# Jellyfish Merkle Tree

Zhenhuan Gao, Yuxuan Hu, Qinfan Wu*

**Abstract.** This paper presents Jellyfish Merkle Tree (JMT), a space-and-computation-efficient sparse Merkle tree optimized for Log-Structured Merge-tree (LSM-tree) based key-value storage, which is designed specially for the Diem Blockchain. JMT was inspired by Patricia Merkle Tree (PMT), a sparse Merkle tree structure that powers the widely known Ethereum network. JMT further makes quite a few optimizations in node key, node types and proof format to find the ideal balance between the complexity of data structure, storage, I/O overhead and computation efficiency, to cater to the needs of the Diem Blockchain. JMT has been implemented in Rust, but it is language-independent such that it could be implemented in other programming languages. Also, the JMT structure presented is of great flexibility in implementation details for fitting various practical use cases.

## 1 Introduction

Merkle tree, since introduced by Ralph Merkle [1], has been a de facto solution to productionized account-model blockchain systems. Ethereum [2] and HyperLedger [3], are examples with various versions of Merkle trees.

Although Merkle tree fits pretty well as an authenticated key-value store holding a huge amount of data in a tamper-proof way, its performance in terms of computation cost and storage footprint are two major concerns where people have been trying to achieve some enhancements. Among them, the sparseness of Merkle trees [2], [4], [5] has been aptly leveraged to cut off unnecessary complexity in more compact and efficient Merkle tree design. Virtually, a Merkle tree in reality has far fewer leaf nodes at the bottom level, a far cry from the intractable size that can be held by a perfect Merkle tree. If sticking rigorously to the original Merkle tree structure, any design will by no means escape from excessive computation and space overhead.

The optimizations offered by the sparseness can also benefit other related data, notably proofs [6]. With sparseness, and its resultant smaller tree sizes, shorter proofs can be properly designed with less CPU time in generation and verification, and less network bandwidth in transmission, benefiting both server and client sides.

Based on earlier works in the industry, we present JMT, a space-time win-win optimized sparse Merkle tree designed specially for such blockchain systems as the Diem Blockchain [7], [8]. JMT is built on top of an LSM-tree based key-value storage, featuring version-based key that circumvents heavy I/O brought about by the randomness of a pervading hash-based key.

* The authors work at Novi Financial, a subsidiary of Facebook, Inc., and contribute this paper to the Diem Association under a Creative Commons Attribution 4.0 International License.

Revised January 12, 2021

This mini whitepaper is structured as follows: First, we give a concise retrospection of Merkle trees in practical uses cases where each leaf node is addressable via a key, followed by the variation and evolvement (Section 2). We then present JMT, including node key structure and its implicit benefits, logical data structures of two node types and related operations (Section 3). We then describe the proof format of JMT with the verification algorithm, paired with examples (Section 4). Finally, we conclude and discuss our future works (Section 5).

## 2 A Retrospection of Addressable Merkle Trees

Merkle trees are widely adopted by the blockchain industry as a cryptographically authenticated, deterministic data structure that can be used to store and map between arbitrary binary data. In the family of Merkle trees, an addressable Merkle tree, becomes the de facto standard of authenticated data structure that keeps record of the global blockchain state. We presume the readers of this paper are familiar with the background of Merkle trees and how Merkle trees are used to allow efficient and secure verification of the contents of large data structures with proofs.

### 2.1 Addressable Merkle Tree (AMT)

An authenticated data structure allows a verifier $V$ to hold a short authenticator $a$, which forms a binding commitment to a larger data structure $D$. An untrusted prover $P$, which holds $D$, computes $f(D) \to r$ and returns both $r$ — the result of the computation of some function $f$ on $D$ — as well as $\pi$ — a proof of the correct computation of the result — to the verifier. $V$ can run $\mathsf{Verify}(a, f, r, \pi)$, which returns true if and only if $f(D) = r$. In the context of the Diem Blockchain, provers are generally validators and verifiers are clients executing read queries.

In summary, an addressable Merkle tree is an authenticated data structure as a form of a binary Merkle tree that stores maps. In a Merkle tree of size $2^h$, the structure $D$ maps every key $i \in [0, 2^h)$ to a value $v_i$. The authenticator is formed from the root of a full binary tree created from the values, labeling leaves as $\mathtt{hash}(i \,\|\, v_i)$ and internal nodes as $\mathtt{hash}(\text{left} \,\|\, \text{right})$, where $\mathsf{H}$ is a cryptographic hash function. Keys are encoded as the binary paths from the root to each leaf which are referred to as *keys* in production.

[1] The function $f$, which the prover wishes to authenticate, is an inclusion proof that a key-value pair $(k, v)$ is within the map $D$.

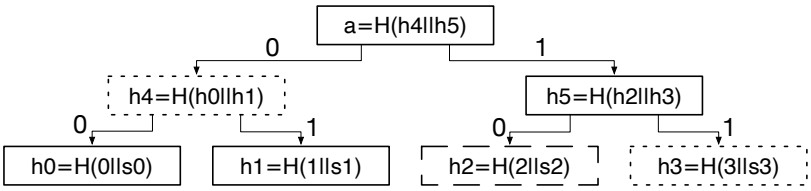



Figure 1: A Merkle tree storing $D = \{0 : \mathsf{s0}, ...\}$. If $f$ is a function that gets the third item (shown with a dashed line) then $r = \mathsf{s2}$ and $\pi = [\mathsf{h3}, \mathsf{h4}]$ (these nodes are shown with a dotted line). $\mathsf{Verify}(a, f, r, \pi)$ verifies that $a \stackrel{?}{=} \mathtt{hash}(\mathsf{h4} \,\|\, \mathtt{hash}(\mathtt{hash}(2 \,\|\, r) \,\|\, \mathsf{h3}))$.

$P$ authenticates lookups for an item $i$ in $D$ by returning a proof $\pi$ that consists of the labels of the sibling of each of the ancestors of node $i$. Figure 1 shows an AMT with 2-bit address where the

[1] Secure Merkle trees must use different hash functions to hash the leaves and internal nodes to avoid confusion between the two types of nodes. While we have omitted this detail in the example for simplicity, the Diem Core uses a unique hash function to distinguish between different hash function types to avoid attacks based on type confusion [?].

addresses of nodes in binary are `0x00`, `0x01`, `0x10`, `0x11`, respectively. The addressing process is also straightforward: Following each bit of the address from MSB to LSB, if it reaches a 0, visit the left child of the current node; otherwise, the right child. Figure 1 also illustrates a lookup for the $3^{rd}$ item which is surrounded by dotted lines, and the nodes included in $\pi$ are shown with dashed lines.

In this way, the root digest, $a$, efficiently authenticates all the data stored at all the leaf nodes, addressed by their keys, altogether by recursively hashing the concatenation of the hashes of the two children.

#### 2.1.1 Tractable Representations

While a full tree with $h$-bit keys (256 bits in Diem) of size $2^h$ is an intractable representation when $h$ is a large value, Figure 2 shows two optimizations that can be applied to transform a naive binary implementation (❶) into an efficient one for a sparse Merkle tree. First, subtrees that consist entirely of empty nodes are replaced with a placeholder value (□ in ❷) whose digest is a predefined constant $D_{digest}$ as used in the certificate transparency system [9]. This optimization creates a representation of a tractable size without substantially changing the proof generation of the Merkle tree.

However, leaves are always stored at the bottom level of the tree, meaning that the structure requires $h$ hashes to be computed on every leaf modification. A second optimization replaces subtrees consisting of exactly one leaf with a single node (❸). In expectation, the depth of any given item is $O(\log n)$, where $n$ is the number of items in the tree. This optimization reduces the number of hashes to be computed when performing operations on the map including proof-related operations.

Some works [5] create an identical default digest only for the empty nodes on the same height $h$ and let $D^h_{default} = H(D^{h-1}_{default} \,\|\, D^{h-1}_{default})$. In this paper, we choose a single default digest $D_{default}$ to represent empty subtrees at any level without ambiguity, especially in our design.

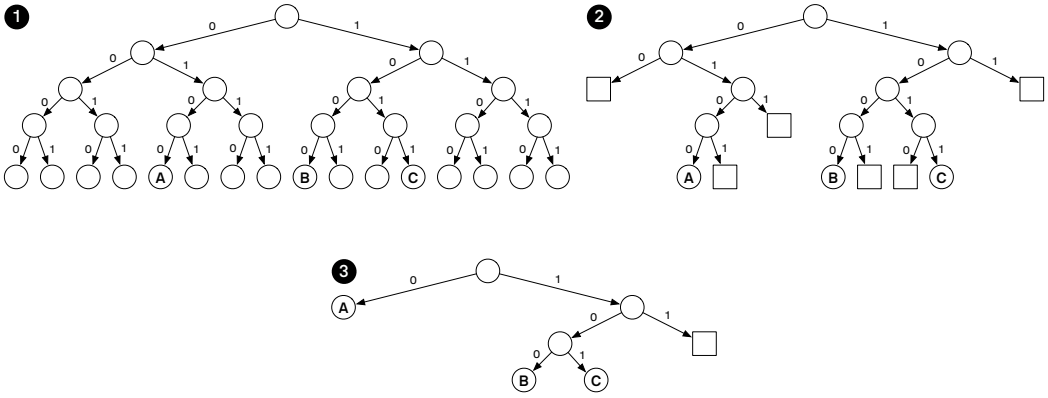



Figure 2: Three versions of a sparse Merkle tree storing $D = \{0100_2 : \mathsf{A}, 1000_2 : \mathsf{B}, 1011_2 : \mathsf{C}\}$.

## 2.2 Addressable Radix Merkle Tree (ARMT)

The same architecture could be extended to a more generalized radix tree since AMT is just a special case with radix $r = 2$. In turn, an ARMT is the compressed version of the corresponding AMT such that every $log_2(r)$ levels of the original AMT are compressed into a single level where each node has maximum $r$ children, though some children may not exist. We will use $\mathrm{AR_rMT}$ to denote an addressable Merkle tree with radix $r$. Notwithstanding $r = 2^k, k \in \mathbb{N}^+$, most ARMT designs choose $\mathrm{AR_{16}MT}$ in production, i.e., each internal node has up to 16 children.

The choice is mostly attributed from the following tradeoff between storage and I/O overhead because an update of a single leaf node can require all the nodes on the path to the root to be updated together.

- Larger $r$ leads to more storage cost for each update in that internal (non-leaf) node size is proportional to $r$ so each update of a leaf node will update all the nodes on the path to the root, resulting in greater write amplification.
- Smaller $r$ means a greater height of the tree that further increases I/O when querying or updating the tree. For example, to traverse 4 levels down, a radix-16 tree only needs one read but a binary tree requires four reads. For a read-heavy service such as Diem Core, `4x` reads can add up to a considerable burden to the underlying storage medium.

As long as this rule is honored, whatever $r$ could be applied to the physical representation/implementation of ARMT to adjust the tradeoff between I/O and space cost to fulfil different requirements.

Figure 3 illustrates an $\text{AR}_{16}\text{MT}$ with 8-bit keys. It is noted that when the tree is sparse, nodes may have some empty children representing non-existence. In this figure, there are only four keys: `0x14`, `0x1A`, `0x1F`, and `0xD5`. Instead of bit by bit, traversing downward consumes one nibble at a time. Similarly, an empty node □ indicates an empty subtree as aforementioned.

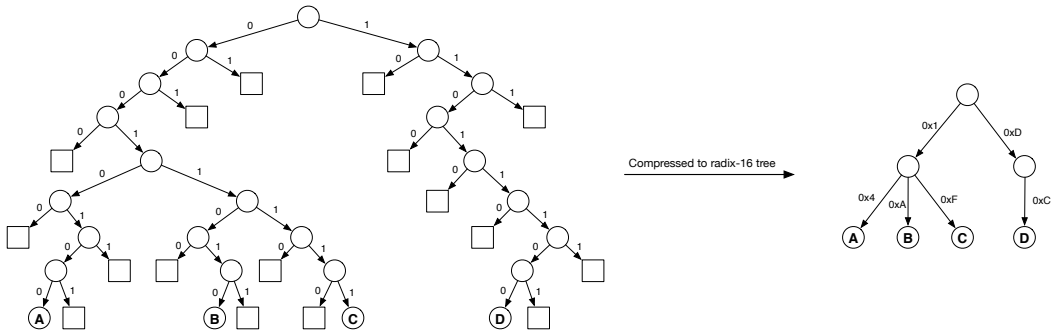



Figure 3: AMT compressed to $\text{AR}_{16}\text{MT}$.

## 2.3 Use case: State Tree

In most mainstream blockchain architectures with an account based model, including the Diem Blockchain, the global state at a specific time represents all the account information of the blockchain at that time. The blockchain itself can be viewed as a state machine where the state can be updated via some transaction execution mechanism.

It is the states at different versions of this state machine that have meaningful data engraved in the chain and agreed upon by all the users. In Diem, a *state tree* showing a global state $S_i$, represents the state of all accounts at version $i$ as a map of key-value pairs, on the assumption that a fork is not allowed. We use *versioning* to distinguish the states after applying each transaction, so $S_i$ is the state of Diem Blockchain after applying the $i$-th transaction. Keys are based on the 256-bit account addresses, and their corresponding value is the authenticator of the account. Therefore, a state tree is a perfect use case of AMT where each leaf node can represent an account with its data and the account address can be uniquely linked to the key of that leaf node.

# 3 Jellyfish Merkle Tree

To hold the global state of the Diem Blockchain, we propose a modified version of $AR_{16}MT$, named Jellyfish Merkle Tree (JMT), with the following features:

- **Version-based Node Key** In lieu of directly adopting keys as raw node keys to lookup nodes from *Storage*, JMT chooses a version-based key schema with multi-fold benefits:
  - Facilitating version-based sharding.
  - Greatly lowering compaction overhead in LSM-tree based storage engines such as RocksDB [10].
  - Smaller key size on average.
- **Less Complexity** JMT has only two physical node types, *Internal Node* and *Leaf Node*.
- **Concise Proof Format** The number of sibling digests in a JMT proof is less on average ($\Theta(log(\textit{number of existent leaves}))$) than that of the same ARMT without optimizations ($log(\textit{number of maximum leaves})$, i.e., the height of the equivalent AMT), requiring less computation and space.

## 3.1 Versioning

Usually, AMT is utilized as a single-version database. In blockchain systems like Diem, a version number is an monotonically increasing unsigned integer that represents the number of transactions the system has executed and applied to the global state authenticated via an AMT. Since this paper is oriented to be application independent, we define *version* as an monotonically increasing unsigned integer used to identify different AMTs between updates so each update bumps the version by one. The new tree reuses unchanged portions generated at previous versions, forming a *persistent data structure* [11], [12]. If an update modifies $m$ out of $n$ leaves, on average $O(m \cdot \log n)$ new nodes are created in the tree that differ from the previous version. This approach allows any upper-layer application to store multiple versions of the state efficiently by only recording the "delta". This feature also allows for efficient recomputation of the state authenticator after performing an update.

## 3.2 Core Building Blocks

To be short, JMT is a modified $AR_{16}MT$ to which the two optimizations introduced in Section 2.1.1 are applied. Moreover, distinguished from the other productionized Merkle tree designs, it has a simpler structure with special node key schema leveraging versioning.

### 3.2.1 Node Key

Since nodes are serialized and stored in a key-value store that supports point lookup, each node is associated with a unique key identifying that node in Storage. JMT adopts a version-based node key schema, by splicing *version* and *nibble path*, as:

$$\texttt{version} \,\|\, \texttt{nibble path}$$

where the node of this key is created at `version` and `nibble path` is the sequence of nibbles on the path from the root node to this node following the given key. At any version, a nibble path itself can uniquely identify a node. Therefore, combining version and nibble path could similarly pinpoint any node across versions from the whole blockchain history. In Diem Core, we assign unsigned 64-bit

integers to version type encoded in big endian, but there can be any reasonable numeric types; Nibble path is encoded in a way that nibbles near to the root show first.

There are two advantages of the version-based node key over widely adopted hash-based node key:

- Compared to a fixed-length hash-based node key, a JMT node key takes less space in a sparse Merkle tree. For a JMT of 256-bit keys and 1 billion leaves, the average height is around 8 nibbles. The average size of all the node keys is about 12 bytes, significantly smaller than 32 bytes of a 256-bit hash key.
- We choose RocksDB as the underlying storage engine for Diem, similar to LevelDB [13], a LSM-Tree key-value storage engine. The versioned JMT could remarkably reduce compactions to zero, thereby achieving the optimal write amplification as one (not including write-ahead log). In LevelDB or RocksDB, all the data is stored as key-value pairs and all keys are sorted by a predefined order, usually lexicographic or reverse-lexicographic order. If a hash key is adopted, each insertion will insert a new key-value pair at a random position within the current key spectrum. But given the JMT node key schema, we could insert the new nodes generated by each version sequentially to append to the current key set in storage according to the lexicographic order because our key schema ensures keys of a high version are always lexicographically greater than that at a lower version. In this case, compaction is no longer necessary as the keys inserted are already ordered. Our experiment shows this schema saves IOPS and disk bandwidth by more than 90 in contrast to hash-based node keys.

#### 3.2.2 Node Types

JMT is made of two kinds of nodes[2]:

- **Internal Node** is an interior node that has at least one child node, as a stepping-stone representing one nibble on the path to leaf nodes. Same as a normal $\text{AR}_{16}\text{MT}$, each internal one can hold up to 16 children, compressing a subtree of 4 levels of AMT into one node. The structure contains indices, versions, and digests of all the child nodes. If any subtree represented by a child is empty, the internal node will leave its slot empty to denote an empty node. For example, if an internal node with key $30 \| \texttt{0x996F}$ has 4 child nodes, its structural layout can look like:

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Version | 12 | | | | | | | 9 | | | | 25 | | | | 18 |
| Digest | $d_0$ | | | | | | | $d_7$ | | | | $d_B$ | | | | $d_F$ |

  where $d_i$ denotes the digest of the child node at index $i$. In this example, the node key of the child node at index `0xB` would be $25 \| \texttt{0x996FB}$. Later in this section we will cover the details about constructing the node key of the next node.

- **Leaf node** is a node that stores user value at the bottom of the tree. Besides the data, it also contains the `key` used for querying the tree of the node and the digest of the data. The `nibble path` field of a leaf node key must be a prefix of its `key`.

**Lookup** When querying the tree at version $v$ with key $k$ to get the associated value, the following steps are performed:

[2] For implementation, two node types only are not able to gracefully handle a special case when the whole tree is empty. Diem Core adopt a third node type called 'Null' expressly for this case but it is definitely not the only workaround.

1. Get the root node key by an "out-of-band" method, where we use $v$ as the root node key in Diem Core, and as the next node key.
2. Query the database to get the next node by its key.
3. Check the node type:
    - If the node is an internal node, read the next nibble $n$ in $k$ as the index to find the next version. If the slot is empty, the node doesn't exist. Otherwise, splice the nibble path of the node and the nibble just read as the next nibble path, i.e., *current nibble path* $\|\, n$. Finally, splice the next version and the next nibble path to form the next node key. Then goto step 2.
    - If the node is a leaf node, read the stored `key` and check whether it matches the $k$. If matches, return the value stored. Otherwise, the node of $k$ does not exist.

**Insertion and Update** Insertion shares similar steps as lookup with slightly different behaviors. Suppose we are going to insert data $d$ with key $k$ at version $v$.

1. Follow lookup steps until find a leaf node, or an internal node where the next nibble slot to be visited of index $n$, is empty.
2. Check the current node type:
    - **Internal Node:** Create a new leaf node from $k$ and $d$ with its node key as $v \,\|\, \textit{current nibble path} \,\|\, n$. Fill `version` as $v$ and `digest` as $\texttt{hash}(k \,\|\, \texttt{hash}(d))$ into the current internal node at `index` $n$ .
    - **Leaf Node:** Check whether $k$ equals the key of the node,
        - If the keys mismatch, the new leaf node will be created likewise. Moreover, a series of cascading internal nodes will be created one by one to represent the shared nibble path within all unvisited nibbles of both keys. Also, both leaf nodes will be both grafted onto the bottom one as child nodes. Afterwards, the uppermost new internal node will be positioned in place of the old leaf node in its parent.
        - If the keys match, the insertion becomes an update. We just have to replace the leaf node with its new value $d$ and update its node key with new version $v$.
3. Until we finish step 2, since we have reached the deepest nodes of a tree, all the updates involving version changes will affect addressing so we have to modify all the ancestor nodes visited. This means from the last leaf node inserted/updated, update its parent internal nodes one by one from the bottom up with the new version and digests of all modified child nodes until the root node.

#### 3.2.3 Taking Extension Node Away

It is noted that we expressly abandon an extension node, introduced by PMT [2], for the reasons below:

- The efficacy brought about by extension nodes will diminish quickly as the tree size grows and in turn becomes less and less sparse. The chance that two leaf nodes share a long common prefix of keys is so rare that the case where an extension node can substitute a long chain of internal nodes is very uncommon.
- Removing the extension node effectively contributes to less complicated code in favor of less potential bug surfaces.

### 3.3 Miscellaneous

Besides fundamental building blocks and operations, JMT can also support a range proof to validate the existence of a set of consecutive leaf nodes. Moreover, with proper design, JMT stands ready and flexible for functionality-wise extensions such as pruning, backup and restore, that are already implemented in Diem Core.

## 4 Proof Format and Verification

Since ARMT is just a physical representation of AMT, irrespective of $r$, the proof format will always be indistinguishable from that of the equivalent AMT.

According to the first optimization mentioned in Section 2.1.1, the proof of a node has been simplified by replacing all empty subtrees as empty nodes with placeholder digest $D_{digest}$. JMT adopts this simplification in the design likewise.

Whereas this format serves well for a dense Merkle tree, it proves verbose and in turn inefficient for a sparse tree attributed to obsessive default digests. For example, if a $h$-bit tree has only one leaf node, the proof still includes $h$ default digests. In fact, a key and only one default digest are adequate to prove its existence. Therefore, similar to [6], we further improve the proof format in JMT to collapse consecutive levels of empty siblings into one, shrinking the nibble path of the node. As a result, the JMT optimized Merkle proof could be classified into three cases under two proof categories:

- Proof of Inclusion
  - A leaf node of the key exists.
- Proof of Exclusion
  - A leaf node of a different key exists, but with the same prefix of the queried key. Counterintuitively, its existence indirectly proves the queried node does not exist.
  - An empty node is on the path to the node during lookup. This means that the corresponding position specified by the key currently belongs to an empty subtree, i.e., the queried node doesn't exist.

Then proof format can be summarized in `C++` syntax as:

```
struct Leaf {
    HashValue address;
    HashValue value_hash;
};

struct Proof {
    Leaf* leaf;
    std::vector<HashValue> siblings;
};
```

Table 2: Proof format of Jellyfish Merkle Tree

In this format, `HashValue` is the digest type that is the output from the `hash` function. `leaf` in proof consists of the node key `key` and the digest of the value stored, `value_hash`. `siblings` denotes that there are digests that can be hashed with the node digest iteratively to obtain the root digest, upper levels first. algorithm 1 shows how to verify a proof.

**Algorithm 1:** JMT Proof Verification

**Input :** Node key $k$, proof $p$, the expected root digest $d_{root}$, and the node value $v$ that needs to be verified against.

**Output:** return true if the $v$ is verified by $p$ and vice versa

```
if v ≠ NULL then                                  // Node existence expects inclusion proof
    if p.leaf ≠ NULL then
        // Prove inclusion with inclusion proof
        if k ≠ p.leaf.key or hash(blob) ≠ p.leaf.value_hash then
            return false
        end
    else          // Expected inclusion proof but get non-inclusion proof passed in
        return false
    end
else                                              // Node absense expects exclusion proof
    if p.leaf ≠ NULL then                          // The inclusion proof of another node
        if k = p.leaf.key or CommonPrefixLengthInBits(p.leaf.key) < Len(p.siblings) then
            return false
        end
    else                                         // Prove exclusion with an exclusion proof
        // Noop
    end
end
if p.leaf = NULL then d_cur ← D_default else d_cur ← hash(p.leaf)
for i ← DigestLengthInBits − Len(p.siblings) − 1 to 0 do
    if the i^th bit from MSB of k = 1 then
        d_cur ← hash(p.siblings[i] ‖ d_cur)
    else
        d_cur ← hash(d_cur ‖ p.siblings[i])
    end
end
return d_cur = d_root
```

We will give an example of each case to elaborate how JMT proof works. The assumption is that we want to verify a proof against the node with key $k_n$ = `0b1000` and data value $v_n$ = `0x1234`. It is worth noting that we exemplify the proof verification algorithm by using 4-bit paths instead of full nibble paths to help with understanding, because one nibble represents 4 bits and proof only concerns bits but not nibbles. In these examples, all the `siblings` are marked grey in figures and their digests are denoted as $d_{path}$ where *path* is represented in binary notation.

## 4.1 Proof of inclusion

Figure 4 gives an example proof showing node with `key=0b1000` exists. The proof has `leaf.key = 0b1000`, `leaf.value_hash = hash(0x1234)` and `siblings =` $[d_{101}, d_{11}(D_{default}), d_0]$.

Then we can prove that the node with key `0b1000` exists by:

1. Check *leaf* $\neq$ `NULL` and *leaf.key* $=$ *key*. So if it is a valid proof, it must be an inclusion proof.
2. Get the first `Len`(*leaf.siblings*) bits of $k_n$, i.e., `0b100`, which is the binary path of the target leaf node.

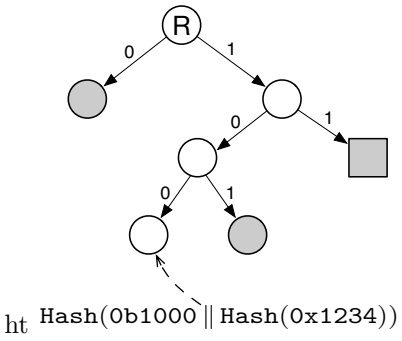


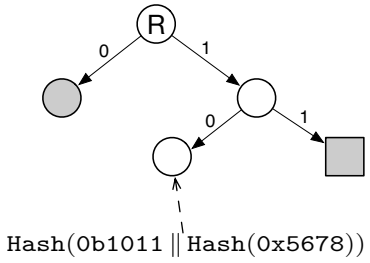

Figure 4: Proof of Inclusion

3. Then we get $d_{100} = \texttt{hash}(k_n \,\|\, \texttt{hash}(v_n))$, and based on the $3^{rd}$ bit from MSB is 0, we get $d_{10} = \texttt{hash}(d_{100}, d_{101})$; similarly, $d_1 = \texttt{hash}(d_{10}, D_{default})$ and root node digest $d_R = \texttt{hash}(d_0, d_1)$.
4. Compare the $d_R$ with the expected. If they are the same, the proof is verified and not vice versa.

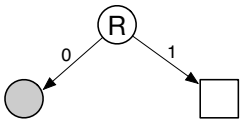
## 4.2 Proof of Exclusion

There are two possible cases, both of which can prove the exclusion of the node. Either another node of a different but having a common prefix key already exists or an empty node exists whose nibble path is a prefix of the query key. The definition may seem obscure, so it's better to give two examples to untangle the complexities between these two cases.

### 4.2.1 Another node existence




(a) Proof of exclusion with another leaf node




(b) Proof of exclusion with empty node on the path

Figure 5: Proofs of Exclusion

Figure 5a gives an example proof that a node with key `0b1011` and value `0x5678` exists. The proof has `leaf.key = 0b1000`, `leaf.value_hash` = $\texttt{hash}(\texttt{0x5678})$ and `siblings` = $[d_{11}(D_{default}), d_0]$. Then we can also use it to prove that the node with key `0b1000` does not exist by:

1. Check *leaf* $\neq$ `NULL` in the proof and *leaf.key* $\neq k_n$. So if it is a valid proof, it must be a non-inclusion proof.
2. Similarly, check the first 2 (number of the siblings) bits of the two keys are identical. Otherwise it is not a valid proof. In our case the bits are `0b10`.
3. Then we get $d_{10} = \texttt{hash}(\textit{leaf.key} \,\|\, \textit{leaf.value\_hash})$, and based on the $2^{nd}$ bit is 0, we get $d_1 = \texttt{hash}(d_{10}, D_{default})$. Finally, $d_R = \texttt{hash}(d_0, d_1)$.
4. Compare $d_R$ with the expected.

#### 4.2.2 Empty node existence

Figure 5b illustrates an example proof showing an empty node exists with path $0b1$, which in turn proves no node with `0b1`-prefixed key exists, including the target node. The proof has `leaf = NULL` but `siblings =` $[d_1(D_{default})]$).

1. Check *leaf* = `NULL`. So if it is a valid proof, it must be a non-inclusion proof.
2. Check the first 1 (number of siblings) bits from MSB of both keys are identical. Otherwise it is not a valid proof. In our case the bits are `0b1`.
3. Then we get $d_1 = D_{digest}$, and based on the $1^{st}$ bit is 1, we can get $d_R = \texttt{hash}(d0, D_{digest})$.
4. Compare $d_R$ with the expected.

### 4.3 Proof Generation

The proof generation can piggyback on a lookup. It is as simple as collecting all the siblings when performing a lookup into a PMT. After a lookup, no matter whether the node of the lookup key exists, the collected siblings on the path with the final node reached, empty or not, make up the final proof that could be verified against its existence or absence.

## 5 Conclusion

We have presented Jellyfish Merkle Tree, an architecture of sparse Merkle tree optimized for computation and space, designed for the Diem Blockchain. Compared to other Merkle trees in production, JMT simplifies node types with only two types and leverages the sparseness to benefit LSM-tree based storage. The space saving can be optimized further if delta encoding is enabled for keys. Moreover, though the proof format and verification algorithm has become more complex, it is the smaller proof size and the less computation overhead of verification that practically benefit users while keeping the algorithm complexity transparent to end users.

### 5.1 Future Work

The JMT node key structure and encoding are expressly optimized for LSM-tree key-value store to avoid compaction. However, when we incorporate pruning features that enable deletion of expired data that out of a recent window, compaction is inevitable since the key of a tombstone will definitely overlap with the current sorted runs. Then the benefit will be greatly diminished by compactions triggered. We may need to look into a smarter way to relieve the overhead brought about by it.

In the long run, we hope our efforts on precise specifications could help the evolvement of practical authenticated data structure or inspire better designs and implementations.

## References

[1] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in cryptology - CRYPTO '87, A conference on the theory and applications of cryptographic techniques, santa barbara, california, usa, august 16-20, 1987, proceedings*, 1987, pp. 369–378.

[2] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger." http://gavwood.com/paper.pdf, 2016.

[3] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on distributed cryptocurrencies and consensus ledgers*, 2016.

[4] B. Laurie and E. Kasper, "Revocation transparency," *Google Research, September*, p. 33, 2012.

[5] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse merkle trees," 2016, pp. 199–215.

[6] H. Park, "Modified merkle patricia trie specification (also merkle patricia tree)." 2018. https://medium.com/aergo/releasing-statetrie-a-hash-tree-built-for-high-performance-interoperability-6ce0406b12ae

[7] The Diem Association, "An Introduction to Diem." https://diem.com/en-us/white-paper/.

[8] Z. Amsden *et al.*, "The Libra Blockchain." https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper.

[9] B. Laurie, "Certificate transparency," *Communications of the ACM*, vol. 57, no. 10, pp. 40–46, 2014.

[10] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *CIDR 2017, 8th biennial conference on innovative data systems research, chaminade, ca, usa, january 8-11, 2017, online proceedings*, 2017. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, pp. 86–124, 1989.

[12] C. Okasaki, "Purely functional data structures," 1999.

[13] S. Ghemawat and J. Dean, "LevelDB." 2011. https://github.com/google/leveldb