



Merkle树介绍

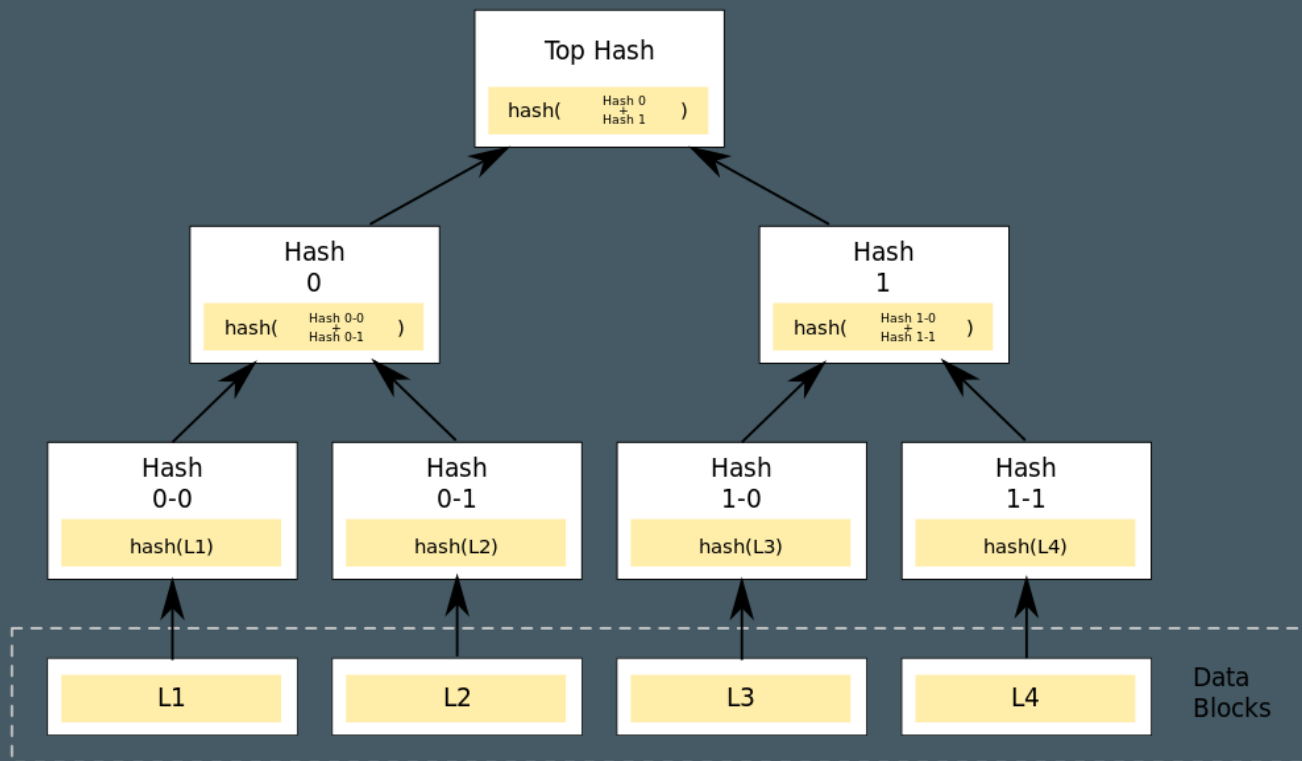
孙炜



- 背景知识
- **Merkle**树介绍
- **Bitcoin**中**Merkle**的应用



定义：Merkle树是一种二插树，它是一种用作快速归纳和校验大规模数据完整性的数据结构。





背景知识

- **Secure hash Function**
 - **Preimage resistant**
 - **Second preimage resistance**
 - **Collision resistant**
- **Lamport One-Time Signature**
- **Winternitz One-Time Signature**



Secure hash Function

- Preimage resistant (单向性)

对于一个哈希函数，很难找到数 m 使对给定的 h （任意给出），满足 $h = H(m)$

- Second preimage resistance (弱抗碰撞性)

对于一个哈希函数，很难找到给定的 m_2 对给定的 m_1 ，满足 $H(m_1) = H(m_2)$

- Collision resistant(强抗碰撞性)

对于一个哈希函数，很难找到数 m_1 和 m_2 ，满足 $H(m_1) = H(m_2)$



Secure hash Function

首先定义 $H\{0,1\}^* \rightarrow \{0,1\}^s$

- 突破preimage和 second preimage的复杂度
随机选择m，直到 $h=H(m)$ ，则 $H(m)$ 可能有 2^s 种可能，对于每一次尝试的结果出现的概率都是相同的且等于 $1/2^s$ ，则平均上来讲需要 $2^s/2$ 尝试才可能找到m,则突破preimage和second preimage的复杂度为 $O(2^s/2)=O(2^s)$

- 突破Collision resistant

这个比突破preimage和second preimage复杂度上要简单，这个问题符合生日悖论，时间复杂度为 $O(\sqrt{2^s})$ ，由生日悖论得知复杂度大于 $O(2^{80})$ 时，才算保证抗强碰撞性，因此 $s=160$ ，即最后哈希函数的比特数要大于160



Secure hash Function

second preimage resistance和 collision resistant

有一种疑惑是collision resistant包含second preimage resistance这里进行证明：

我们只要找到一个哈希函数，这个哈希函数满足second preimage resistance但是不满足collision resistant即可。我们现在定义一个哈希函数 $H(X)$ 满足collision resistant，同时我们定义另外一个哈希函数为：

$$H'(X) = \begin{cases} 0^l & \text{if } X=0^l \text{ or } X=1^l \\ H(X) & \text{其他情况} \end{cases}$$

很明显上述的函数定义不满足collision resistant的定义。现在我们只要证明 $H'(X)$ 满足第二原像就可以了。在抗第二原像中，对于任意一组数据 $(X^*, H'(X^*))$ ，由于其选择是随机的，一致的因此我们可以得到 $P[X^*=0^l \vee X^*=1^l] \leq \text{negl}(n)$ ；如果 X^* 没有落在这两个特殊的值上面，则根据 H' 的定义得出很难找到一个 X' 使得 $H'(X^*)=H'(X')$ ，则 $P[H'(X^*)=H'(X') \wedge (X' \neq X^*) \wedge (X^* \neq 0^l \wedge X^* \neq 1^l)] \leq P[H(X^*)=H(X') \wedge (X' \neq X^*)]$



Lamport One-Time Signature

$$P[H'(X^*)=H'(X') \wedge (X' \neq X^*) \wedge (X^* \neq 0^l \wedge X^* \neq 1^l)] \leq P[H(X^*)=H(X') \wedge (X' \neq X^*)] \leq \text{negl}(n)$$

$$P[H'(X^*)=H'(X') \wedge (X' \neq X^*)] \leq P[H'(X^*)=H'(X') \wedge (X' \neq X^*) \wedge (X^* \neq 0^l \wedge X^* \neq 1^l)] + P[X^*=0^l \vee X^*=1^l] = \text{negl}(n) + \text{negl}(n)$$

则函数 H' 为这样一个函数，它满足second preimage resistance，但是不满足collision resistant



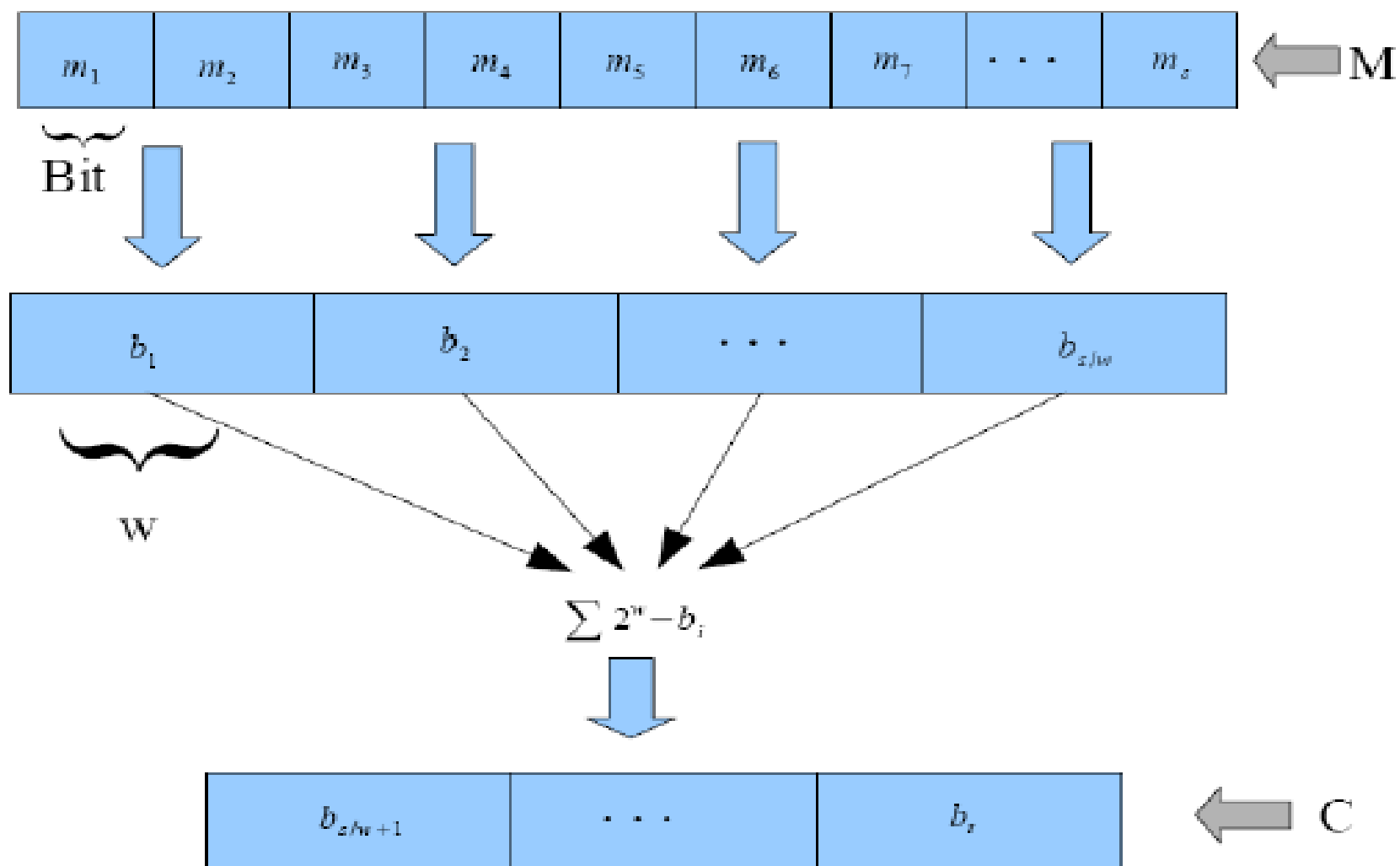
Lamport One-Time Signature

缺陷是公钥和私钥的空间存储空间太大，因为为了满足collision resistant，则复杂度需要满足 $O(2^k)$ ，哈希函数bits为160bits，则公钥和私钥一共需要 $k \times 160 \times 2 = 320k$ bits，一般一个消息在计算哈希之前会被编码，所以一般k也为160，则最后的空间占用为 $160 \times 160 \times 2 = 64000$ bytes。

Winternitz One-Time Signature减少了存储上的开销。



Winternitz One-Time Signature





Winternitz One-Time Signature

- 计算key

选择一个数 w 作为参数，计算 $t = \lceil s/w \rceil + \lceil \lceil \log_2 \lceil s/w \rceil + 1 + w \rceil / w \rceil$ ，其中一个比较大的 w 会降低存储的空间，但是会增加计算的时间。选 t 个随机数 X_1, X_2, \dots, X_t ，则 $(X_1 || X_2 || X_3 \dots X_t)$ 作为私钥。公钥 $Y_i = H^{2^w - 1}(X_i)$

- 签名消息

对于给定消息 $M = m_1, m_2, m_3, \dots, m_k$ ， $m_i \in \{0, 1\}$ ，将 M 分成 $\lceil s/w \rceil$ 块分别记为 $b_1, b_2, b_3, \dots, b_{\lceil s/w \rceil}$ 。现在把 b_i 看成是整形数编码，并计算它checksum C ， $C = \sum_{i=1}^{\lceil s/w \rceil} 2^w - b_i$ ，然后将算出来的 C 在进行分成 $\lceil \lceil \log_2 \lceil s/w \rceil + 1 + w \rceil / w \rceil$ 块，记为 $b_{\lceil s/w \rceil + 1}, \dots, b_t$ ，我们将 b_i 看成整数，则 $\text{sig}_i = H^{b_i}(X_i)$ ，则 $\text{sig} = (\text{sig}_1 || \text{sig}_2 \dots \text{sig}_t)$ 为消息 M 的签名。

- 签名验证

根据消息算出 b_1, \dots, b_t ，计算出 $\text{sig}'_i = H^{2^w - b_i}(\text{sig}_i)$ ，最后如果 $H(\text{sig}'_1 || \dots \text{sig}'_t)$ 等于 $H(Y_1 || \dots Y_t)$



Winternitz One-Time Signature

w在这里起到的作用

- 计算key

对于 $Y_i = H^{2^w-1}(X_i)$ ，其计算时间的等于 $(2^w-1)*\text{hash}_{\text{time}} + \text{random}_{\text{time}}$ 由于 $t \approx s/w$ ，则总的花费时间约为 $s/w * ((2^w-1)*\text{hash}_{\text{time}} + \text{random}_{\text{time}})$ ，其复杂度为

$O(2^w)*\text{hash}_{\text{time}} + O(1/w)*\text{random}_{\text{time}}$ ，所以从上面可以看出计算key的时间依赖于**w**的大小

- 签名消息

一个 sig_i 的计算 $\text{sig}_i = H^{b_i}(X_i)$ ($b_i \leq 2^w-1$)，平均哈希计算的数学期望为

$\sum_{j=1}^{2^w-1} 2^j * p(j)$ ，由于把 b_i 作为整数对待，它的第j位值为1的分布式离散的，且概率相同，均为 $1/w$

则平均的哈希计算的数学期望为 $\sum_{j=1}^{2^w-1} 2^j * p(j) = \sum_{j=1}^{2^w-1} 2^j / w = \frac{2^w-2}{w}$

则签名时间 $s/w * (2^w-2)/w * \text{hash}_{\text{time}} = O(2^w)$

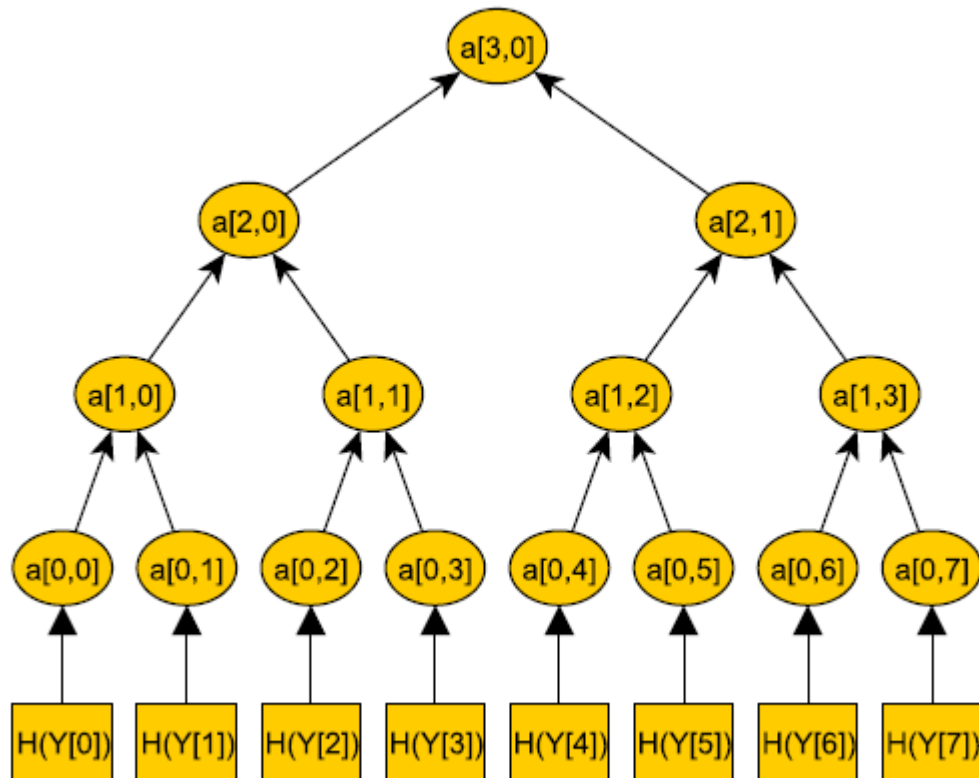
- 签名验证

同上，因为最后算的 b_i 的宽度基本一致



Merkle树

上述的两种one-time signature的问题在于公钥的管理，公钥多（一条消息一个），占用空间大，而Merkle树则提供了一种更加实用的管理方法，它用一个公钥签名多条消息。

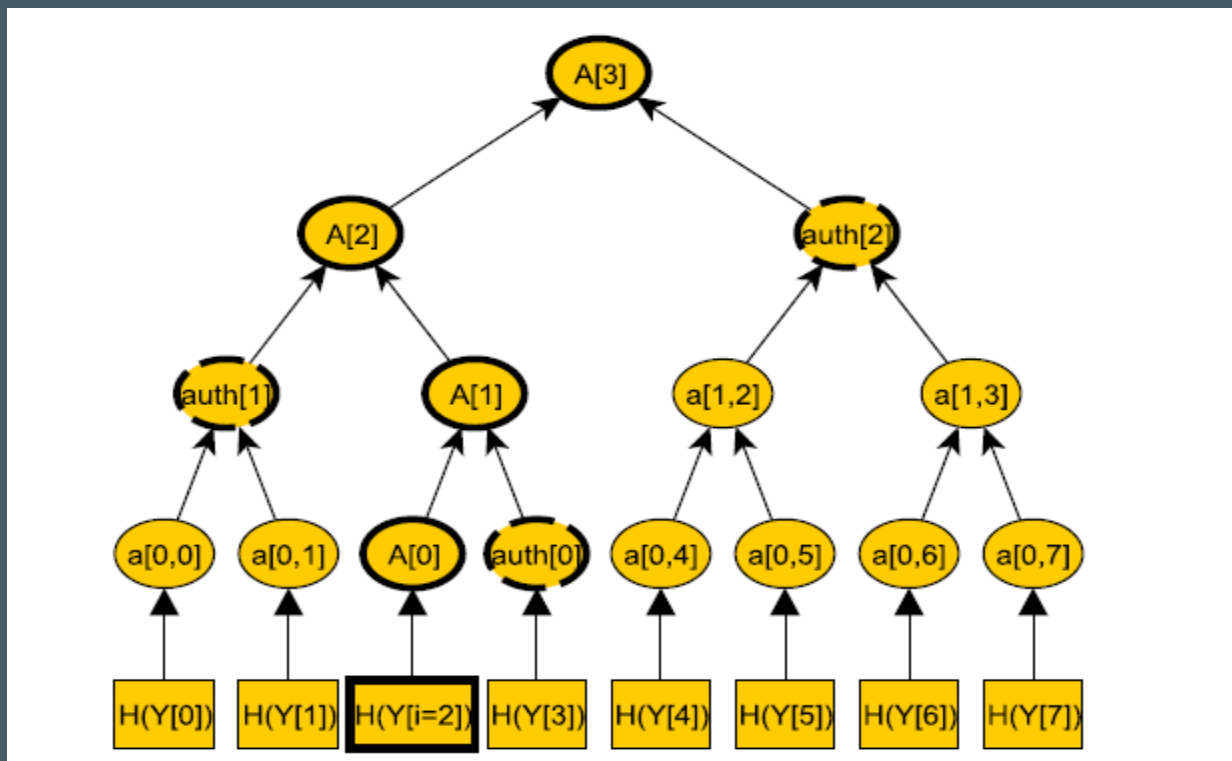




Merkle树

签名过程

对消息M进行Merkle树签名，首先需要用其他one-time signature方法对M进行签名，得到 sig' 为了计算出从节点 A_i 到根节点的路径，需要知道它的兄弟节点，称之为 auth_i ，则对于消息M，其Merkle的签名一条签名路径为 $(\text{sig}' || \text{auth}_1 || \text{auth}_2 \dots \text{auth}_{n-1})$

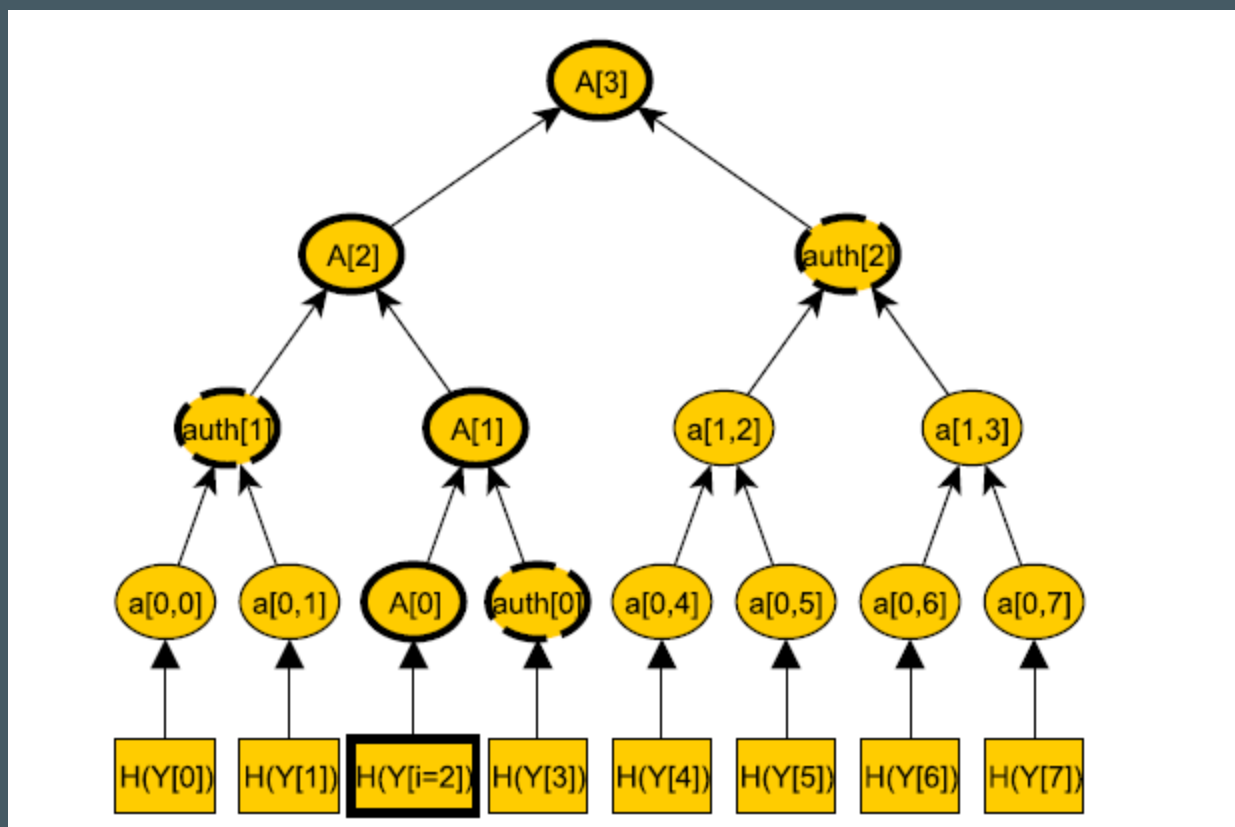




Merkle树

验证过程

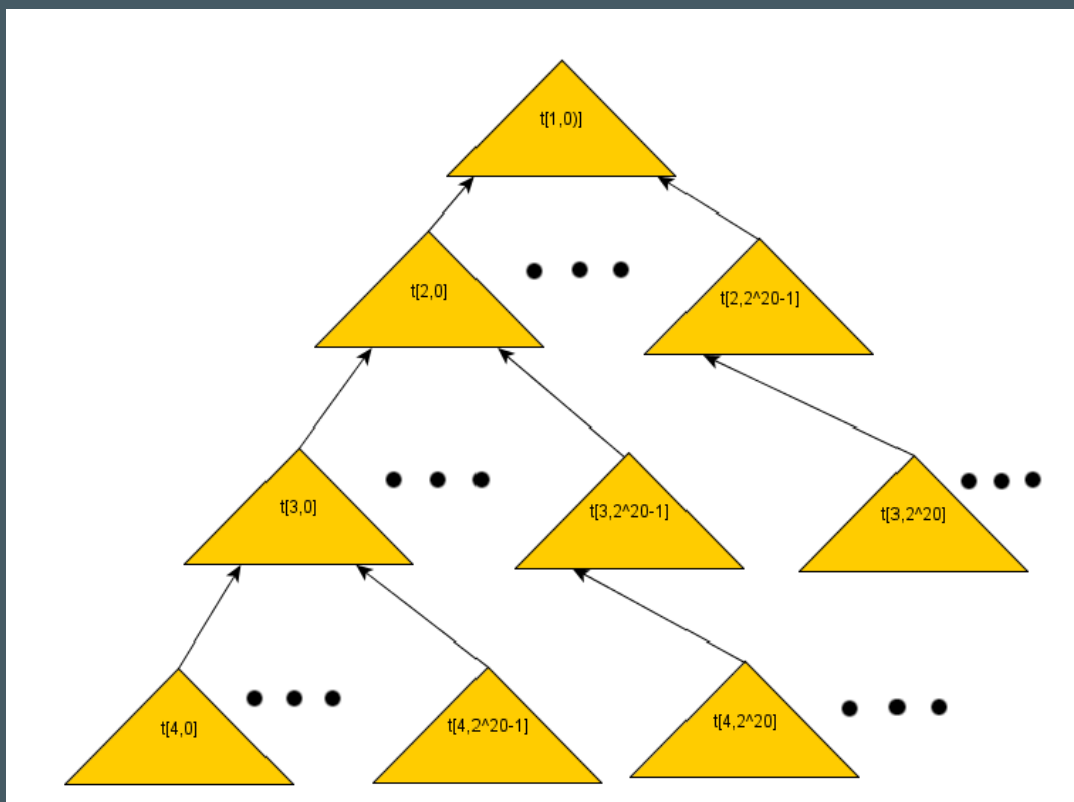
首先对M进行验证，然后 $A_0 = H(Y_{i0})$ ，然后再逐层的去计算树上的节点，最后得到的根节点进行比较，如果160bits哈希函数应用在这里，则签名的大小为 $\text{sig}' + n \times 160$ 。





Merkle树

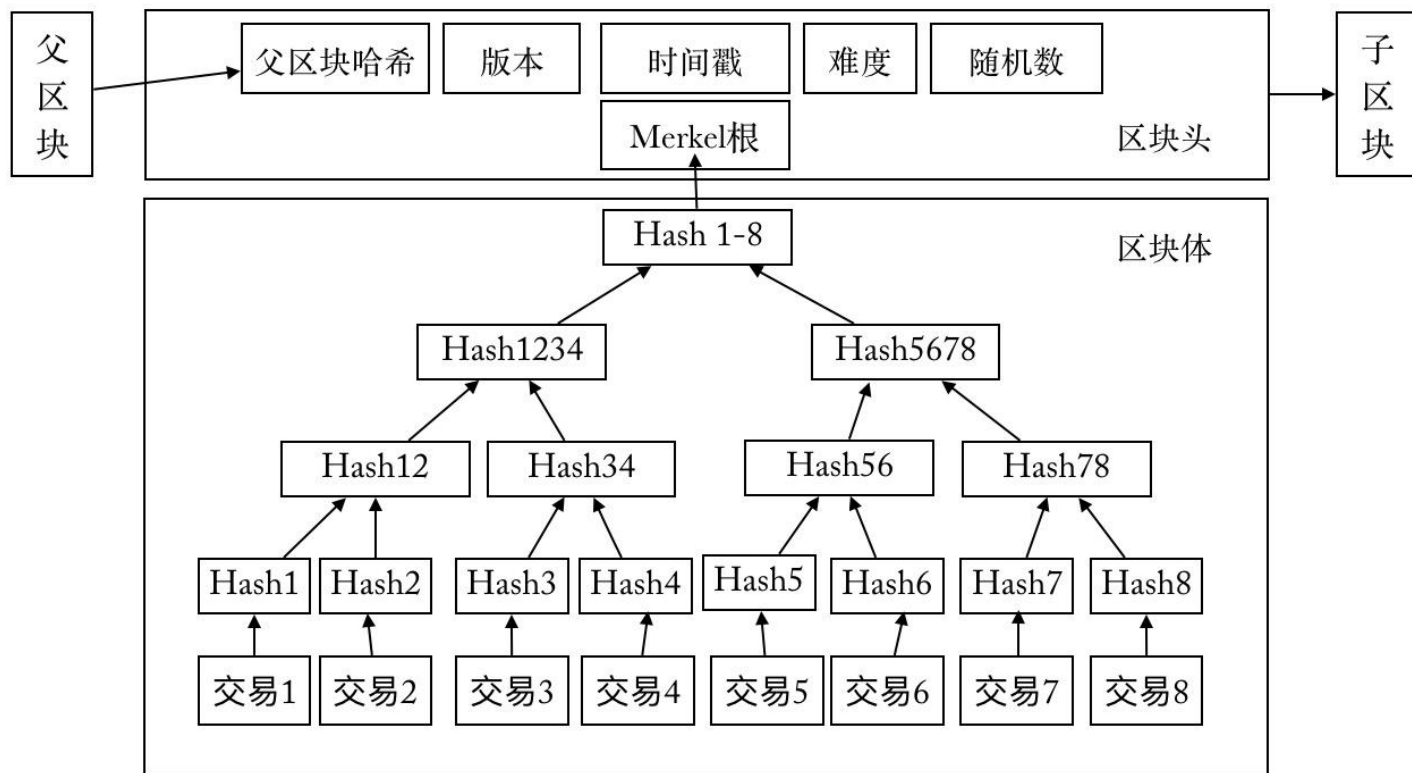
Merkle树仍然是一种受限的签名树，例如如果签名 2^{80} 个叶子节点几乎是不可能完成的，这里可以构建子树的形式来解决。这样减少每个树的规模，最后如图我们构建4棵 2^{20} 的Merkle树





Bitcoin 中的Merkle树应用

Merkle树在bitcoin中对交易进行验证，其被包含在区块头中，如下图所示





Bitcoin 中的Merkle树应用

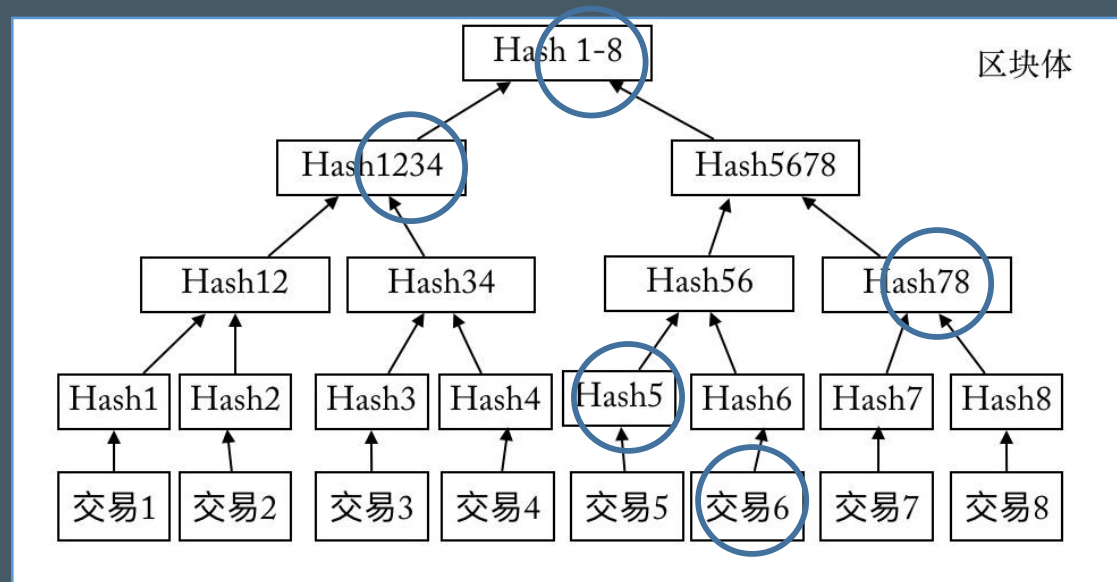
验证

以验证文件为例说明，

- 获得根哈希
- 获得每块的哈希列表
- 通过根哈希来验证哈希列表的有效性
- 下载通过哈希列表来验证每个下载的数据是否有效

Bitcoin中spv节点交易验证（过程类似）

- spv节点向邻节点索要（Merkleblock消息）,哈希值到根节点的哈希序列（5,78,1234,1-8）来验证交易的存在和正确性





Bitcoin 中的Merkle树应用

代码主要在merkle.h merkle.cpp merkleblock.h和merkleblock.cpp中

核心函数

static void MerkleComputation(const std::vector<uint256>& leaves, uint256* proot,
bool* pmutated, uint32_t branchpos, std::vector<uint256>* pbranch)

```
while (count < leaves.size()) {
    uint256 h = leaves[count];
    bool matchh = count == branchpos;
    count++;
    int level;
    // For each of the lower bits in count that are 0, do 1 step. Each
    // corresponds to an inner value that existed before processing the
    // current leaf, and each needs a hash to combine it.
    for (level = 0; !(count & ((uint32_t)1 << level)); level++) {
        if (pbranch) {
            if (matchh) {
                pbranch->push_back(inner[level]);
            } else if (matchlevel == level) {
                pbranch->push_back(h);
                matchh = true;
            }
        }
        mutated |= (inner[level] == h);
        CHash256().Write(inner[level].begin(), 32).Write(h.begin(), 32).Finalize(h.begin());
    }
    // Store the resulting hash at inner position level.
    inner[level] = h;
    if (matchh) {
        matchlevel = level;
    }
}
```



Bitcoin 中的Merkle树应用

```
// Check that the header is valid (particularly PoW). This is mostly
// redundant with the call in AcceptBlockHeader.
if (!CheckBlockHeader(block, state, consensusParams, fCheckPOW))
    return false;

// Check the merkle root.
if (fCheckMerkleRoot) {
    bool mutated;
    uint256 hashMerkleRoot2 = BlockMerkleRoot(block, &mutated);
    if (block.hashMerkleRoot != hashMerkleRoot2)
        return state.DoS(100, false, REJECT_INVALID, "bad-txnmrklroot", true, "hashMerkleRoot mismatch");
}
```



Q&A