differentiate a node by its level in a tree and its index in a level. Let N_i^j denote the node on level j, with index i. Then the tree will be similar to a triangular matrix, where the next level has twice as many elements as the previous level. This notation is useful because, in an k-ary tree, we can easily locate descendants of a node N_i^j. For example, all children of the node are nodes $N_{k_i}^{j+1}$, $N_{k_i+1}^{j+1}$, ... $N_{k_i+k-1}^{j+1}$. For a binary tree, two children of N_i^j are N_{2i}^{j+1} and N_{2i+1}^{j+1}. can also use the notation with hashed, i.e. hash h_j is a hash on j^{th} level on i^{th} place.

Author of [22] introduces us with "Flat Coordinates". It is a way to identify every node in a tree by a unique number $s \in N$. The author also gives us algorithms for transforming node coordinates from N_i^j to the flat index N_s and backward. This approach makes storing the tree in an array possible. Unfortunately, there is a problem with updating the tree because the appending algorithm

needs to shift the array. However, it can be useful for trees that have fixed sizes. We will N_i^j use notation because it is easier and more human-friendly. The author of [22] uses "Flat Coordinates" to implement his Flat Trees. They are like ordinary Merkle trees but use "Flat Coordinates".

3.2. Merkle tree creation

Algorithm (*alg. 1*) is a pseudocode that creates a k-ary Merkle tree from an array of values. It will store the whole tree in the memory. It uses an additional function *pad_list(nodes, k)* which, for the given set of nodes with cardinality divisible by k, returns the given set of nodes, or else, returns the given set with a duplicated last node at the end to make the cardinality of the set divisible by k. If $k = 2$, function *pad_list(nodes, 2)*, ensures that the set nodes cardinality is even. It isn't necessary to duplicate all the node's descendants in the pad list function. We need a divisible by k amount of hashes, so we can duplicate only a hash instead of a node.

The most resource-consuming operation in the tree creation is hashing. For a perfect binary tree with n leaves to compute the first layer of nodes, we will compute n hashes, for the next n , and so on.

Therefore, the time complexity of Merkle tree creation is

$$T(n) \approx n + \frac{n}{2} + \frac{n}{2^2} + \ldots + 2^{\log(n)} < 2n \tag{1}$$

For the perfect k-ary Merkle tree, the claim is similar.

$$T(n) \approx n + \frac{n}{k} + \frac{n}{k^2} + \ldots + k^{\log k(n)} < 2n \tag{2}$$

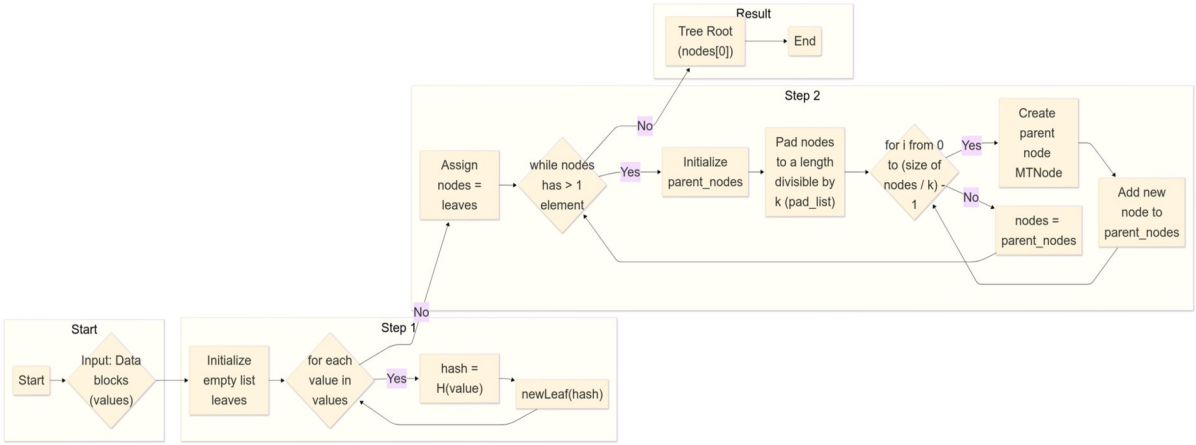

Figure 3: Algorithm 1: Create k-ary Merkle tree

Algorithm (*alg. 1*) is inefficient in terms of memory complexity. To be precise, it stores n leaf nodes, which contain one hash each, and O(n) inner nodes, which contain one hash and $k + 1$ pointers.

It is useful to know that the height of an *k*-ary Merkle tree is *logk(n)*. For a binary Merkle tree, the height is *log2(n)*. It is quite obvious that trees with a bigger arity are shorter and have fewer nodes. Unfortunately, it doesn't affect the algorithms a lot. The main effects are implementational.

Appending of a new element, i.e. appending of a leaf, can be accomplished by a complete rebuild of the tree. Then, the complexity of appending a new element to the tree equals a new tree creation. In other words, complexity is *O(n)* hashing. However, there is a solution to the problem. We can mark duplicated nodes. If we have a marked leaf, swap it with the new element. If we have a duplicated inner node, we must swap it with a new node with the new leaf in its descendants; that is more complicated. If the tree is full and no duplicate nodes are left, then we increase the tree's height by changing the root. The root becomes an internal node. In theory, the appending of a new element leads to the recalculation of the root. The appended element affects only one node on each level, so it should require *O(logk(n))* hashing operations.

3.3. Membership proofs

As was said before, it is possible to create a proof of element membership for the Merkle tree. There are different ways to prove data inclusion. The first is through some equality verification (let's name it "verifiable proof"), and the second is through the direct search of the element in the memory (let's name it "searching proof").

The first type of proof that was already mentioned ("verifiable proof") is a membership proof, which is a set of pairs of hash (hashes) and direction ($\{(h_i, d_i) | i = \overline{1, k}\}$). It might be helpful to append node parameters or other information to the proof to see what we are proving in more detail. For example, it is possible to append a version of a tree to the proof [21] or provide the node with a number of currently appended leaves [21], instead of the version. We might need the version of a tree to know what hash version in a root we need to compare our computed hash in root with.

The "verifiable proof" is a membership proof for one data block. However, if we combine several proofs for some nodes, we will get proof of membership for a set of nodes named ranged existence. In [22], we can see a lot of different types of proof derived from the "verifiable proof". The main types worth mentioning are the already mentioned ranged existence, delete proof, proper removal proof, update and ranged update proofs, which prove that a single node or a set of nodes was updated within a tree, etc. The main idea of the derived types is to prove the existence of the node and its neighbor nodes.

Authors of [21] use "searching proofs". Instead of giving you the set of hashes, it provides you with path from the root to the node. Their path is a concatenation of a version of a tree and a concatenation of direction, named the radix path [20]. In a binary tree, we can associate 0 with the left child and 1 with the right child, so the radix path is a sequence of bits that help you locate the node. In [21], trees are 16-ary trees, so every inner node has 16 children. One child is marked as a hexadecimal number, i.e., a symbol in {0, 1, ..., 9, A, B, ..., F }. To verify the proof, you need to store the whole tree. Fortunately, it makes the proofs smaller and verification faster because all the hashes are computed, and you only need to find the node in a tree using comparisons instead of hashing. It is possible to use "searching proofs" along with the "verifiable proofs" if needed.

Algorithm (*alg. 2*) is a pseudocode for "verifiable membership proof" generation for a binary Merkle tree. Firstly, it searches for the required element by searching over the set of leaves, which, unfortunately, is unsorted. The search requires *O(n)* comparison operations. When the node is found, the algorithm (*alg. 2*) searches for the corresponding hash values in *O(log2(n))* operations to generate membership proof. Because all hash values are already calculated, the last complexity is also evaluated in comparison operations.

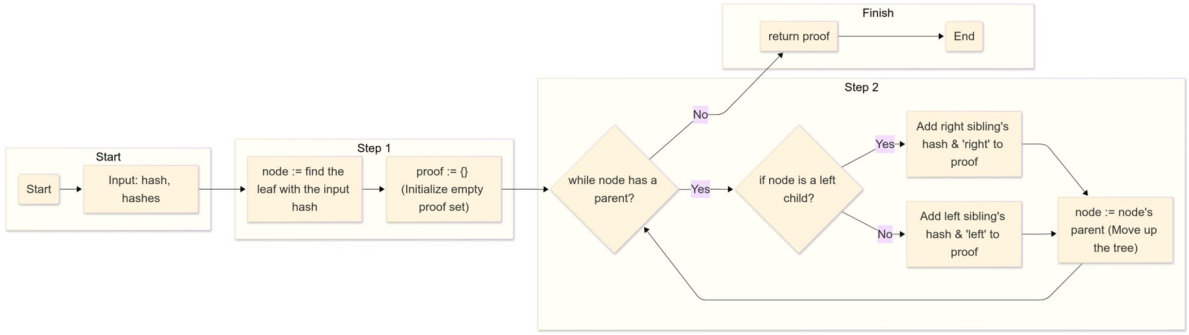

Figure 4: Algorithm 2: Generate binary Merkle tree proof

In a binary Merkle tree, a membership proof (per Algorithm 2) consists of a set of pairs. Each pair contains a hash value and a positional indicator specifying whether the hash should be concatenated from the left or right. Consequently, the size of such a proof is on the order of *O(log2(n))*. For a k-ary Merkle tree, proof construction methodologies differ, primarily in the structure of the proof elements. Consider a verifier who, at a given step, holds a hash h_i and needs to compute its parent hash, defined as $h = H(h_1, h_2, ..., h_i, ..., h_k)$. The proof must supply all sibling hashes of h_i.

One efficient approach is to structure the proof element as a pair of concatenated strings:

A left-side string: $l = h_1||h_2||...||h_i-1$

A right-side string: $r = h_i+1||h_i+2||...||h_k$

In this model, if h_i is the first element (i=1), the left string l is empty. Likewise, if h_i is the last element (i=k), the right string r is empty. To proceed, the verifier computes the parent hash using the received elements as $H(l||h_i||r)$.

The other way is to represent the step of the proof like a whole set of additional hashes and an id of position, where the hash from the previous step must be concatenated. The first strategy requires less space because there is no id in it, and has less complexity to verify because we don't have to find appropriate places to put the hashes. However, both strategies have the same time complexity. The size of the proof for k-ary Merkle tree is $O((m - 1)logm(n))$. Algorithm (*alg. 3*) shows the second approach for an k-ary Merkle tree membership proof cration.

Algorithms (*alg. 4*), (*alg. 5*) that checks membership proof uses hashing operation. For the k-ary tree, algorithms will perform $(k - 1) logk(n)$ hashing operations. So the complexity of verification of membership proof is $O((k - 1) logk(n))$, where n is the number of elements in the tree.

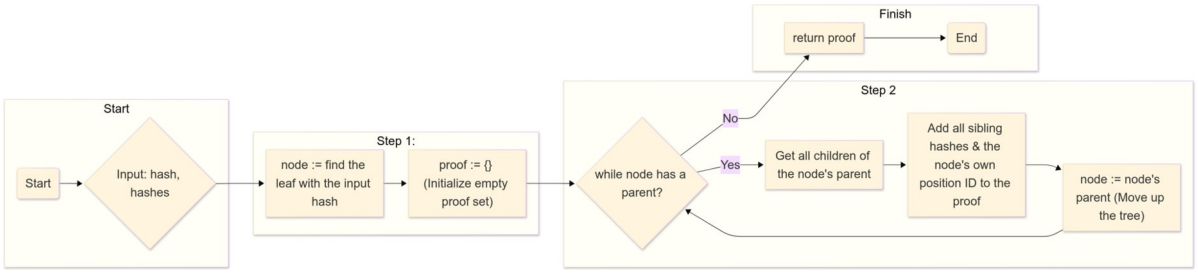

Figure 5: Algorithm 3: Generate k-ary Merkle tree proof

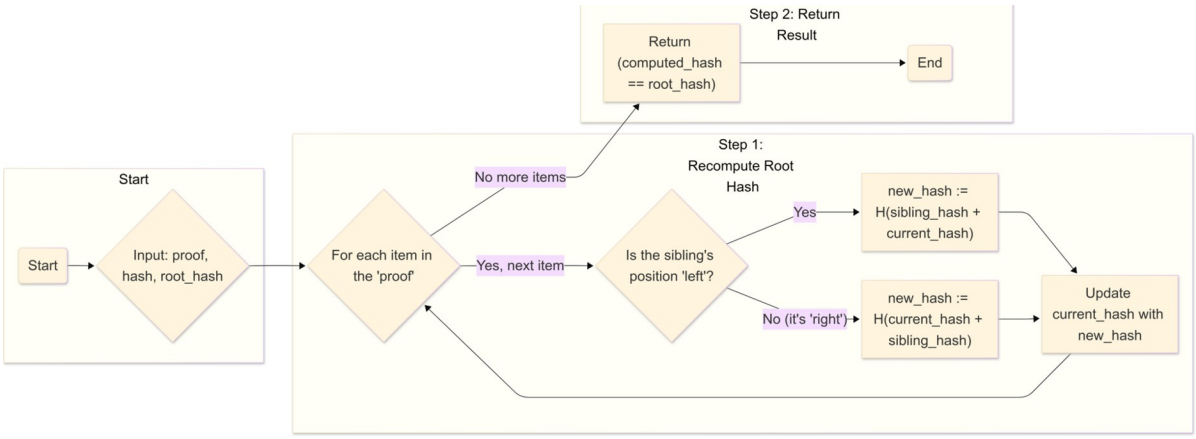

Figure 6: Algorithm 4: Check binary Merkle tree membership proof

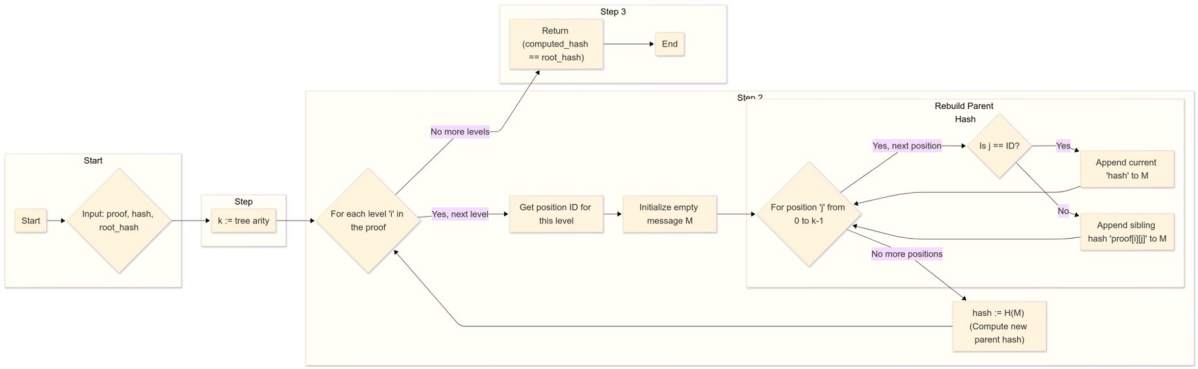

Figure 7: Algorithm 5: Check general membership proof

4. Merkle tree extensions

As previously established, numerous definitions and variations of Merkle trees exist. Merkle trees and their extensions can be categorized based on several key characteristics:

1. The tree structure is not necessarily binary. It can be designed as a generic k-ary tree, where each internal node has at most k child nodes. Implementations such as binary (k=2) or 16-ary (k=16) trees are common.
2. The tree's topology may be perfect or imperfect, as well as balanced or unbalanced, depending on the specific application and construction algorithm.
3. The overall structure can be either static (constant) or adaptive, dynamically changing in response to data modifications.
4. The digest function itself can be customized. This includes variations in its input parameters or the computational logic used to produce the hash output.
5. The tree may incorporate specialized node types beyond the standard leaf and internal nodes to support extended features.
6. The fundamental tree can be augmented with additional data substructures to provide enhanced functionality.

4.1. Sparse Merkle Tree (SMT)

A Sparse Merkle Tree (SMT) [2] is an authenticated data structure conceptually modeled as a perfect Merkle tree of intractable size. The structure assumes a distinct leaf node for every possible output of its underlying cryptographic hash function. Consequently, for a hash function with a 256-bit output space, such as SHA-256, the tree would conceptually comprise 2^{256} leaf nodes. To use that behavior, we can assume that every leaf has an additional field id. With that field, we can

efficiently append new elements. Moreover, because of that field, we don't have to search for the element in the membership proof generation. So we will win in speed, but occasionally, we lose in memory. To bind the hash of the leaf to its id, we define the hash of a leaf node as *H(id||data)*.

Remark. Sometimes (like in [21]), a sparse Merkle tree is called an Addressable Merkle Tree (AMT). Both names are correct and give some intuitive information about the tree. We will use the first name "Sparse Merkle Tree".

To create an empty *k*-ary sparse Merkle tree of height h, one can use pseudocode (*alg. 6*), where the input size must be equal to kh, where k, h ∈ N, k ⊠ 2. If we are using SHA-256, We can have sizes 2^{256} (k = 2, h = 256), 4^{128} (k = 4, h = 128), etc.

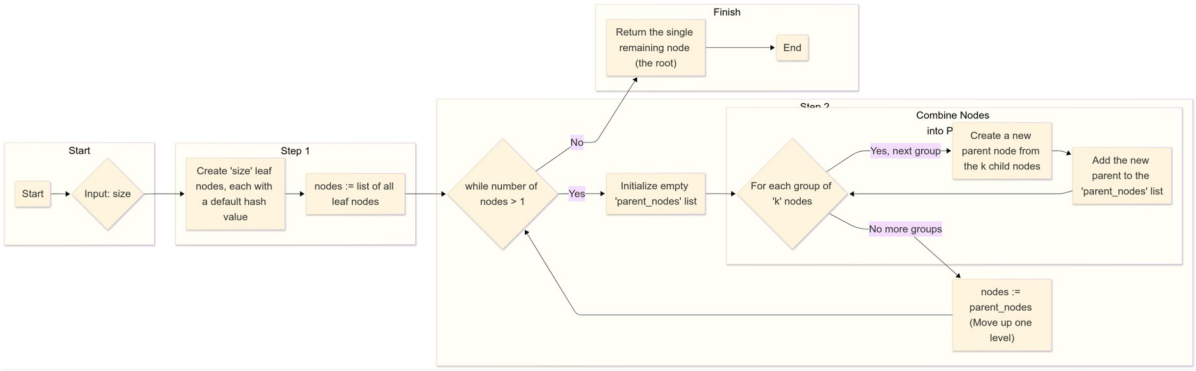

Figure 8: Algorithm 6: Generate empty k-ary sparse Merkle tree.

Appending a new leaf here means changing the existing leaf hash because the structure has already been created, and updating the tree. The algorithm for appending a new element is much simpler than the one for a binary Merkle tree because we know where the new node must be, and there is no need to rebuild the entire tree or find an empty place. To find a leaf algorithm, use O(1) operations. To update the tree, we need to update the leaf, the changed leaf's parent and its parent, and so on. If we use pointer parent in all the nodes, then it is a trivial task. If we save all the nodes using the mentioned notation, then, if we change leaf, we N_i^j know that its parent node is node $N_{[\frac{i}{k}]}^{j-1}$ (or in other words children of a node N_i^j is node $N_{k_i+a}^{j+1}$, $a = \overline{0, k-1}$).

The algorithm of membership proof generation for sparse Merkle tree is similar to (alg. 3) for *k*-ary Merkle tree proof generation. The only difference is the search of a node. The size of the membership proof is also the same as for a *k*-ary tree, i.e. *O((k – 1) logk(n))*. Verification of the membership proof has no change compared to (*alg. 4*). However, if the hash of a leaf is computed as *H(id||data)*, then it is reasonable to attach the id of a node to the proof and use the right hash value for the proof verification at the beginning.

For a sparse Merkle tree, there is also a method of proving element exclusion. The check of the exclusion is the same algorithm as the check of inclusion because if the element exists, we know the id of the leaf where the hash of the data must be located. Therefore, if the algorithm (*alg. 4*), for the SMT proof, returns True, then the element exists, and if the algorithm returns False, the element doesn't exist.

4.2. Indexed Merkle Tree (IMT)

An Indexed Merkle tree is a Merkle tree, extended with additional structure [23]. To be precise, leaves are organized in a linked list. Leaves have four fields: *val, nextId, nextVal,* and *hash*. The hash field equals the hash value of the concatenation of three previous fields. Because storing data directly in the leaves might be insecure and memory inefficient, it is possible to use *val* and *nextVal* fields to store data hashes. Internal nodes are the same as the usual Merkle tree internal nodes. Like the sparse Merkle tree, the indexed Merkle tee can be an intractable structure, but it is possible to

make it more compact because the newly appended leaf will be located in the most left-most free position. That is why we can store the biggest left-most nonempty subtree of the tree.

Algorithm (*alg. 7*) is presented in the form of a pseudocode for creating a new empty k-ary indexed Merkle tree with a predefined height. It is pretty much the same as the other tree-creation algorithms ((*alg. 1*), (*alg. 6*)). The only difference is the arguments for leaf creation are different (like with sparse Merkle trees).

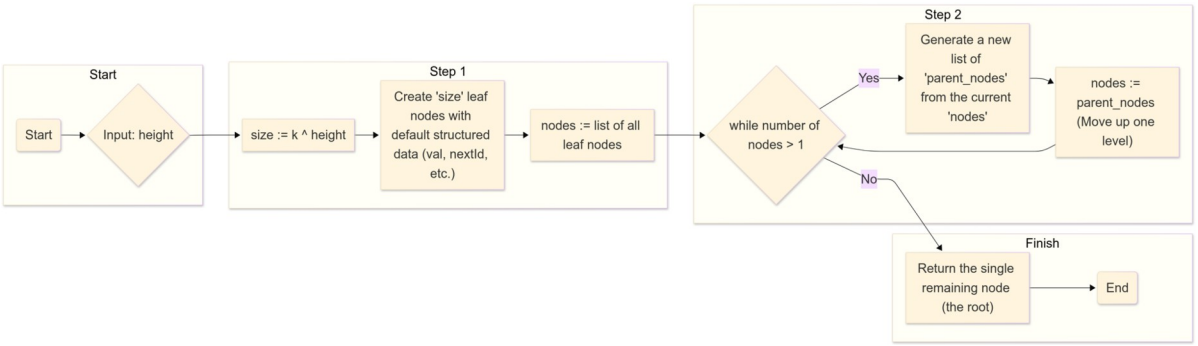

Figure 9: Algorithm 7: Generate empty k-ary indexed Merkle tree

Appending a new value requires finding the leaf in *O(n)* comparison operations in a sorted linked list, and then two updates of the root in *O(log2(n))* operations of finding hash values for each update.

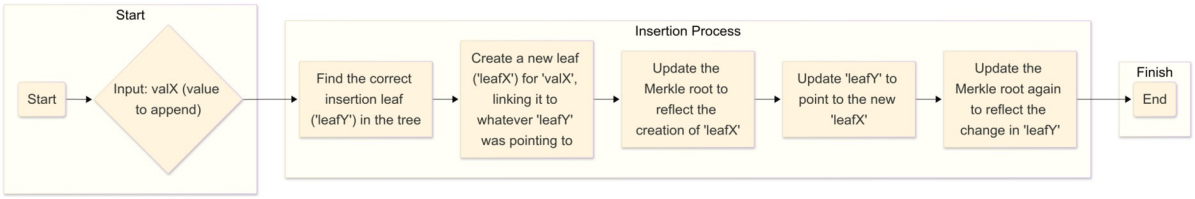

Figure 10: Algorithm 8: Append element to indexed Merkle tree

There is an algorithm for appending multiple leaves at once for the indexed Merkle tree. It is the most efficient when we append the sorted set of values [23, 24]. For more, we can efficiently append a Merkle subtree or indexed Merkle subtree to the existing indexed Merkle tree [24].

Algorithm (*alg. 9*) for the membership proof generation is similar to previous algorithms but has some differences. The main difference is the additional return values. We need an algorithm to return the set of searched node values because the hash value of a node isn't only the hash of a value but a hash of the concatenation of node values: *val, nextId, nextVal.* We can't know the last two values beforehand, unlike the id in a space Merkle tree.

Algorithm for generating membership proof (*alg. 9*) [24], firstly look for a leaf that has the same value as the one searched for or as close as possible to its value. It requires *O(n)* comparison operations. Then, the algorithm searches for the hash values of the required vertices in *O(log2(n))* comparisons. Because the structure formed by leaves is sorted (the further, the greater the values are), we can find exactly where our desired element should be. Therefore, if the verification of the membership proof returns True, then our element belongs to the tree, and if the algorithm returns False, then the element does not belong to the tree. So, the membership proof is also an exclusion proof. The proof verification algorithm is the same as for other trees but uses the proper concatenation for the leaf node values.

4.3. Verkle Trees (VT)

Verkle Tree [10], i.e., Very Short Merkle Tree, is a bandwidth-efficient alternative to Merkle Tree. The main difference is the replacement of cryptographic hash functions with Vector Commitments. That change affects the size of the proof for k-ary Verkle Tree. It works better if the k is bigger. In k-ary

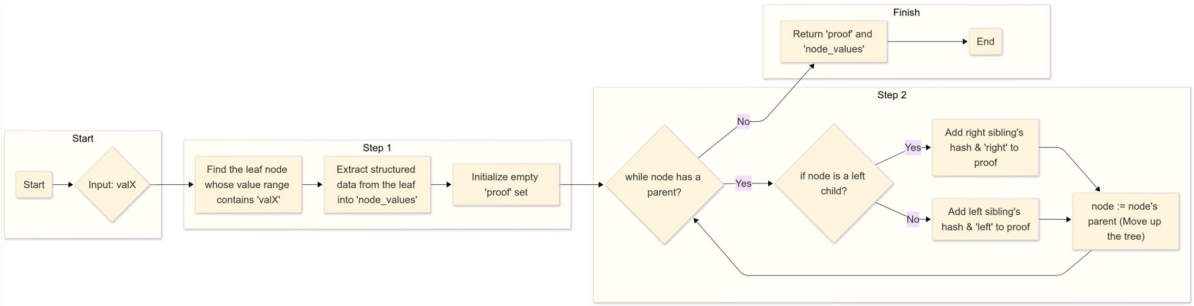

Figure 11: Algorithm 9: Generate membership proof for indexed Merkle tree

Merkle trees, root's hash value is equal to $H(h_1, h_2, ..., h_k)$, where h_i, $i = \overline{1,k}$ are hashes of root children. If we need Prover to prove that hash hj is really in the tree, Prover needs to send the Verifier all h_i, $i = \overline{1,k}$, $i \neq j$. However, if we use vector commitments, then we can calculate the hash of the root using the next formula: $H'(h_1, h_2, h_3, ..., h_k) = H'(h_1, ..., h_{j-1})H'(h_j)H'(h_{j+1}..., h_k)$. So, Prover needs to give only two values instead of $k-1$. That change makes the proof size of the Verkle tree more compact than an ordinary Merkle tree proof ($O(logk(n))$ instead of $O(k\ logk(n))$). The only downgrade is the usage of vector commitments that are more time-consuming than hashing, which makes the complexity of tree creation and tree element insertion harder than for the Merkle tree ($O(k_n)$ instead of $O(n)$ for construction [10]). However, it is worth mentioning that bandwidth is typically much more expensive than computational power in the applications.

4.4. Radix Merkle Tree (RMT)

Radix Merkle is built on top of the Merkle tree but with a different goal. For more, it uses "search proof". However, as said before, is possible to use both types of proof.

The difference between the Merkle tree and the radix Merkle tree is described in Fig. 3. The Radix Merkle Tree [20] consists of leaves, branches (or just an inner node), and root nodes. In addition to them, radix trees, for example, in Ethereum [20], also use the Extension node, which is a node containing a section of a radix path spanning two or more nodes without any side branches that are common for two or more child nodes. Leaves in a radix Merkle tree aren't always located on the last level. A leaf can even be a child of the root. These different node types and the way the tree stores the information creates a very different tree structure.

It is common to use the 16-ary radix Merkle tree. Because every inner node has $16 = 2^4$ child nodes, we can name every child node with a hexadecimal digit. Because every symbol takes exactly 4 bits, it is common to use nibbles.

Definition 4.1 (Nibble). A nibble refers to four consecutive binary digits or half of an 8-bit byte.

The term nibble is used to describe a child of a node, and so to define a path from the root to some leaf. Radix tree has a different proof format that uses nibbles. The membership proof is a radix path from the root to the leaf. In other words, the radix path is just an ordered set of directions to the next nodes, i.e., nibbles. We can concatenate those hexadecimal digits into one hexadecimal number representing the path.

The main advantage of a radix tree [20] is the ease of locating data from an address's radix path, sometimes called node key. The next most valuable property is the ease of appending a new element, such that a tree is still sorted. A radix tree's disadvantage is that it is sparsely populated

with leaves at varying heights until the radix tree is nearing full saturation. Therefore, if the lookup advantages of a radix tree could be transferred to a *k*-Merkle tree, say by using radix paths in all TXs, then a very dense data structure with leaves on a common level and fast look-up times could be created.

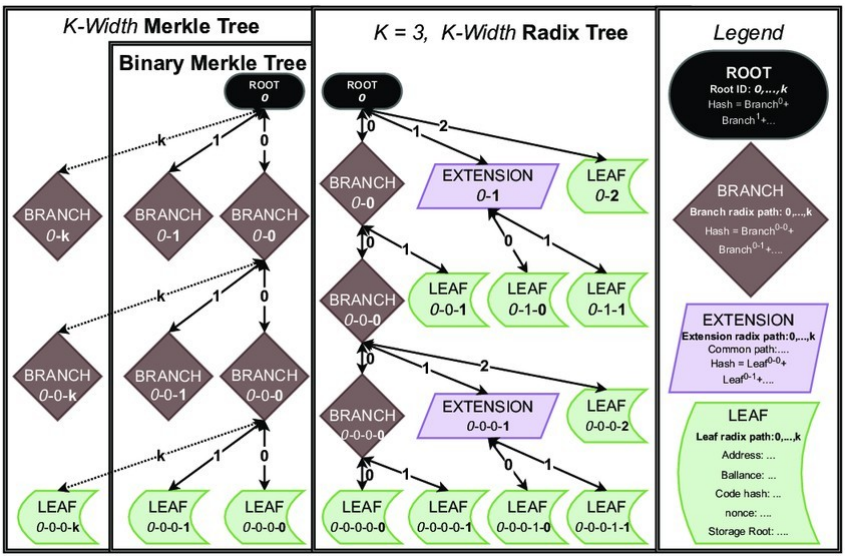

Figure 12: Comparison of binary Merkle, k-Merkle, and k-radix tree structures [20]

Because the radix path is a hexadecimal number, it is possible to use the account address as a radix path [20]. That gives us a tool to record the users.

Authors of [20] give an example of radix tree creation for Ethereum account management. A 32-byte long Ethereum address is read in hex, where each hex value is read as a path in the $k = 16$-radix tree. A hex value between 0 and 15 corresponds to a path from the current node to one of its children ranging from 0 to 15. Once the last occupied node is passed, regardless of whether all hexes in the address have been read, a leaf node containing the account is created. If a leaf node is reached during this process, that leaf node is replaced with a branch node, and the account leaves are moved down a level. When a path contains a series of nodes that do not branch off (i.e., each node has only one child), that sequence is compressed into a single extension node. This optimization avoids using multiple branch nodes to represent a simple, linear path.

4.4.1. Relative Index and Time Stamped Merkle Hash Tree (RITS-MHT)

The Relative Index and Time Stamped Merkle Hash Tree (RITS-MHT) is a data structure proposed in [11]. It was specifically designed for data auditing in cloud computing environments [25]. This data structure is a tree that integrates a Merkle Tree (MT) with a radix path for each node. This design significantly improves search efficiency, reducing the computational cost of finding a data block from *O(n)* —as found in Wang's protocol [6] — to a much faster *O(logn)*. Additionally, the structure tracks the time of the last data modification, which serves to guarantee data freshness. In [11] described algorithms for inserting and removing data and its signing.

4.4.2. Merkle Patricia Trie (MPT)

Merkle Patricia tree (or Merkle Patricia trie) [19, 20] is a radix Merkle tree implementation used in Ethereum. In Ethereum, every block header contains three Merkle trees for three kinds of objects [19]: transactions, receipts, and state, which aims at allowing light clients to make and get verifiable answers to many kinds of queries. The transactions tree serves to verify the inclusion of a transaction within a block, while the receipts tree facilitates the retrieval of all instances corresponding to a particular event (e.g., amount of gas used or any event logs from the

transactions), and the state tree to check the current balance of an account, whether an account exists, or to simulate running transactions on a given contract.

Ethereum requires a tree data structure that can quickly recalculate a tree root after an edit, insert, or delete operation. The tree root must depend on the data and not on the order in which updates are made. Note that regular Merkle trees do not satisfy this requirement. Furthermore, the data structure must prevent Denial-of-Service attacks where malicious attackers may craft transactions to make the tree as deep as possible and, hence, slow updates. To stop this, the data structure must have bounded depth.

In The Merkle Patricia Tries, proof of membership, i.e., key, is encoded using a special Hex-Prefix (HP) encoding [19]. The nibble is appended to the beginning of the key to signify parity and terminator status. Parity denotes if the length of a key is even or odd. Terminator status denotes whether the node is an extension or a leaf node. Note that if the original key value was of even length, an extra zero nibble would be appended to achieve overall evenness. This ensures that the key can be properly represented in bytes.

4.4.3. Jellyfish Merkle Tree (JMT)

Jellyfish Merkle Tree [21] is a modified version of $AR_{16}MT$, i.e. Addressable (Sparse) Radix 16-ary Merkle Tree, with the following features:

- **Version-based Node Key.** JMT chooses a version-based key schema with multi-fold benefits:
  - Facilitating version-based sharding.
  - Greatly lowering compaction overhead in LSM-tree (Log-Structured Merge-tree) based storage engines such as RocksDB.
  - Smaller key size on average.
- **Less Complexity.** JMT has only two physical node types, Internal Node and Leaf Node. Ex- tension node is removed, because it is unlikely for two paths to share a long common part of the path.
- **Concise Proof Format.** The number of sibling digests in a JMT proof is less on average ($\Theta(log(\ number\ of\ existent\ leaves))$) than that of the same A(S)RMT without optimizations ($\Theta(log(\ number\ of\ maximum\ leaves))$), i.e., the height of the equivalent A(S)MT), requiring less computation and space.

Authors of [21] created Jellyfish Merkle Tree as a tree optimized for computation and space, designed for the Diem Blockchain. The proof format and verification algorithm are complex, but the tree has a smaller proof size and less computation overhead of verification that practically benefits users while keeping the algorithm complexity transparent to end users.

5. Improvements

5.1. Storage space improvement

5.1.1. Pruned Merkle Tree (PMT)

In some cases, using Merkle trees to compute digest, i.e. the hash in root, there is a need to duplicate the node. Authors of [17, 18] say that it is an ineffective way and propose their approach, firstly named Merkle Trim Tree (MTT), and then named Pruned Merkle Tree (PMT). They change the construction algorithm by delaying the computation of the digest of the first node and other nodes to the root. Such a construction allows us to save a little of memory by not copying blocks. An example of the difference between the Merkle tree and the Pruned Merkle tree is shown in Fig. 13. Because of not copying the node, the algorithm for PMT creation will save space equal to

two nodes in comparison with the usual MT creation algorithm. If the tree is perfect, then the pruned Merkle tree will look like the usual Merkle tree.

5.1.2. Default hashes for SMT

A sparse Merkle tree is a big mathematical structure, so despite being fast, sparse Merkle trees need a lot of memory because there are a lot of empty nodes, so, there is a necessity to use some memory-saving strategies.

The problem with memory can be solved by associating the empty subtree, i.e. subtree without non- default leaves, with one node with some default hash [2, 21]. We can do it in two ways: by predefined hash value hash for the whole tree or by one predefined hash for one level of the tree, i.e. for a tree of height h, we have $hash_h = hash$, $hash_{h-1} = H(hash_h, hash_h)$, $hash_{h-2} = H(hash_{h-1}, hash_{h-1})$, etc. However, that idea is incomplete because it slows down non-membership proof. The next way is not only to change the empty subtree by one node but also to leverage the leaf to a higher level. Unfortunately, that approach leads to higher algorithm complexity.

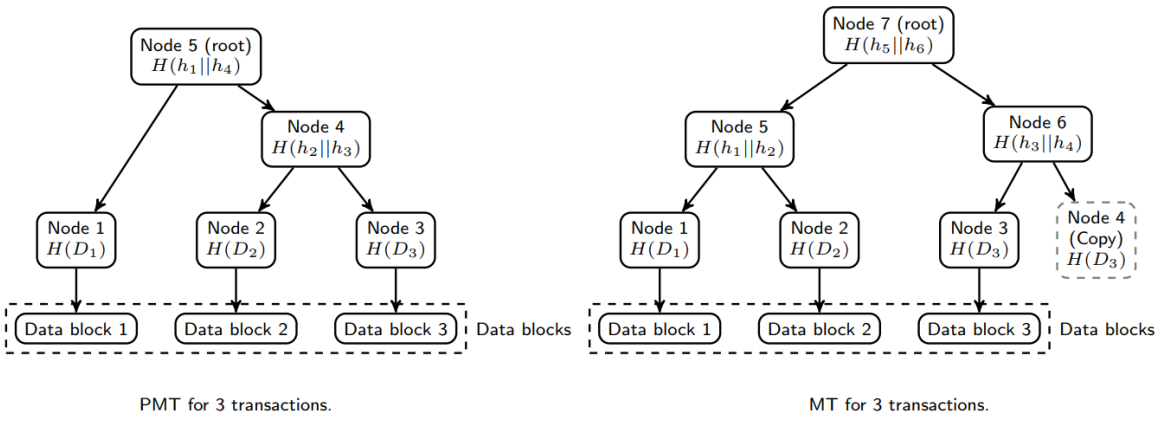

Figure 13: The comparison of transactions in the traditional Merkle tree (MT) and Pruned Merkle Tree (MTT) to form the root for three transactions [17, 18]

5.1.3. Tree inner nodes storing

Another way to save memory for a sparse Merkle tree is not to save inner nodes hash in the memory, because it is possible to calculate them when needed using leaf hashes. There is an algorithm, TREEHASH [26, 27], which can compute the root of a tree and require only leaves hashes. The algorithm can compute the hash in a root of a tree in $O(n)$ time, using $O(log(n))$ space in the process. TREEHASH is created for binary trees, but extending it to work with k-ary tree isn't a problem, (alg. 10).

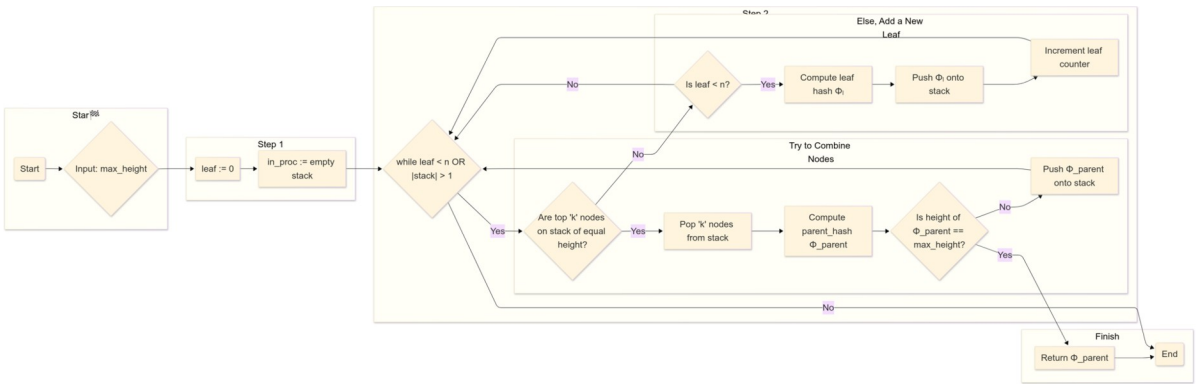

Figure 14: Algorithm 10: TREEHASH for k-ary tree

Algorithm (*alg. 11*) can generate proofs and require only leaves hashes to work. It is similar to algorithm (*alg. 1*), which creates the tree, but with additional steps and without the tree creation. Its membership proof generation requires you to calculate *O(n)* hashes for the whole tree to create the membership proof, so it is memory-efficient but time-inefficient. The mentioned algorithm is a straightforward method for calculating the proof. There are better approaches to it [26–28].

5.1.4. Time and space compromise solution

To find a compromise between time and space, authors of [2] define caching strategies based on capturing branches. The main idea is to store hashes of some nodes, and then use saved hashes to skip some computations. On average, there will be faster proofs. Author of [28] proposes a proof computing algorithm working with O(log(n)/h) operations per output and O(log(n(2h/h))) space, using some saved subgraphs.

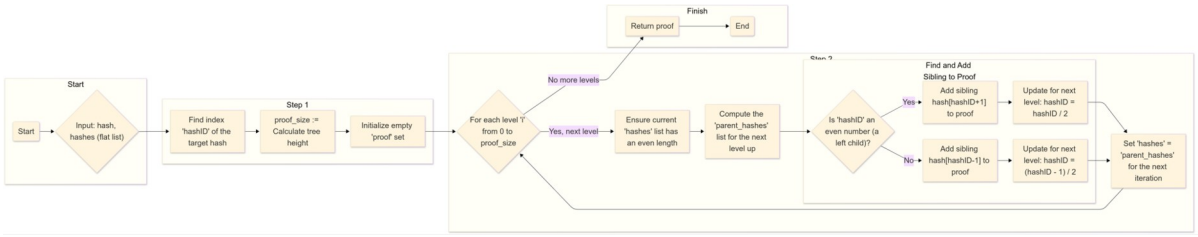

Figure 15: Algorithm 11: Generate binary Merkle tree proof from hashes

5.2. Membership proof size improvement

Average membership proof size improvement. In [29] described a strategy similar to building a Hafman code tree. It reduces the size of membership proof for frequently used elements but increases the membership proof size for rarely used elements. This approach makes the proof size less on average than the same for the usual Merkle tree.

Verkle Trees (VT). As was said before, Verkle Trees has a small membership proof because they use committing instead of hashing. That, reduce the proof size from O(k logk(n) to O(logk(n), by the price of computations. There is no problem with implementing that strategy in sparse Merkle trees, indexed Merkle trees, or even in radix Merkle trees.

5.3. Logic improvement

The Odd and Even Merkle Hash Tree (O&E MHT), proposed in [8], presents an alternative method for root hash computation. The fundamental principle of this approach is to deviate from sequential node processing. Instead, nodes are segregated based on their index parity: all odd-indexed nodes are aggregated to form a left branch, while all even-indexed nodes constitute a right branch. An example of this structure is illustrated in Fig. 14.

Notably, this construction does not reduce algorithmic execution time; on the contrary, it increases the overall complexity of the implementation. The Modified Merkle Hash Tree (MMHT), also presented in [8], is a technique designed to avoid the need for node duplication. The methodology involves partitioning the initial set of hashes into two distinct subsets. Following this, two intermediate hash values are computed from these respective subsets, potentially using different schemes. The final root hash is then derived by computing the hash of the concatenation of these two intermediate results.

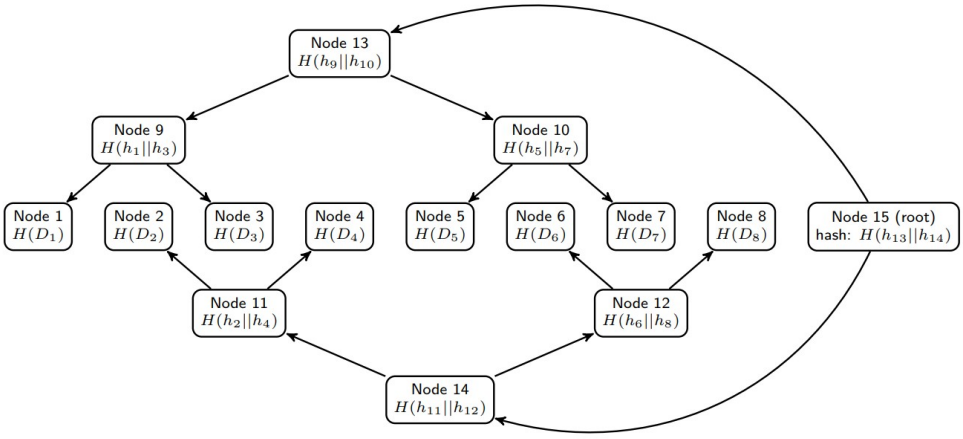

Figure 16: Example of perfect Odd and even modified Merkle hash tree (O&E MHT) [8]. Here h_i is a hash value of i-th node

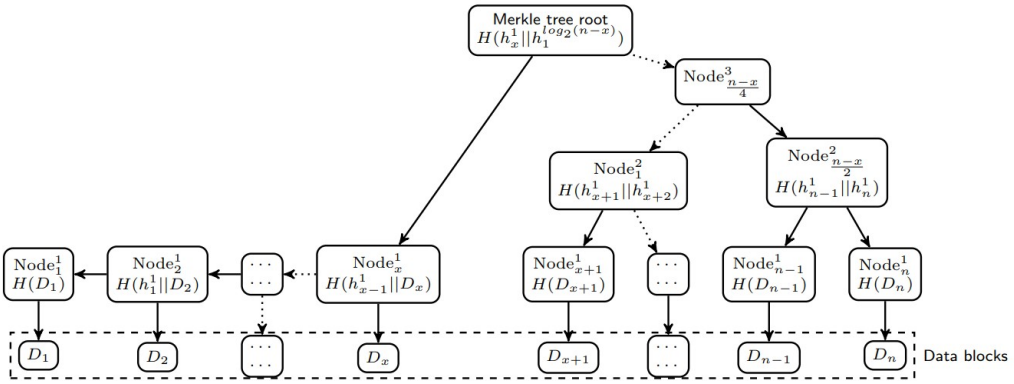

Figure 17: Binary modified Merkle hash tree (2-ary MMHT) [8]. Here h^j_i is a hash in i-th node in j-th row. That is also true for a Node notation. $Node^j_i$ is a node on i-th place in j-th row

As shown in Fig. 17, for a binary Modified Merkle hash tree (2-ary MMHT) there is a set of x values that digest is computed as a chain, and the set that digest is computed as the usual binary Merkle tree scheme. We can use the tree construction to avoid duplication of nodes in the Merkle tree. MMHT algorithm process steps are described in algorithm 1 in [8]. Authors are using a Count trigger value to give a number of nodes to be concatenated together. As shown in Fig. 17 to avoid duplications we need to have $x+2^k$ data blocks, so the second part will form the perfect Merkle tree of height k. However, it is possible to give fewer values than needed to have the perfect Merkle tree. So the complexity of the approach for a tree that is divided into two parts of length x and t is equal to the complexity of an ordinary Merkle tree with t elements + x.

There are some problems with MMHT. The main of them is the time to get to the first blocks. The less the sequence number of the block, the bigger the time is. Because of that, for the first blocks, the proof size is bigger, the modification time is bigger, etc.

In article [30] about recursive STARKs, the author introduces us to recursive proving. using recursive STARKs we prove some statements for leaves, then for nodes in the next level and up to the root node. The proof is valid if all the proofs are valid. It is an AND logic. Because AND logic is harder than OR, authors of [31] propose to use OR logic. They write a lot of advantages of using OR logic in proof aggregation. Unfortunately, that work is in progress and there is no practical implementation yet.

6. Comparison and results

All the mentioned trees have different constructions, use cases, and complexities. Table 1 describes the main advantages and disadvantages of those constructions.

Table 1
Merkle tree types comparison

Tree type	Advantages	Disadvantages
k-ary MT	Simple implementation.	Average speed.
k-ary SMT	Faster insertion and deletion compared to k-ary MT, support nonmembership proof.	Requires a lot of memory (leaf for every possible output of the hash function).
k-ary IMT	Faster insertion and deletion compared to k-ary MT. Can be built on top of an existing linked list. Has a smaller downgrade for appending multiple elements at once than SMT.	Slow search of node, which can affect the speed of membership proof creation, element insertion, etc., when the tree is nearly full.
k-ary VT	Short proofs, so faster proof exchange and proof verification.	Hard creation and insertion because of the use of vector commitments instead of hashes.
k-ary RMT	Short proofs, and even shorter proofs for proofs with multiple nodes with a common path.	Complex implementation, memory inefficient.

Theoretical results of the complexity of the main algorithms, i.e. new element insertion, membership proof generation and verification are shown in a table 2.

In Table 2, “h.” means complexity in hashing, and “c.” means complexity in comparisons. There are two exceptions. VT uses a commitment function instead of a hashing and RMT uses a comparison function instead of hashing. MP search means MP creation while the whole tree structure is available

Table 2
Theoretical complexity for trees with *n* leaves

	k-MT	*k*-SMT	*k*-IMT	*k*-VT	*k*-RMT
tree size (nodes)	O(n)	O(n)	O(n)	O(n)	O(n)
proof size (pairs)	O(k log_k(n))	O(k log_k(n))	O(k log_k(n))	O(log_k(n))	O(k log_k(n)) nibbles
Creation (h)	O(n)	O(n)	O(n)	O(kn)	O(n)
Inserting (h)	O(k log_k(n))	O(k log_k(n))	O(k log_k(n))	O(k log_k(n))	O(k log_k(n))
Node search (c)	O(n)	O(1)	O(n)	O(n)	O(1)
MP search (c)	O(n)	O(log_k(n))	O(n)	O(n)	O(1)
MP creation (h)	O(n)	O(n)	O(n)	O(n)	O(1) c.
MP verification (h)	O(k log_k(n))	O(k log_k(n))	O(k log_k(n))	O(log_k(n))	O(log_k(n)) c
NMP search (c)	-	O(log_k(n))	O(n)	O(n)	O(1)
NMP creation (h)	-	O(n)	O(n)	O(n)	O(1) c.
MMP verification (h)	-	O(k log_k(n))	O(k log_k(n))	O(log_k(n))	O(log_k(n)) c

Conclusions

There are a lot of Merkle tree constructions for different purposes. They have their pros and cons. If we need a simple tree implementation, then the best choice is the sparse Merkle tree (SMT). If the purposeof a tree is to have short proofs for big bandwidth, then the best choices are the Verkle Tree (VT) and the radix Merkle tree (RMT). If the purpose of a tree is to store a huge amount of data, then the best choices are sparse Merkle tree (SMT) and indexed Merkle tree (IMT). If we are constantly modifying the tree, or we need fast proofs having nearly unlimited memory on a Prover and Verifier side, then the best choice is the radix Merkle tree (RMT). And finally, if we are lacking in memory, then the standard Merkle tree could be the best choice. For the best performance, it is better to implement the mentioned in a section 5 improvements if possible.

Declaration on Generative AI

While preparing this work, the authors used the AI programs Grammarly Pro to correct text grammar and Strike Plagiarism to search for possible plagiarism. After using this tool, the authors reviewed and edited the content as needed and took full responsibility for the publication's content.

References

[1] R. Merkle. A Digital Signature based on a Conventional Encryption Function, in: Advances in Cryptology — CRYPTO '87. CRYPTO 1987, vol. 293, 1987, 369–378. doi:10.1007/3-540-48184-2_32

[2] D. Rasmus, P. Tobias, P. Roel, Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs, Cryptology ePrint Archive, Paper 2016/683.

[3] R. Chandran, Pros and Cons of Merkle Tree, in: Artificial Intelligence and Communication Technologies, 2023, 649–653. doi:10.52458/978-81-955020-5-9-61

[4] V. Zhebka, et al., Methodology for Choosing a Consensus Algorithm for Blockchain Technology, in: Digital Economy Concepts and Technologies Workshop, DECaT, vol. 3665, 2024, 106–113.

[5] M. Saqib Niaz, G. Saake, Merkle Hash Tree based Techniques for Data Integrity of Outsourced Data, in: Workshop Grundlagen von Datenbanken, vol. 1366, 2015, 66–71.

[6] Q. Wang, et al., Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 847–859. doi:10.1109/TPDS.2010.183

[7] F. Kipchuk, et al., Assessing Approaches of IT Infrastructure Audit, in: IEEE 8th Int. Conf. on Problems of Infocommunications, Science and Technology, 2021. doi:10.1109/picst54195.2021.9772181

[8] A. Riadh, et al., Data Integrity Time Optimization of a Blockchain IoT Smart Home Net- work Using Different Consensus and Hash Algorithms, Wireless Commun. Mobile Comput. 2021(1) (2021). doi:10.1155/2021/4401809

[9] A. Riadh, et al., Peer-to-Peer User Identity Verification Time Optimization in IoT Blockchain Network, Sensors 23 (2023). doi:10.3390/s23042106

[10] J. Kuszmaul, Verkle Trees, 2019. URL: https://api.semanticscholar.org/Corpus

[11] N. Garg, S. Bawa, RITS-MHT: Relative Indexed and Time Stamped Merkle Hash Tree based Data Auditing Protocol for Cloud Computing, J. Netw. Comput. Appl. 84 (2017) 1–13. doi:10.1016/j.jnca.2017.02.005

[12] Y. Kostiuk, et al., Models and Algorithms for Analyzing Information Risks during the Security Audit of Personal Data Information System, in: 3rd Int. Conf. on Cyber Hygiene & Conflict Management in Global Information Networks (CH&CMiGIN), vol. 3925, 2025, 155–171.

[13] P. Rogaway, T. Shrimpton, Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance, In: Fast Software Encryption, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, 371–388.

[14] D. Catalano, D. Fiore, Vector Commitments and their Applications, Cryptology ePrint Archive, Paper 2011/495.
[15] R. Patgiri, M. Dutta Borah, HEX-BLOOM: An Efficient Method for Authenticity and Integrity Verification in Privacy-Preserving Computing, Cryptology ePrint Archive, Paper 2021/773.
[16] R. C. Merkle, A Digital Signature Based on a Conventional Encryption Function, Advances in Cryptology, CRYPTO'87. Lecture Notes in Computer Science, vol. 293, 1988, 369–378. doi:10.1007/3-540-48184-2_32
[17] M. Gracy, B. Jeyavadhanam, MTTBA- A Key Contributor for Sustainable Energy Consumption Time and Space Utility for Highly Secured Crypto Transactions in Blockchain Technology, 2022. doi:10.48550/arXiv.2209.13431
[18] M. Gracy, R. Jeyavadhanam, A. A. Raj, Enhancing transaction verification through pruned merkle tree in blockchain, J. Theor. Appl. Inf. Technol. (2023) 6421–6433.
[19] Haitz S´aez de Oc´ariz Borde, An Overview of Trees in Blockchain Technology: Merkle Trees and Merkle Patricia Tries, 2022.
[20] A. Kudzin, et al., Scaling Ethereum 2.0s Cross-Shard Transactions with Refined Data Structures, Cryptography 6 (2022). doi:10.3390/cryptography6040057
[21] Z. Gao, Y. Hu, Q. Wu, Jellyfish Merkle Tree, diem, 2021.
[22] H. Schoenfeld, Dynamic Merkle-Trees, Vixra.org, 2023.
[23] ZIDEN.IO, Indexed Merkle Tree — A More Optimized Solution for ZK proofs, 2023. URL: https://blog.ziden.io/indexed-merkle-tree-a-more-optimized-solution-for-zk-proofs-c75b0d0b1786
[24] Aztec, Indexed Merkle Tree, 2023. URL: https://docs.aztec.network/aztec/concepts/storage/trees/indexed_merkle_tree
[25] Y. Kostiuk, et al., Effectiveness of Information Security Control using Audit Logs, in: Cybersecurity Providing in Information and Telecommunication Systems, vol. 3991, 2025, 524–538
[26] M. Szydlo, Merkle Tree Traversal in Log Space and Time, in: Advances in Cryptology, EUROCRYPT, 2004, 541–554.
[27] M. Jakobsson, et al., Fractal Merkle Tree Representation and Traversal, in: Topics in Cryptology — CT-RSA 2003. CT-RSA 2003. Lecture Notes in Computer Science, vol. 2612, 2003, 314–326. doi:10.1007/3-540-36563-X_21
[28] P. Berman, M. Karpinski, Y. Nekrich, Optimal Trade-off for Merkle Tree Traversal, Theor. Comput. Sci. 372.1 (2007), 26–36. doi:10.1016/j.tcs.2006.11.029
[29] O. Kuznetsov, et al., Adaptive Restructuring of Merkle and Verkle Trees for Enhanced Blockchain Scalability, arXiv, 2024. doi:10.48550/arXiv.2403.00406
[30] G, Kaempfer, StarkWare: Recursive STARKs, 2022 URL: https://medium.com/starkware/recursive-starks78f8dd401025
[31] O. Kuznetsov, et al., Efficient and Universal Merkle Tree Inclusion Proofs via OR Aggregation, arXiv, 2024. doi:10.48550/arXiv.2405.07941