山东大学
SHANDONG UNIVERSITY

Project4(c) SM3 构建 Merkle 树

学院 网络空间安全

专业 网络空间安全

学号 202200460149

班级姓名 网安 22.1 班张弛

2025 年 8 月 6 日

目录

1 实验任务

Project 4: SM3 的软件实现与优化

a)：与 Project 1 类似，从 SM3 的基本软件实现出发，参考付勇老师的 PPT，不断对 SM3 的软件执行效率进行改进

b)：基于 sm3 的实现，验证 length-extension attack

c)：基于 sm3 的实现，根据 RFC6962 构建 Merkle 树（10w 叶子节点），并构建叶子的存在性证明和不存在性证明

2 SM3 构建 Merkle 树

2.1 Merkle 树

首先我们了解 Merkle 树是什么：

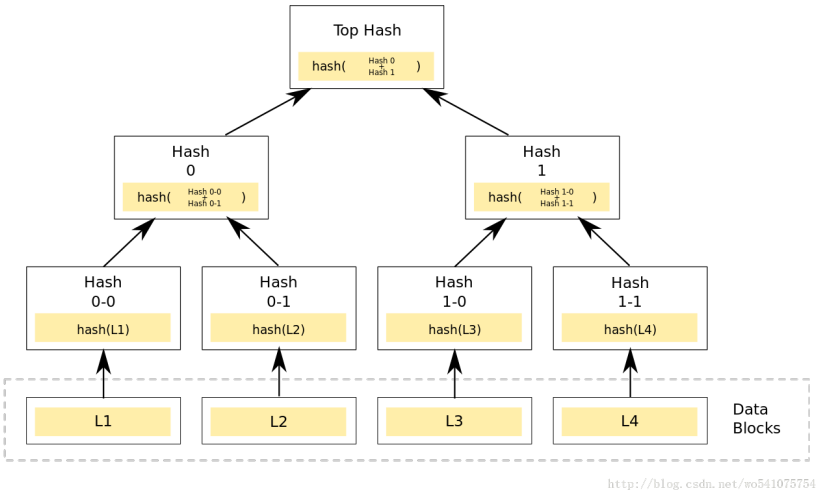

图 1 Merkle 树

由上图可以看出，Merkle 树是存储 hash 值的一棵树，可以看做 Hash List 的泛化。Merkle 树的叶子是数据块 (例如，文件或者文件的集合) 的 hash 值。非叶节点是其对应子节点串联字符串的 hash。

在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就合并，得到了一个”子哈希“。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做 Merkle Root。

而 RFC6962 表示：将单独多出来的叶子节点作为新计算的哈希值进入上一层

2.2 SM3 构建 Merkle 树具体代码实现

首先是数据处理部分：

`Bytes concat_bytes(const Bytes& a, const Bytes& b)` 将两个字节数组

拼接成一个新的字节数组。首先复制第一个数组，然后将第二个数组的内容插入到尾部。

`string bytes_to_hex(const Bytes& b)` 将字节数组转换成十六进制字符串。利用字符串流，格式化每个字节为两位十六进制数字，不足补零。

```
Bytes concat_bytes(const Bytes& a, const Bytes& b) {
    Bytes res = a;
    res.insert(res.end(), b.begin(), b.end());
    return res;
}
string bytes_to_hex(const Bytes& b) {
    stringstream ss;
    ss << hex << setfill('0');
    for (auto x : b) ss << setw(2) << (int)x;
    return ss.str();
}
```

接着是处理叶子节点，`Bytes hash_leaf(const string& data)` 对叶子节点的数据做哈希。根据 RFC6962，叶子节点哈希前要加前缀字节 `0x00`，再对拼接结果调用 SM3 哈希，得到叶子哈希。

```
Bytes hash_leaf(const string& data) {
    string prefixed = "\x00" + data;
    return sm3_hash(prefixed);
}
```

然后是处理非叶子节点:`Bytes hash_node(const Bytes& left, const Bytes& right)` 对非叶子节点的左右子节点哈希做哈希。根据 RFC6962，先加前缀字节 `0x01`，然后拼接左右子节点哈希，调用 SM3 哈希得到父节点哈希。

```
Bytes hash_node(const Bytes& left, const Bytes& right) {
    Bytes prefix = {0x01};
    Bytes combined = prefix;
    combined.insert(combined.end(), left.begin(), left.end());
    combined.insert(combined.end(), right.begin(), right.end());
    return sm3_hash(string(combined.begin(), combined.end()));
}
```

最后将上述函数进行总和，vector<vector<Bytes>> build_merkle_tree(const vector<string>& leaves_data) 函数构建 Merkle 树。首先对所有叶子数据调用 hash_leaf 计算叶子哈希，构成树的第一层。然后逐层将相邻的两个哈希用 hash_node 合并成父哈希，如果遇到单数节点则直接提上去，直到只剩一个根哈希。返回包含每层哈希的二维数组。

```
vector<vector<Bytes>> build_merkle_tree(const vector<string>& leaves_data) {
    vector<Bytes> level;
    level.reserve(leaves_data.size());
    for (auto& d : leaves_data)
        level.push_back(hash_leaf(d));

    vector<vector<Bytes>> tree;
    tree.push_back(level);

    while (level.size() > 1) {
        vector<Bytes> next_level;
        int n = (int)level.size();
        next_level.reserve((n + 1) / 2);
        for (int i = 0; i < n; i += 2) {
            if (i + 1 == n) {
                next_level.push_back(level[i]);
            } else {
                next_level.push_back(hash_node(level[i], level[i + 1]));
            }
        }
        tree.push_back(next_level);
        level = move(next_level);
    }
    return tree;
}
```

2.3 存在性与不存在性证明

存在性证明：

如果需要知道某个叶子是否存在于 Merkle 树中，只需要找出与该叶子节点两两合并路径上的各个节点，并根据这条路径不断合并得到根节点，判断此根节点是否与已知的根节点相同，若相同则完成了存在性证明。

不存在性证明：若是在存在性证明的过程中发现不存在就可以证明不存在。

我们使用两个函数来实现：

`vector<Bytes> get_inclusion_proof(const vector<vector<Bytes>>& tree, size_t leaf_index)` 函数根据叶子索引获取该叶子到根路径的存在性证明。证明由每层的兄弟节点哈希组成。遍历每层，根据叶子索引定位兄弟节点的哈希加入证明列表。若兄弟不存在，则用空的字节数组占位。

```
vector<Bytes> get_inclusion_proof(const vector<vector<Bytes>>& tree, size_t
    leaf_index) {
    vector<Bytes> proof;
    size_t index = leaf_index;
    for (size_t level = 0; level + 1 < tree.size(); ++level) {
        size_t sibling_index = (index % 2 == 0) ? index + 1 : index - 1;
        if (sibling_index < tree[level].size())
            proof.push_back(tree[level][sibling_index]);
        else
            proof.push_back({});
        index /= 2;
    }
    return proof;
}
```

`bool verify_inclusion_proof(const string& leaf_data, const vector<Bytes>& proof, const Bytes& root, size_t leaf_index)` 函数验证存在性证明。先计算叶子的哈希，再利用证明路径中兄弟节点哈希递归计算父节点哈希，顺序根据叶子索引奇偶决定合并顺序。最后计算得到的哈希与根哈希比较，判断是否匹配。

```
bool verify_inclusion_proof(const string& leaf_data, const vector<Bytes>&
    proof, const Bytes& root, size_t leaf_index) {
    Bytes hash = hash_leaf(leaf_data);
    size_t idx = leaf_index;
    for (auto& sibling_hash : proof) {
        if (sibling_hash.empty()) {
        } else {
            if (idx % 2 == 0)
                hash = hash_node(hash, sibling_hash);
            else
                hash = hash_node(sibling_hash, hash);
```

```
        }
        idx /= 2;
    }
    return hash == root;
}
```

3 实现结果

我们在 main 函数中使用了 10000 个字符串数据, 分别是 leaf_0 到 leaf_99999, 运行代码测试得到结果如下:

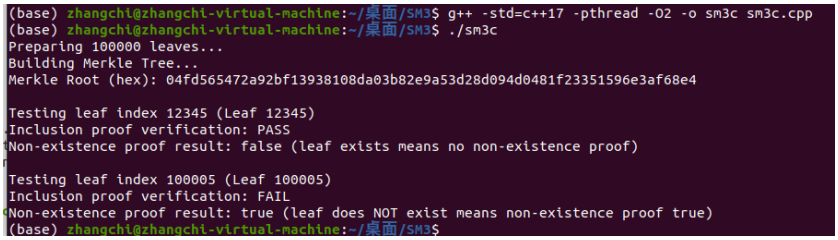

图 2 实验结果

可以看到根节点正常输出，而验证元素是否是叶子结点也正确。