Project 4 —问题三：基于 SM3 的 Merkle 树（RFC6962 风格）构建与证明

自动生成

2025 年 8 月 14 日

摘要

本文档给出基于 SM3 的 Merkle 树（采用 RFC6962 风格的前缀区分：叶子与内部节点）的精确定义、构造算法、包含性证明（inclusion/audit path）和不存在性证明（non-inclusion）的构造方法。并针对 100,000 叶子给出实现时的工程估算、优化建议与复杂度分析，便于直接用于实验或报告。

目录

1 约定与符号

- $\mathrm{H}(\cdot)$ 表示底层哈希函数（本文采用 SM3），输出固定为 32 字节（256 位）。
- 连接（拼接）记作 $x\|y$。
- 设叶子原始数据为字节串 ℓ_i （$i = 0, \ldots, n-1$）。
- 使用 RFC6962 风格的字节前缀以区分叶子与内部节点：

  $$\mathrm{LeafHash}(\ell) := \mathrm{H}\left(0x00 \,\|\, \ell\right), \qquad \mathrm{NodeHash}(L, R) := \mathrm{H}\left(0x01 \,\|\, L \,\|\, R\right),$$

  其中 L, R 为长度固定的子节点哈希（32 字节）。
- 当层节点数为奇数时，本文采用 **复制最后节点**（duplicate last node）的约定进行配对计算：若当前层节点序列为 $(N_0, \ldots, N_{m-1})$ 且 m 为奇数，则把 N_{m-1} 与自身配对计算其父节点：

  $$\widetilde{N}_{\lfloor m/2 \rfloor} := \mathrm{NodeHash}(N_{m-1}, N_{m-1}).$$

  （注：某些实现也采用「提升/直接上提最后节点」策略，请与协议保持一致；RFC6962 的常见实现采用复制方式以保证定义上的对称性。）

2 树的构造（数学表示）

给定 n 个叶子 $\ell_0, \ldots, \ell_{n-1}$，构建过程如下：

1. 计算叶层哈希：

   $$L_i := \mathrm{LeafHash}(\ell_i), \qquad i = 0, \ldots, n-1.$$

2. 将叶层作为第 0 层：$\mathcal{L}^{(0)} = (L_0, \ldots, L_{n-1})$。对于每一层 $t \geq 0$，若当前层节点序列为 $\mathcal{L}^{(t)} = (N_0^{(t)}, \ldots, N_{m_t-1}^{(t)})$，其上一层（父层）按下列规则构造：

   $$N_j^{(t+1)} = \begin{cases} \mathrm{NodeHash}\left(N_{2j}^{(t)},\, N_{2j+1}^{(t)}\right), & 0 \leq 2j+1 < m_t, \\ \mathrm{NodeHash}\left(N_{2j}^{(t)},\, N_{2j}^{(t)}\right), & 2j+1 \geq m_t \text{ (复制最后节点)}. \end{cases}$$

3. 重复直到某层只剩一个节点，该节点即为树根（root）。

用迭代符号表示，若 $m_0 = n$，构造高度为 $h = \lceil \log_2 n \rceil$（上取整），最终根为

$$\mathrm{Root} = \mathcal{L}_0^{(h)}.$$

3 包含性证明（Inclusion / Audit Path）

3.1 定义

对于叶子索引 i，包含性证明 $\pi(i)$（也称审计路径）是由每层与目标节点配对的「兄弟节点哈希」序列组成：

$$\pi(i) = \big(s_0, s_1, \ldots, s_{h-1}\big),$$

其中 s_t 是第 t 层（从叶层 $t = 0$ 起）的目标节点 $N_{k_t}^{(t)}$ 的兄弟节点哈希（若该层目标节点是偶数索引则兄弟为右侧 $N_{k_t+1}^{(t)}$，若为奇数索引则兄弟为左侧 $N_{k_t-1}^{(t)}$）。在复制最后节点的策略下，即使某层节点被与自身配对，其兄弟值等于自身（因此包含在证明中即可）。

3.2 证明构造算法（伪代码）

```
// 输入: 叶索引 i (0-based), 叶层哈希数组 level0[0..n-1]
// 输出: proof = list of sibling hashes s_0..s_{h-1}
function build_inclusion_proof(i, level0):
    proof = []
    idx = i
    level = level0
    while len(level) > 1:
        if idx % 2 == 0:
            // 左子节点
            sibling_idx = idx + 1
            if sibling_idx < len(level):
                proof.append(level[sibling_idx])
            else:
                // 复制最后节点时, 兄弟为自身
                proof.append(level[idx])
        else:
            sibling_idx = idx - 1
            proof.append(level[sibling_idx])
        // 走向父层
        next_level = []
```

```
        for j in range(0, len(level), 2):
            left = level[j]
            if j+1 < len(level):
                right = level[j+1]
            else:
                right = left
            next_level.append(NodeHash(left, right))
        idx = idx // 2
        level = next_level
    return proof
```

3.3 验证算法（伪代码）

```
// 输入: leaf (原始叶子 bytes), index i, proof list, expected_root
function verify_inclusion(leaf, i, proof, expected_root):
    h = LeafHash(leaf)
    idx = i
    for s in proof:
        if idx % 2 == 0:
            h = NodeHash(h, s)
        else:
            h = NodeHash(s, h)
        idx = idx // 2
    return h == expected_root
```

4 不存在性证明（Non-inclusion）

严格而紧凑的非存在性证明通常依赖于**对叶键（或叶值）进行全序维护**。常见方案如下（适用于“按键排序”的情形）：

4.1 思想

若叶子集合按某个键（或叶值本身）排序为

$$k_0 < k_1 < \cdots < k_{n-1},$$

要证明某个键 k^* 不存在，可以找到排序后 $k_j < k^* < k_{j+1}$ 的相邻两个已存在键 k_j, k_{j+1}，并分别提供它们的包含性证明和对应的键值。验证者检查键值关系并验证这两个包含证据，从而断定 k^* 不在集合中。

边缘情况：

- 若 $k^* < k_0$，则提供 k_0 的包含证据并指出 k^* 位于该键左侧。
- 若 $k^* > k_{n-1}$，则提供 k_{n-1} 的包含证据并指出 k^* 位于该键右侧。

4.2 构造伪码（假设维护了按键排序的索引）

```
// 输入: 查询键 k_star, sorted_keys = [k_0..k_{n-1}], leaf_map: key
  ->leaf_bytes
function build_noninclusion_proof(k_star, sorted_keys, leaf_map):
    pos = binary_search_insert_position(sorted_keys, k_star)
    if pos < len(sorted_keys) and sorted_keys[pos] == k_star:
        return ("present", build_inclusion_proof(index=pos, ...))
    // neighbor indices:
    left_idx = pos - 1 if pos - 1 >= 0 else None
    right_idx = pos if pos < len(sorted_keys) else None
    proof = {}
    if left_idx is not None:
        proof["left_key"] = sorted_keys[left_idx]
        proof["left_inclusion"] = build_inclusion_proof(left_idx,
           ...)
    if right_idx is not None:
        proof["right_key"] = sorted_keys[right_idx]
        proof["right_inclusion"] = build_inclusion_proof(right_idx,
           ...)
    return ("absent", proof)
```

验证者：

- 验证提供的相邻键确实按排序条件将 k^* 包夹；
- 验证相邻键的包含证明（调用 `verify_inclusion`）；
- 若上述均成立，则 k^* 不存在于集合中。

5 针对 100,000 叶子的工程考量

5.1 规模估算

- 每个哈希输出：32 字节。
- 叶层存储：$n \times 32 = 100{,}000 \times 32 = 3{,}200{,}000$ 字节 ≈ 3.05 MiB。

- 整棵树理论上的节点数（如果完全存储所有节点）最多小于 $2n$，因此总内存约 $2n \times 32 \approx 6.1$ MiB，这对现代机器几乎是微不足道的。

5.2 树高与证明长度

$$h = \lceil \log_2 100{,}000 \rceil = 17 \quad (\text{因} 2^{17} = 131072).$$

因此包含性证明长度最多为 17 个 32 字节哈希，证明大小约 $17 \times 32 = 544$ 字节（不含索引/其他元数据）。

5.3 时间复杂度

- 构建树（自底向上）的哈希调用次数约为 $n - 1$（内部节点个数），因此时间复杂度为 $O(n)$ 次哈希。
- 单个包含性证明的构建复杂度为 $O(\log n)$（因为需沿路径收集 h 个兄弟哈希）。
- 验证包含性证明为 $O(\log n)$ 次哈希。

5.4 并行与批量优化建议

- **叶哈希并行化**：对 n 叶子，叶哈希（LeafHash）为独立任务，可用多线程/进程或批量 SIMD 并行化，显著提高吞吐。
- **分层并行**：每一层的节点 pair-wise 计算相互独立，也可并行计算；但需要同步等待上一层完成。
- **I/O 与内存**：尽量把所有中间哈希保存在连续数组(例如 C 中的 `uint8_t buf[2*n*32]`)，减少指针与对象开销，提高缓存命中率。
- **流式构建（外存/分块）**：如果叶子数远大于内存，可采用外存分块合并（类似归并排序 / map-reduce）方式构建根哈希。

6 证明传输与压缩

- 包含性证明大小：$\leq h$ 个哈希（$h \approx 17$），约 544 字节，适合网络传输。
- 非存在性证明（以邻居方式）：最多需传输两个包含性证明与两个键值（若键大小较小），总体大小通常在几 KB 以内。
- 可对证明进行进一步压缩（如可用交叉引用、去重或按层打包），以减少重复哈希的传输。

7 示例（小规模演示伪例）

Listing 1: Python 风格示例（伪代码）

```
# 假设有 leaves: list of bytes
def build_merkle(leaves):
    level = [leaf_hash(l) for l in leaves]
    all_levels = [level]  # optional: keep all levels
    while len(level) > 1:
        next_level = []
        for i in range(0, len(level), 2):
            left = level[i]
            if i+1 < len(level):
                right = level[i+1]
            else:
                right = left
            next_level.append(node_hash(left, right))
        all_levels.append(next_level)
        level = next_level
    root = level[0]
    return root, all_levels

# 包含性证明示例:
root, levels = build_merkle(leaves)
proof = build_inclusion_proof(index, levels[0])
ok = verify_inclusion(leaves[index], index, proof, root)
```

8 注意事项与实现细节

- **字节顺序**：在拼接 $0x00\|\ell$ 或 $0x01\|L\|R$ 时，必须明确字节序（哈希输出按字节序列传递，无需进一步大小端转换），并在所有实现中保持一致。
- **去重/缓存**：当批量生成多个证据时，上下游层可能重复使用相同内部节点哈希，建议缓存以节省计算。
- **按键排序**：若要支持紧凑且可证明的不存在性证明，必须在插入时维护叶子键的全序或使用外部数据结构（例如 B-tree / 有序数组）以便快速查找相邻键。
- **与 RFC6962 的一致性**：在系统设计阶段确认对奇数节点的处理策略（复制 vs 提升）与想要兼容的系统/标准保持一致。

9 小结

本节给出了基于 SM3（32 字节输出）的 Merkle 树构造与证明算法，说明了包含性与不存在性证明的数学表示、构造与验证方法，并对 100,000 叶子的实现给出内存/时间估算与优化建议。总体结论是：对于 100k 级别的叶子，完全在内存中构建与证明是非常可行且高效的；包含性证明长度仅约 17 个哈希（≈ 544 字节），非常紧凑，适合网络传输与验证。

若你需要，我可以接着为你：

- 把上述伪代码改写为可直接运行的 Python（含 SM3 实现）并在会话中构建 100k 叶子的 Merkle 树以给出实际耗时和根哈希；或
- 生成用于 Overleaf 的图示（例如树结构与包含路径示意图）；或
- 按你指定的「奇数节点处理策略」把算法与证明代码调整为完全一致的实现。