第3篇

基于 Hash 函数的数字签名方案

Johannes Buchmann[1]　Erik Dahmen[1]
Michael Szydlo[2]

[1] Technische Universität Darmstadt
[2] Akamal Technologies

数字签名已经成为保护网络和其他 IT 基础设施安全的关键技术。数字签名提供数据的真实性、完整性和不可抵赖性。数字签名已广泛地应用于身份识别和认证协议。因而，存在安全的数字签名算法对保护 IT 安全是至关重要的。

现今实际应用的数字签名算法有 RSA[31]、DSA[11] 和 ECDSA[15]。由于它们的安全性是基于大合数分解和计算离散对数的困难性，所以不是量子免疫的。

本章介绍的基于 Hash 函数的数字签名方案提供了一个十分有趣的选择。像任何其他的数字签名方案一样，基于 Hash 函数的数字签名方案使用了一个加密的 Hash 函数。它们的安全性基于 Hash 函数的抗强碰撞性。事实上，我们将要提出的基于 Hash 函数的数字签名方案是安全的，当且仅当相关的 Hash 函数是抗强碰撞的。对于存在用一个私钥签署多个文件的数字签名方案，存在抗强碰撞函数被看作是最起码的要求。签名方案将文件（任意比特串长）映射为数字签名（固定比特串长）。这说明数字签名算法事实上是 Hash 函数。那些 Hash 函数必须是抗强碰撞的：如果可以构建具有相同签名的两个文件，签名不再被认为是安全的。这种说法表明只要存在可用一个密钥签署多个文件的任何数字签名方案，就存在基于 Hash 函数的数字签名方案。因此，基于 Hash 函数的数字签名方案是最重要的抗量子签名候选方案。尽管没有抗量子计算机的证明，但是它们的安全性需要是最小的。此外，每一个新的加密 Hash 函数产生一个新的基于 Hash 函数的签名方案。所以安全的签名方案构造在数论或代数理论中是独立的困难算法问题。这从对称密码中构造就足够了。这导致了基于 Hash 函数的数字签名方案另外的巨大优势。相关的 Hash 函数可以基于可用的硬件和软件资源进行选择。例如，如果该签名方案要在一个芯片上实现，从而减少代码尺寸和优化其运行时间，可以选择使用在芯片上已经实现的基于 AES 的 Hash 函数。

基于 Hash 函数的数字签名方案是由 Ralph Merkle 发明的[23]。Merkle 的工作从一次性签名方案开始，尤其是借鉴 Lamport 和 Difie 的工作[18]。一次性签名是更根本的。构造一个安全的一次性签名方案仅需要一个单向函数。如 Rompel 的工作所示[28]，单向函数对安全的数字签名是必要且充分的。所以一次性签名方案其实是数字签名方案的最基本类型。然而，它们有严重的缺陷。由一个秘密签名密钥和一个公开验证密钥组成的一个密钥对只能用于签名和验证单个文件。对大多数的实际应用是不够的。Merkle 的思想是利用 Hash 树将多个一次性验证密钥（Hash 树的叶子）的有效性降低到一个公钥（Hash 树的根）的有效性。Merkle 最初的构造并不是充分有效的，特别是与 RSA 签名方案比较。然而与此同时，已发现许多改善的方案。现今，基于 Hash 函数的签名是代替 RSA 和椭圆曲线最有前途的签名方案。

1 基于 Hash 函数的一次性签名方案

本节阐释了安全性仅基于加密 Hash 函数的抗强碰撞性的一个签名方案。它们是抗量子计算的优秀候选方案。

1.1 Lamport-Diffie 一次性签名方案

在文献[18]中提出了 Lamport-Diffie 一次性签名方案（LD-OTS）。设 LD-OTS 的安全性参数 n 是正整数。LD-OTS 使用一个单向函数 $f:\{0,1\}^n\to\{0,1\}^n$ 和一个加密的 Hash 函数 $g:\{0,1\}^*\to\{0,1\}^n$。

LD-OTS 密钥对生成。随机均匀选择 $2n$ 个长为 n 比特位字符串组成 LD-OTS 的签名密钥 X，

$$X=(x_{n-1}[0],x_{n-1}[1],\cdots,x_1[0],x_1[1],x_0[0],x_0[1])\in_R\{0,1\}^{(n,2n)} \tag{1}$$

LD-OTS 的验证密钥 Y 是

$$Y=(y_{n-1}[0],y_{n-1}[1],\cdots,y_1[0],y_1[1],y_0[0],y_0[1])\in\{0,1\}^{(n,2n)} \tag{2}$$

其中

$$y_i[j]=f(x_i[j]),\quad 0\leqslant i\leqslant n-1,\ j=0,1 \tag{3}$$

所以 LD-OTS 密钥生成需要 f 的 $2n$ 次赋值。签名和验证密钥是 $2n$ 个长为 n 的比特字符串。

LD-OTS 签名生成。用具有如方程(1)中签名密钥 X 的 LD-OTS 对文件 $M\in\{0,1\}^*$ 签名。设 $g(M)=d=(d_{n-1},\cdots,d_0)$ 是 M 的消息摘要。那么 LD-OTS 的签名是

$$\sigma=(x_{n-1}[d_{n-1}],\cdots,x_1[d_1],x_0[d_0])\in\{0,1\}^{(n,n)} \tag{4}$$

该签名是 n 比特字符串序列，每个字符串长度为 n。挑选它们作为消息摘要 d 的函数。如果在 d 中第 i 个比特是 0，那么签名中第 i 个比特字符串是 $x_i[0]$，反之为 $x_i[1]$。签名不需要 f 的赋值。签名的长度是 n^2。

LD-OTS 验证。验证式(4)中 M 的一个签名 $\sigma=(\sigma_{n-1},\cdots,\sigma_0)$，验证者计算消息摘要

$d=(d_{n-1}, \cdots, d_0)$。然后检查是否

$$(f(\sigma_{n-1}), \cdots, f(\sigma_0)) = (y_{n-1}[d_{n-1}], \cdots, y_0[d_0] \tag{5}$$

签名验证需要 f 的 n 次赋值。

例 1　设 $n=3, f:\{0,1\}^3 \to \{0,1\}^3$, $x \mapsto x+1 \bmod 8$, 且设 $d=(1,0,1)$是消息 M 的哈希值。我们选择签名密钥

$$\boldsymbol{X} = (x_2[0], x_2[1], x_1[0], x_1[1], x_0[0], x_0[1])$$

$$= \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \in \{0,1\}^{(3,6)}$$

且计算相应的验证密钥

$$\boldsymbol{Y} = (y_2[0], y_2[1], y_1[0], y_1[1], y_0[0], y_0[1])$$

$$= \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \in \{0,1\}^{(3,6)}$$

$d=(1,0,1)$的签名是

$$\boldsymbol{\sigma} = (\sigma_2, \sigma_1, \sigma_0) = (x_2[1], x_1[0], x_0[1]) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \in \{0,1\}^{(3,3)}$$

例 2　我们给出一个例子说明为什么 LD-OTS 的签名密钥仅使用一次。设 $n=4$,假设签名者用相同的签名密钥签署两个摘要是 $d_1=(1,0,1,1)$和 $d_2=(1,1,1,0)$的消息。那些摘要的签名分别是 $\sigma_1=(x_3[1], x_2[0], x_1[1], x_0[1])$和 $\sigma_2=(x_3[1], x_2[1], x_1[1], x_0[0])$。那么攻击者从签名密钥知道 $x_3[1]$, $x_2[0]$, $x_2[1]$, $x_1[1]$, $x_0[0]$, $x_0[1]$。她能利用这个信息对具有摘要 $d_3=(1,0,1,0)$和 $d_4(1,1,1,1)$的消息产生有效的签名。这个例子可以一般化到任意的安全参数 n。因而,攻击者也仅能对特定的摘要生成有效的签名。只要用于计算消息摘要的 Hash 函数是加密安全的,攻击者就不能找到合适的消息。

1.2　Winternitz 一次性签名方案

生成 LD-OTS 的密钥和签名是十分高效的,但是签名的尺寸是很长的。本节描述的 Winternitz OTS(W-OTS)生成明显的短签名。该思想是利用一次性签名密钥的一个字符串在消息摘要中同时签署多个比特位。这个建议首先出现在 Merkle 的论文中[23],该论文提到该方法是在 1979 年受 Winternitz 建议的影响将 Merkle OTS 一般化产生的。然而尽作者所知,文献[10]首次描述了 Winternitz OTS 的全部细节。正如 LD-OTS 一样,W-OTS 使用了单向函数 $f:\{0,1\}^n \to \{0,1\}^n$ 和一个加密的 Hash 函数 $g:\{0,1\}^* \to \{0,1\}^n$。

W-OTS 密钥对生成。挑选一个 Winternitz 参数 $\omega \geqslant 2$,它是同时签署多个比特位的

数目。那么

$$t_1 = \left\lceil \frac{n}{\omega} \right\rceil, \quad t_2 = \left\lfloor \frac{\lfloor \log_2 t_1 \rfloor + 1 + \omega}{\omega} \right\rfloor, \quad t = t_1 + t_2 \tag{6}$$

是确定的。签名密钥 X 是

$$X = (x_{t-1}, \cdots, x_1, x_0) \in_R \{0, 1\}^{(n, t)} \tag{7}$$

其中比特字符串 x_t 均匀的随机选择。

验证密钥 Y 可通过计算 f 作用在每个签名密钥比特字符串 $2^\omega - 1$ 次来得到。所以有

$$Y = (y_{t-1}, \cdots, y_1, y_0) \in \{0, 1\}^{(n, t)} \tag{8}$$

其中

$$y_i = f^{2^w - 1}(x_i), \quad 0 \leqslant i \leqslant t - 1 \tag{9}$$

分别地，密钥生成需要 f 的 $t(2^\omega - 1)$ 次赋值计算，且签名和验证密钥的长度分别是 $t \cdot n$ 比特。

W-OTS 签名生成。签署具有消息摘要 $g(M) = d = (d_{n-1}, \cdots, d_0)$ 的消息 M。首先，将最少数量的零置于 d 之前使得 d 的长度能被 ω 整除。扩展的字符串 d 被分成 t_1 个长度是 ω 的字符串 $b_{t-1}, \cdots b_{t-t_1}$。那么

$$d = b_{t-1} \| \cdots \| b_{t-t_1} \tag{10}$$

其中 $\|$ 表示首尾相连。然后，比特字符串 b_i 由集合 $\{0,1,2,\cdots,2^\omega - 1\}$ 中的整数确定且计算校验和

$$c = \sum_{i=t-t_1}^{t-1} (2^\omega - b_i) \tag{11}$$

因为 $c \leqslant t_1 2^\omega$，c 的二进制表示的长度小于

$$\lfloor \log_2 t_1 2^w \rfloor + 1 = \lfloor \log_2 t_1 \rfloor + \omega + 1 \tag{12}$$

最少的零置于这个二进制表示之前使得扩展的字符串的长度被 ω 整除。扩展的字符串被分成 t_2 个长度为 ω 的块 $b_{t_2-1}, \cdots, b_0$。那么

$$c = b_{t_2-1} \| \cdots \| b_0$$

最后计算 M 的签名为

$$\sigma = (f^{b_{t-1}}(x_{t-1}), \cdots, f^{b_1}(x_1), f^{b_0}(x_0)) \tag{13}$$

在最坏的情况下，签名生成需要 f 的 $t(2^\omega - 1)$ 次赋值。W-OTS 的签名大小是 $t \cdot n$。

W-OTS 验证。对于签名 $\sigma = (\sigma_{t-1}, \cdots, \sigma_0)$ 的验证，计算比特字符串 $b_{t-1}, \cdots, b_0$ 作为 1.1 节的解释。如果

$$(f^{2^\omega - 1 - b_{t-1}}(\sigma_{n-1}), \cdots, f^{2^\omega - 1 - b_0}(\sigma_0)) = (y_{n-1}, \cdots, y_0) \tag{14}$$

那么我们进行检查。如果签名是有效的，那么 $\sigma_i = f^{b_i}(x_i)$，且

$$f^{2^\omega - 1 - b_i}(\sigma_i) = f^{2^\omega - 1}(x_i) = y_i \tag{15}$$

对 $i = t-1, \cdots, 0$ 成立。在最坏的情况下，签名验证需要 f 的 $t(2^\omega - 1)$ 次赋值。

例 3 设 $n=3, \omega=2, f: \{0, 1\}^3 \to \{0, 1\}^3$，$x \to x+1 \bmod 8$，且 $d=(1, 0, 0)$。得到

$t_1=2$，$t_2=2$ 且 $t=4$。我们选择签名密钥

$$\boldsymbol{X}=(x_3,\ x_2,\ x_1,\ x_0)=\begin{bmatrix}1 & 0 & 0 & 1\\ 1 & 0 & 1 & 1\\ 1 & 0 & 1 & 0\end{bmatrix}\in\{0,\ 1\}^{(3,\ 4)}$$

且 f 三次作用在 $\boldsymbol{X}$ 中的比特字符串上，计算得到验证密钥：

$$\boldsymbol{Y}=(y_3,\ y_2,\ y_1,\ y_0)=\begin{bmatrix}0 & 0 & 1 & 0\\ 1 & 1 & 1 & 0\\ 0 & 1 & 0 & 1\end{bmatrix}\in\{0,\ 1\}^{(3,\ 4)}$$

在 d 之前置一个零且将扩展的字符串分成长度为 2 的块产生 $d=01\parallel 00$。验证和 $c=(4-1)+(4-0)=7$。在 c 的二进制表示前置一个零且将扩展字符串分成长度为 2 的块产生 $c=01\parallel 11$。签名是

$$\boldsymbol{\sigma}=(\sigma_3,\ \sigma_2,\ \sigma_1,\ \sigma_0)=(f(x_3),\ x_2,\ f(x_1),\ f^3(x_0))=\begin{bmatrix}0 & 0 & 1 & 1\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 1\end{bmatrix}\in\{0,\ 1\}^{(3,\ 4)}$$

验证签名通过计算

$$(f^2(\sigma_3),\ f^3(\sigma_2),\ f^2(\sigma_1),\ \sigma_0)=\begin{bmatrix}0 & 0 & 1 & 0\\ 1 & 1 & 1 & 0\\ 0 & 1 & 0 & 1\end{bmatrix}\in\{0,\ 1\}^{(3,\ 4)}$$

且同验证密钥 $\boldsymbol{Y}$ 比较。

例 4　我们给了一个例子阐述为什么 W-OTS 的签名密钥只使用一次。设 $\omega=2$，假设签名者使用相同签名密钥签署具有摘要 $d_1=(1,\ 0,\ 0)$ 和 $d_2=(1,\ 1,\ 1)$ 的两个消息。那些摘要的签名分别是 $\sigma_1=(f(x_3),x_2,f(x_1),f^3(x_0))$ 和 $\sigma_2=(f(x_3),f^3(x_2),f(x_1),x_0)$。攻击者可以利用这个消息去计算具有摘要 $d_3=(1,\ 1,\ 0)$ 的消息签名是 $\sigma_3=(f(x_3),\ f^2(x_2),\ f(x_1),\ f(x_0))$。而这个例子可以推广到任意的安全参数 n。因而对特定的摘要，攻击者能产生有效的签名。只要用于计算消息摘要的 Hash 函数是密码安全的，他不能找到恰当的消息。

2　Merkle 树认证方案

在上一节最后介绍的一次性签名方案对大部分的实际情况是不够的，这是因为每个密钥对只能用于一次签名。在 1979 年 Ralph Merkle 提出了解决这一问题的一个方法[23]。他的思想是使用一个完全二叉 Hash 树减少一个任意的有效性到单个公钥的有效性，而不是一次性验证密钥固定的数量，即 Hash 树的根。

Markle 签名方案(MSS)使用任意加密的 Hash 函数和任意的一次性签名方案。为方便解释我们设 $g=\{0,\ 1\}^{*}\rightarrow\{0,\ 1\}^{n}$ 是一个加密的 Hash 函数。我们也假定一个一次性签名方案已经选定。

2.1 MSS 密钥对生成

签名者选择 $H\in N$，$H\geqslant 2$，那么生成的密钥对将能签署或验证 2^H 个文件。注意到能否用一个密钥对签署任意多个文件是签名方案的重要区别，譬如 RSA 和 ECDSA。然而，实际上这一数字也受限于签名产生的设备或者某些政策。签名者生成 2^H 个一次性密钥对(X_j, Y_j)，$0\leqslant j<2^H$。这里 X_j 是签名密钥而 Y_j 是验证密钥。它们都是比特字符串。Merkle 树的叶是摘要 $g(Y_j)$，$0\leqslant j<2^H$。Merkle 树的内节点根据以下的构造规则计算得到：一个父节点是它的左孩子和右孩子首尾相连的 Hash 值。MSS 的公钥是 Merkle 树的根。MSS 私钥是 2^H 个一次性签名密钥序列。更精确地，记 Merkle 树的节点是 $v_h\lceil j\rceil$，$0\leqslant j<2^{H-h}$，其中 $h\in\{0,\cdots,H\}$是节点的高度。那么

$$v_h[j]=g(v_{h-1}[2j]\parallel v_{h-1}[2j+1]),\quad 1\leqslant h\leqslant H,\ 0\leqslant j<2^{H-h} \tag{16}$$

图 1 所示为高度 $H=3$ 的 Merkle 树。

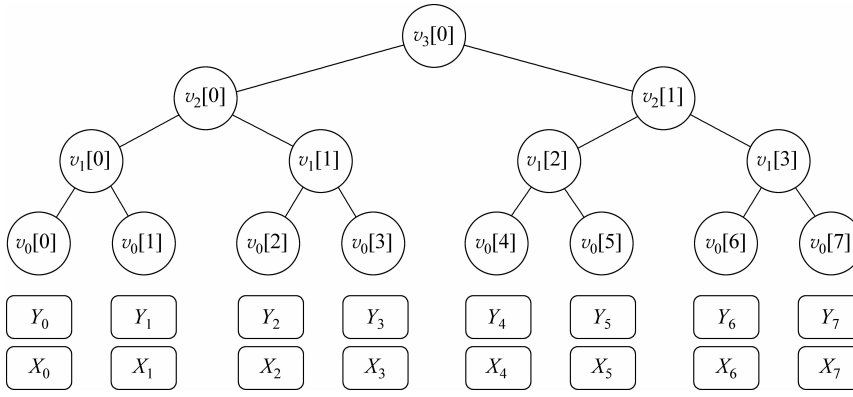

图 1　高度 $H=3$ 的 Merkle 树

MSS 密钥对的生成需要计算 2^H 个一次性密钥对和 Hash 函数的 $2^{H+1}-1$ 次赋值。

2.2 高效的根计算

为了计算 Merkle 树的根，不必储存所有的 Hash 树。作为替代，应用算法 1(树哈希算法)。该算法的基本观点是连续计算叶且尽可能计算它们的父节点。为了储存节点，树哈希算法使用一个具有通常的推入和弹出运算的栈 STACK 设备。Merkle 树的高度 H 是树哈希算法的输入。Merkle 树的根是输出，即 MSS 的公钥。算法 1 使用子程序 LEAFCALC(j)计算第 j 个叶。程序 LEAFCALC (j)计算第 j 个一次性密钥对和从第 j 个一次性验证密钥中计算第 j 个叶，如上所述。

算法 1　树哈希

输入：高度 $H\geqslant 2$；

输出：Merkle 树的根；

(1) for $j=0,\cdots,2^H-1$ do

① 计算第 j 个叶：$NODE_1 \leftarrow LEAFCALC(j)$。

② While $NODE_1$ 有相同的高度作为 STACK 顶部节点 do。

- 从栈中弹出顶部节点：$NODE_2 \leftarrow STACK.pop()$。
- 计算它们的父节点：$NODE_1 \leftarrow g(NODE_2 \parallel NODE_1)$。

③ 在栈上推入一个父节点：STACK.push($NODE_1$)。

(2) 设 R 是储存在栈上的单个节点：R←stack.pop()。

(3) Return R。

图 2 说明了树哈希算法计算 Merkle 树节点的顺序。在这个例子里，在栈上储存节点的最大数是 3。这发生在生成节点 11 且推入栈之后。通常，树哈希算法需要在栈中储存至多 H 个所谓的尾节点。为了计算高度为 H 的 Merkle 树的根，树哈希算法需要调用 2^H 个 LEAFCALC 子程序，和 2^H-1 次 Hash 函数的赋值。

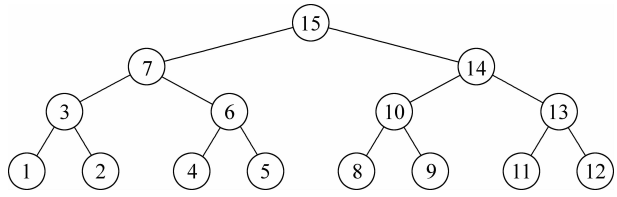

图 2　树哈希算法

2.3　MSS 签名生成

MSS 连续使用一次性签名密钥生成签名。为了签署消息 M，签名者首先计算 n 比特摘要 $d=g(M)$。然后他使用第 s 个一次性签名密钥 X_s，$s\in\{0,\cdots,2^H-1\}$ 生成摘要的一次性签名 σ_{OTS}。Merkle 签名将包含这个一次性签名和对应的一次性验证密钥 Y_s。为了给验证者证明 Y_s 的真实性，签名者包含了指标 s 以及验证密钥 Y_s 的认证路径，其中认证路径是 Merkle 树的节点序列 $A_s=(a_0,\cdots,a_{H-1})$。这个指标和认证路径允许验证者构建一条从叶 $g(Y_s)$ 到 Merkle 树根的路径。在认证路径里的节点 h 是从叶 $g(Y_s)$ 到 Merkle 树根的路径里高度为 h 的节点的兄弟：

$$a_h=\begin{cases}v_h\lceil s/2^h-1\rceil, & \text{如果}\lfloor s/2^h\rfloor\equiv 1 \bmod 2\\ v_h\lceil s/2^h+1\rceil, & \text{如果}\lfloor s/2^h\rfloor\equiv 0 \bmod 2\end{cases} \tag{17}$$

而 $h=0,\cdots,H-1$。图 3 指出 $s=3$ 的一个例子。所以第 s 个 Merkle 签名是

$$\sigma=(s,\sigma_{OTS},Y_s,(a_0,\cdots,a_{H-1})) \tag{18}$$

2.4　MSS 签名验证

前面提到的 Merkle 签名验证由两步构成。第 1 步，验证者使用一次性验证密钥 Y_s 通过各自的一次性签名方案的验证算法验证摘要 d 的一次性签名 σ_{OTS}。第 2 步，验证者通过构造从第 s 个叶 $g(Y_s)$ 到 Merkle 树根的路径 $(p_0,\cdots,p_H)$ 来验证一次性验证密钥 Y_s 的真伪。他使用指标 s，验证路径 $(a_0,\cdots,a_{H-1})$ 以及下面的结构。

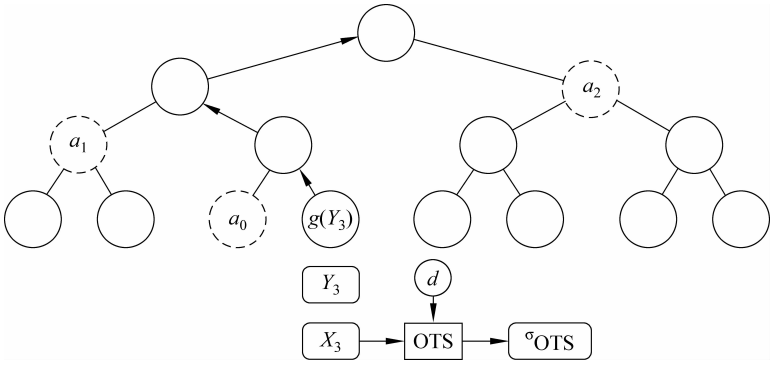

图 3　$s=3$ 的 Merkle 签名生成

注：虚节点表示叶 $g(Y_3)$的认证路径。箭头指示从叶 $g(Y_3)$到根的路径。

$$p_h = \begin{cases} g(a_{h-1} \parallel p_{h-1}), & \text{如果} \lfloor s/2^{h-1} \rfloor \equiv 1 \bmod 2 \\ g(p_{h-1} \parallel a_{h-1}), & \text{如果} \lfloor s/2^{h-1} \rfloor \equiv 0 \bmod 2 \end{cases} \tag{19}$$

其中 $h=1, \cdots, H$，且 $p_0 = g(Y_s)$。指标 s 用来决定认证路径节点和从叶 $g(Y_s)$到 Merkle 树根路径的节点连接顺序。一次性验证密钥 Y_s 认证成功当且仅当 p_H 等于公钥密码。

3　利用伪随机数产生器产生一次性密钥对

根据第 2 节 MSS 的描述，MSS 私钥包括 2^H 个一次性签名密钥。对大部分的实际应用储存如此庞大的数据量都不可行。文献[3]建议，使用确定的伪随机数产生器(PRNG)可以节省空间且只需储存 PRNG 的种子。那么每个一次性签名密钥必须产生两次，一次是生成 MSS 公钥密码且另一次是在签名阶段。

随后，设 PRNG 是加密安全的伪随机数产生器，即输入一个 n 比特种子 SEED_{in}输出一个随机数 RAND 和一个更新种子 SEED_{out}，并且它们都是 n 比特长。

$$\begin{aligned} &\text{PRNG}: \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n \\ &\text{SEED}_{\text{in}} \mapsto (\text{RAND}, \text{SEED}_{\text{out}}) \end{aligned} \tag{20}$$

3.1　利用伪随机数产生器产生 MSS 密钥对

下面介绍一下如何利用伪随机数产生器(PRNG)产生 MSS 密钥对。第一步是均匀随机选择一个 n 比特种子 SEED_0。为了产生一次性签名密钥，我们使用一个种子序列 SEEDOTS_j，$0 \leqslant j < 2^H$。他们反复计算使用

$$(\text{SEEDOTS}_j, \text{SEED}_{j+1}) = \text{PRNG}(\text{SEED}_j), \quad 0 \leqslant j < 2^H \tag{21}$$

而 SEEDOTS_j 用于计算第 j 个一次性签名密钥。

例如，在 W-OTS 的情况下(见 1.2 节)第 j 个签名密钥是 $X_j = (x_{t-1}, \cdots, x_0)$。在这个签名密钥中长为 n 的 t 个比特字符串是用 SEEDOTS_j 产生。

$$(x_i, \text{SEEDOTS}_j) = \text{PRNG}(\text{SEEDOTS}_j),\ i = t-1, \cdots, 0 \tag{22}$$

在每次调用 PRNG 时更新种子 SEEDOTS_j。这说明为了计算签名密钥 X_j 只有种子方面的知识是必不可少的。当计算 SEEDOTS_j 时，产生签名密钥 X_{j+1} 的新种子 SEED_{j+1} 也被确定。图 4 展示了利用 PRNG 产生一次性签名密钥。

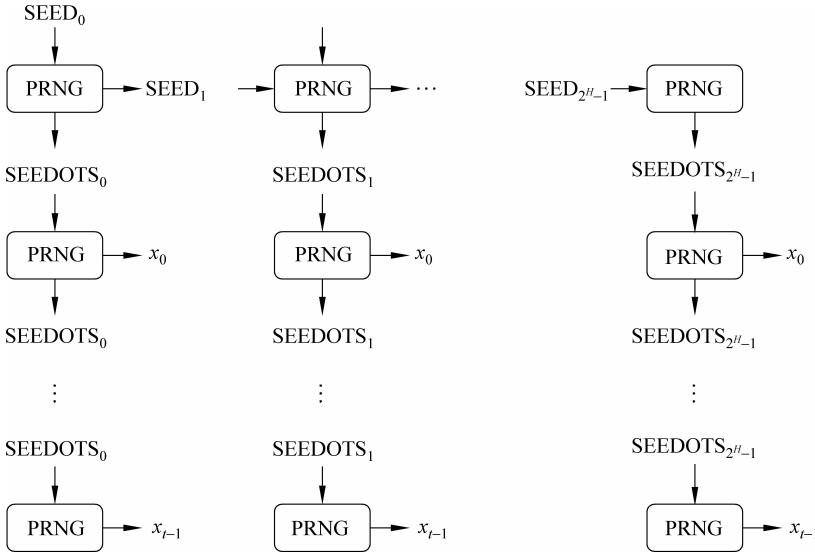

图 4　利用 PRNG 产生一次性签名密钥

如果使用这个方法，最初的 MSS 私人密钥是 SEED_0。它的长度是 n。在产生签名密钥 X_j 时，它会被确定的种子 SEED_{j+1} 取代。

3.2　利用 PRNG 产生 MSS 签名

对比最初的 MSS 签名生成，在签名产生之前必须计算一次性签名密钥。当计算出签名密钥时，下一个签名的种子必须更新。

3.3　前向安全

除了减少私人密钥的尺寸，利用 PRNG 产生一次性签名密钥还有其他的好处。这使得只要 PRNG 是前向安全的，MSS 也是前向安全，即意味着从实际的种子计算出前一个种子是不可行的。签名方案的前向安全意味着在撤销之前发出的所有签名仍然有效。MSS 是前向安全，因为实际的 MSS 私人密钥只能对即将产生的签名生成一次性签名密钥，但是不能伪造前一个。

4　认证路径的计算

在本文中我们将提出高度 H 的 Merkle 树的各种遍历技术。虽然这种技术的使用对验证者是透明的，验证者不需要知道如何生成一组输出，但仅知它们是正确的。因而，这

个技术可以用于需要连续叶认证路径生成和输出的任何结构上。

第一个遍历算法在结构上非常简单且允许储存和计算之间的各种折中。对于参数的一个选择，所需的总空间上界是 $1.5H^2/\log H$ 哈希值，并且在最坏情况下计算工作是每次输出计算 $2H/\log H$ 个树节点。

第二个 Merkle 树遍历算法比以前已知的算法具有更好的空间和时间的复杂性。具体来说，该算法要求每轮最多计算 $2H$ 个树节点和要求存储小于 $3H$ 个节点值。不存在 Merkle 树的遍历算法要求时间小于 $O(H)$ 同时空间小于 $O(H)$，在这个意义下我们证明了这种复杂性是最优化的。

分析前两个算法，一个叶和一个内部节点的计算各算作一个单独的基本操作①。

第三个 Merkle 树的遍历算法与第二个算法有相同的时间和空间复杂性。然而，它的常数因子有显著提高且设计用于实际的执行。叶的计算和内部节点的计算之间是有区别的。为了遍历高度 H 的树，大约需要 $H/2$ 个叶和 $3H/2$ 个内节点的计算。

4.1 经典的遍历

Merkle 树遍历所面临的挑战是要确保所有节点值在需要的时候能以节省时间和空间的方式计算并准备好。为了鼓励新的算法，首先讨论预计平均每一轮的计算，并回顾经典的 Merkle 树遍历。

(1) 平均成本。树里每一个节点是认证路径最后的部分，所以一个有用的措施是估量一次计算每个节点值的总成本。在高度 h 里有 2^{H-h} 个右(左)节点，且如果独立计算，每次花费 $2^{h+1}-1$ 个运算。进位舍入，这是 $2^{H+1}=2N$ 个运算，或者每轮刚好两个运算。每一个高度 $h(0\leqslant h<H)$ 的成本加在一起，我们期望每轮平均需要 $2H=2\log(N)$ 次运算。

(2) 三个组成部分。正如数字签名方案，树的遍历算法由三部分构成：密钥生成，输出和验证。在密钥生成过程中，计算第 1 个认证路径和一些即将出现的认证节点值。

输出过程由 N 轮构成，每个叶子 $s\in\{0,\cdots,N-1\}$。在第 s 轮，第 s 个叶子的认证路径 AUTH_i，$i=0,\cdots,H-1$ 是输出。此外，修改算法的状态以备下一步的输出。

对于第 2 节描述的 Merkle 树的验证阶段与传统验证阶段是一样的。

注意：除了表示当前的认证节点 AUTH_h，需要一些符号描述用于计算即将出现的节点的栈。定义 STACK_h 是一个对象，即包含第 2 节中树哈希算法 1 中所描述的节点值的栈。STACK_h.initialize 和 STACK_h.update 将是设置和逐步执行树哈希的方法。

下面给出算法介绍。

(1) 密钥的生成和设置。密钥生成的主要任务是计算和公布根值。这是第 2 节描述的树哈希算法的直接应用。在这个计算的过程中，每一个节点值被算出，且重要的是记录初始值 AUTH_i，以及每一个 AUTH_i 的即将出现的值。

如果用 $v_h[j]$ 表示高度 h 的第 j 个节点，有 $\mathrm{AUTH}_h=v_h[1]$(这些都是右节点)。高度 h 的“即将出现的”认证节点是 $v_h[0]$(这些都是左节点)。在已完成树哈希算法的状态下，这些节点用来初始化 STACK_h。

① 这不同于测量计算总成本，其中包括调度算法本身等。

算法 2　密钥的生成和设置。

(1) 初始认证节点。对每一个 $h\in\{0,1,\cdots,H-1\}$：计算 $\mathrm{AUTH}_h=v_h[1]$。

(2) 初始的下一个节点。对每一个 $h\in\{0,1,\cdots,H-1\}$：用单一节点的值 $\mathrm{AUTH}_h=v_h[0]$设置 STACK_h。

(3) 公钥。计算并公布树根 $v_H[0]$。

(2) 输出和更新。Merkle 的树遍历算法是运行树哈希算法的一个实例，对于每个高度 h 计算这一层的下一个节点值。每 2^h 轮，认证路径将转移到 h 层的右边，因而需要一个新节点(它的兄弟)作为高度 h 的认证节点。

每一轮的树哈希算法状态有两个计算单元更新。在 2^h 轮后完成节点值的计算，且对于该层的下一个认证节点开始一个新的树哈希实例。

为了明确说明如何更新 AUTH 节点，我们观察如何容易地确定需要更新的高度：高度 h 需要更新的充要条件是 2^h 均匀等分 $s+1$，$s\in\{0,\cdots,N-1\}$ 表示当前的轮数。此外，我们注意到在第 $s+1+2^h$ 轮，认证路径将通过高度为 h 的第 $(s+1+2^h)/2^h$ 个节点。因此，它兄弟的值，(所需新的即将出现的 AUTH_h)由 2^h 叶子值确定，且从叶子数 $(s+1+2^h)\oplus 2^h$ 开始，其中⊕表示位运算 XOR。

用这种语言在算法 3 中概述了 Merkle 的经典遍历算法。

算法 3　经典 Merkle 树遍历。

(1) 设置 $s=0$。

(2) Output：

- 对每一个 $h\in[0,H-1]$输出 AUTH_h。

(3) 更新 AUTH 节点：

对所有的 h 使得 2^h 等分 $s+1$。

- 设置 AUTH_h 是在栈 STACK 里唯一的节点值。
- 设置 $\mathrm{startnode}=(s+1+2^h)\oplus 2^h$。
- STACK_h.initialize(startnode,h)。

(4) 建立栈：

对每一个 $h\in[0,H-1]$：

- STACK_h.update(2)。(每个栈收到两个更新)

(5) loop：

- 设置 $s=s+1$。
- 如果 $s<2^h$ 返回第 2 步。

4.2　分形 Merkle 树遍历

选择术语“分形”是因为要在 Merkle 树大的结构里关注一些小的二叉树。

算法的关键是计算和保留在输出算法中每一步选择的节点值。我们通过使用固定高度 h 的子树集合来描述这个选择。下面从符号开始且提供了算法的直观描述。

符号。从高度为 H 的一个 Merkle 树 TREE 开始，我们介绍更多的符号来处理子树。首先，选择一个子树的高度 $h<H$。我们设在 TREE 里的节点 v 的高度是从 v 到 TREE 的叶子的路径的长度(因此，TREE 的叶子的高度是零)。考虑高度至少是 h 的节点 v。我们定义节点 v 的 h 子树是在 TREE 里高度为 h 且 v 是根的唯一子树。为了简化序列，假设 h 是 H 的除数，且设比值 $L=H/h$ 是子树的层数。我们说节点 v 的一个 h 子树是“在第 i 层”，即对 $i\in\{1,2,\cdots,H\}$ 节点 v 的高度是 ih。对于每一个 i，在第 i 层存在 2^{H-ih} 个这样的 h-子树。

如果对所有的 $i<L$，TREE_i 的根是 TREE_{i+1} 的叶，我们称一列 h 子树 TREE_i($i=1,\cdots,L$)是 h 子树的叠放列。我们阐明子树的符号且在图 5 中提供了 h 子树可视化的叠放列。

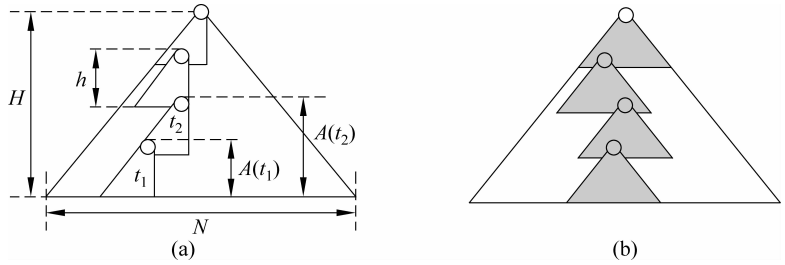

图 5　h 子树可视化叠放列

图 5(a)所示的 Merkle 树的高度是 H，从而叶数 $N=2^H$。每一个子树的高度是 h。标记子树 t_1 和 t_2 的高度为 $A(t_1)$和 $A(t_2)$。图 5(b)储存一个小的集合，即那些叠放的子树，而不是储存所有的树节点。叶的原像作为下一个输出包含在最低的子树里；全部的认证路径包含在子树的叠放集合中。

1. 现有的和期望的子树

(1) 静态视图。如前所述，我们储存一部分节点值，并且随着时间的变化更新储存的值。特别地，在输出阶段的任意点，将存在一系列的叠放现有的子树，如图 5 所示。当我们储存这个节点的时候，就说在树 TREE 上用圆形区域替换一个节点 v。总存在 L 对于每个 $i\in\{1,\cdots,L\}$使得子树 EXIST_i 的每一个节点上有圆形区域(除了它们的根以外)。对于 EXIST_1 中任意的叶，在设计中相应的认证路径完全包含在现有的子树叠放集合中。

(2) 动态视图。除了上面包含以后所需的认证路径的现有的子树集合，我们将有一个期望的子树集合。如果树 EXIST_i 的根有指标 a，根据高度 ih 的节点顺序，那么 DESIRE_i 被定义为具有指标 $a+1$ 的 h 子树(只要 $a<2^{H-ih}-1$)。在 $a=2^{H-ih}-1$ 的情况下，EXIST_i 是这一层的最后一个子树，并且没有相应的期望的子树。特别是，在 L 层不存在一个期望的子树。图 6(a)描绘了相邻的现有的和期望的子树。

顾名思义，我们需要计算在期望的子树里的圆形区域。通过调整树哈希算法的应用完成 DESIRE_i 的根。为了这个目的，调整树哈希算法将需要的圆形区域保存在 DESIRE_i 里，而不是放弃它们，其次要早一轮终止，且不计算根。使用树哈希的变形，我们看到每一

个计算得到的期望子树有一个保存中间圆形区域的尾巴。在图 6(b)描绘了这个动态计算，并说明了部分完成的子树及它们相关的尾巴。

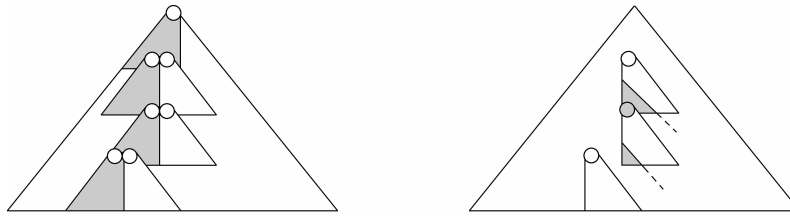

图 6　相邻的现有的子树和期望的子树以及子树的动态计算

图 6(a)中灰色的子树对应现有的子树(见图 5)，而白色的子树对应期望的子树。当现有的子树被用光，期望的子树逐步构建起来。图 6(b)指出来自图 6(a)的期望的子树集合，但灰色的部分对应已经计算的节点且虚线对应尾部的圆形区域。

2. 直观算法

我们现在直观地介绍主要算法，解释为什么现有的子树 EXIST_i 始终可用。

概述。遍历的目的是连续的输出认证路径。按照设计，期望的子树应始终包含下一个认证路径作为输出，同时期望的子树在每一轮包含越来越完整的圆形区域，直到现有的子树失效。

当 EXIST_i 用在最后一次输出，我们说它死了。在那时刻，相邻子树 DESIRE_i 必须计算完成，即将值分配给所有的节点除了它的根(因为后面的节点已经是父树的一部分)。然后树 EXIST_i 作为 DESIRE_i 重生。首先 EXIST_i 所有的旧的圆形区域被放弃；然后 DESIRE_i(以及它们相关的值)的圆形区域被 EXIST_i 采用(一旦发生这种情况，那么将开始计算新的和相邻的子树 DESIRE_i。)如果能确保树 DESIRE_i 上的圆形区域总是能按时计算，以这种方式会看到总有完成的期望的子树 EXIST_i。

修正树哈希算法。如上所述，我们用来计算期望的树的工具是一个应用于 DESIRE_i 根的经典树哈希算法的修改版。此版本的不同之处在于这样两种情况：第一，早一轮停止算法(从而跳过根计算)；第二，每一个高度大于 ih 的圆形区域保存在树 DESIRE_i 里。出于计数的目的，我们不会考虑这些作为正确尾巴的一部分而保存的圆形区域。

摊销计算。对一个特殊的层 i，我们回忆树 DESIRE_i 的计算成本是 $2 \cdot 2^{ih}-2$，因为我们忽略了根的计算。在同一时间，我们知道 EXIST_i 将用于 2^{ih} 输出轮。可以通过计算每轮树哈希的二次迭代来分摊在此期间计算的 DESIRE_i。事实上，在需要之前刚好提前一轮 DESIRE_i 准备好。

因而对每一层，分配两个计算单元确保期望的树能按时完成计算。进而每一轮计算的总量是 $2(L-1)$。

3. 解决方案和算法介绍

三个阶段。我们现在更精确地描述这个主要算法。它包含三个阶段：密钥生成阶段、

输出阶段和验证阶段。在密钥生成阶段(可通过一个相对功能强大的计算机脱机执行),计算树的根并输出,作为一个公钥使用。此外,迭代输出阶段需要一些设置,即在最初现有的子树圆形区域的计算。这些都储存在计算机执行输出阶段。

输出阶段由许多轮组成。在 s 轮过程中,第 s 个叶的认证路径是输出。此外,一些圆形区域被放弃且另一些圆形区域用于计算,以便为后面的输出做好准备。

验证阶段同 Merkle 树传统的验证阶段是相同的且如前所述。我们再次注意到算法生成的输出与传统算法产生的输出没有区别。因此我们没有公布验证阶段的细节,但仅有密钥的生成阶段和输出阶段。

密钥生成。首先,计算和储存叠放现有的子树的最左边集合的圆形区域。每一个相关的圆形区域有一个值、一个位置和一个高度。此外,创建期望的子树列表,对于每一层 $i<L$,在修改的树哈希算法中使用空栈进行初始化。

回忆叶的指标记为 $s\in\{0,1,\cdots,N-1\}$,我们初始化计数器 $\mathrm{DESIRE}_i.\mathrm{position}$ 为 2^{ih},Merkle 树的叶作为下一步计算。

算法 4 密钥生成和设置。

(1) 初始子树,对每一个 $i\in\{1,2,\cdots,L\}$:

- 在现有的子树第 i 层计算所有的(非根)圆形区域。
- 在每个 i 层创建一个空的期望的子树(除了 $i=L$),且叶的位置初始化到 2^{ih}。

(2) 公钥 计算并公布树根。

输出和更新阶段。执行阶段的每一轮由以下过程构成:产生一个输出,现有的子树的死亡和重生,和增长的期望的子树。

在第 s 轮,输出是由第 s 个叶相关的认证路径构成。对于认证路径的圆形区域将储存在现有的子树里。

当需要一个来自给定的现有的子树圆形区域的最后认证路径已经输出,刚死的现有的子树"重生",那么子树不再有用,且我们说它"死"了。至此,相应的期望的子树已经完成,且作为这个完成的所需的子树。注意到每 2^{ih} 轮在第 i 层需要一个新的子树,所以每 2^{ih} 轮在现有的树里的圆形区域都被丢弃。更技术性地,在第 s 轮且 $s=0(\mathrm{mod}\ 2^{ih})$,老树 EXIST_i 里的圆形区域被丢弃。完成的树 DESIRE_i 变成新树 EXIST,并且建立一个新的、空的期望的子树。

在最后一步我们增长每一个只有很少的比特还没有完成的期望的子树。更具体地,我们应用两个计算单位到新的子树或者已经开始调用的树哈希算法。下面简要地介绍该算法。

4. 时间和空间分析

(1) 时间分析。如上所述,算法给每一个期望的子树分配两个计算单元。这里,计算单元被称为 LEAFCALC,或者哈希值的计算。因为最多存在 $L-1$ 个期望的子树,每一轮的计算总量是

$$T_{\max}=2(L-1)<2H/h \tag{23}$$

算法 5　分层 Merkle 树遍历。

(1) 设定 $s=0$。

(2) 输出：认证路径的叶子数 s。

(3) 下一个子树。对每一个 $i\in\{1,2,\cdots,L\}$，EXIST_i 不再需要，即 $s=0(\mathrm{mod}\ 2^{ih})$。

- 在 EXIST_i 上删除圆形区域。
- 将树 EXIST_i 重新命名为树 DESIRE_i。
- 建立新的、空的树 DESIRE_i(如果 $s+2^{hi}<2^H$)。

(4) 增长子树。对每一个 $i\in(1,2,\cdots,h)$；增长子树通过应用两个计算单元到修改的树哈希算法(除非 DESIRE_i 被完成)。

(5) 增加 s 并循环返回到第 2 步(同时 $s<2^H$)。

(2) 空间分析。算法所需的空间总量，或等价的，可用圆形区域需要的数量，可通过简单的计数进行限制，这个值来自于现有的子树、需要的子树或尾巴。

首先，存在 L 个现有的子树和多达 $L-1$ 个期望的子树，并且每一个包含多达 $2^{h+1}-2$ 个圆形区域，因为我们不储存根。此外，在 $i>0$ 层期望的子树相关的尾巴最多包含 $(hi+1)$个圆形区域。如果在尾巴里计算不属于所需的子树的圆形区域，那么“正确的”尾巴最多包含 $h(i-1)+1$ 个圆形区域。增添这些值，可以获得和$(2L-1)(2^{h+1}-2)+h\sum_{i=1}^{L-2}i+1$，因而有以下限制：

$$\mathrm{Space}_{\max}\leqslant(2L-1)(2^{h+1}-2)+L-2+h(L-2)(L-1)/2 \tag{24}$$

一个稍微差些的限制简写成：

$$\mathrm{Space}_{\max}<2L2^{h+1}+HL/2 \tag{25}$$

(3) 折中方案。分析解决方案，我们提出了时间和空间之间的一种折中方案。一般的，对于更大的子树，算法运行速度更快，但是需要更大的空间。在折中方案中影响空间和时间的参数是 h；计算成本低于 $2H/h$，所需的空间限制在 $2L2^{h+1}+HL/2$ 以上。另外，用 h 来表示空间限制在 $2H2^{h+1}/h+H^2/2h$。

(4) 较低空间解决方案。如果对需要很少空间的参数感兴趣，这存在一个最优的 h，由于实际上对十分小的 h，尾部圆形区域增长的数量十分显著(当 $H^2/2h$ 变得非常大的时候)。近似值是 $h=\log H$。可以通过对空间区分表达找到一个精确值：$2H2^{h+1}/h+H^2/2h$。对于 $h=\log H=\log\log N$ 的选择，我们获得

$$T_{\max}=\frac{2H}{\log H} \tag{26}$$

$$\mathrm{Space}_{\max}\leqslant\frac{5}{2}\frac{H^2}{\log H} \tag{27}$$

这些结果是有趣的，因为它们逐步提高了 4.1 节中 Merkle 的时间和空间的结果。Merkle 的方法需要 $T_{\max}=2H$ 和 $\mathrm{Space}_{\max}\approx H^2/2$。

5. 额外的储存

现在回到主要算法，并解释较小的技术改造是如何提高限制空间的常数，最终产生声

称的结果。

虽然这个修改不会影响时间或空间花费的复杂性分类，在某些情况下空间限制折半作为实际的利益。为了保留原有的简化形式，在主要的阐述之后提到的分析是稍多技术性的。这基于以下两点修改：

(1) 在现有的子树中存在的圆形区域不再使用。

(2) 期望的子树总是处在部分完成的状态。

事实上，我们发现在现有的子树中的圆形区域可能被丢弃几乎和圆形区域进入相应的期望的子树一样快。修改如下：

(1) 在树 EXIST_i 放弃圆形区域，只要它们不再需要。

(2) 忽略 2 个单位的首次应用到修改后的树哈希算法。

我们注意到通过第 2 次修改，期望的子树正好依旧完整。对所有的层 $i<L$ 包含在 EXIST_i 和 DESIRE_i 里的圆形区域的数量有些小的变化，可由下面的表达式界定。

$$\mathrm{Space}_{\mathrm{EXIST}_i}+\mathrm{Space}_{\mathrm{DESIRE}_i}\leqslant 2^{2^h+1}-2+(h-2) \tag{28}$$

这几乎是以前限制 $2(2^{2^h+1}-2)$的一半。在这里标记的数量 $h-2$ 是包含在 DESIRE_i 里圆形区域的最大值超过在 EXIST_i 里放弃的圆形区域数量。估计方程(28)，通过计算，我们修改节中的空间限制，得到：

$$\mathrm{Space}_{\max}\leqslant(L)(2^{h+1}-2)+(L-1)(h-2)+L-2+h(L-2)(L-1)/2 \tag{29}$$

我们再次获得一个简单的限制：

$$\mathrm{Space}_{\max}<L2^{h+1}HL/2 \tag{30}$$

特地选择 $h=\log H$，通过降低参数从$\frac{5}{2}$到$\frac{3}{2}$，我们将以上的结果提高到

$$\mathrm{Space}_{\max}\leqslant\frac{3}{2}\cdot\frac{H^2}{\log H} \tag{31}$$

空间边界的证明。这里证明了方程(28)的论断，对任何层 i 在 EXIST_i 里的圆形区域数量加上 DESIRE_i 里圆形区域的数量的状态少于 $2\cdot 2^{2^h}-2+(h-2)$。这个基本的观测反映了一个事实，期望子树的增长略快于现有的子树的收缩。不失一般性，为了简化阐述，我们没有规定子树的指标，且在给定的层 i 限制我们的注意力在现有的期望的子树对上。

第 1 次修改确保圆形区域较以前更持续地传回，所以我们量化它。子树 EXIST_i 有 2^h 个叶子且不再需要每一个叶子，也不是它上面的一些内部节点。这些叶子在第 $2^{(i-1)h}a-1$ 轮完成，其中 $a\in\{1,\cdots,2^h\}$。我们通过观察叶子在每一轮传回的时间来确定圆形区域返回的数量，每两轮在高度 $ih+1$ 上有一个圆形区域，每四轮在高度 $ih+2$ 上有一个圆形区域，等等。我们对任何时间返回的数量感兴趣直到时间 $2^{(i-1)h}a-1$；这是最好的整数函数的和：

$$A+[A/2]+[A/4]+[A/8]+\cdots+[A/2^h]$$

用二进制表示法将 a 写成 $a=a_0+2^1a_1+2^2a_2+\cdots+2^ha_h$，这个和也是

$$a_0(2^1-1)+a_1(2^2-1)+a_2(2^3-1)+\cdots+a_h(2^{h+1}-1)$$

计算 DESIRE_i 里相应圆形区域的成本也可计算出一个类似的表达式。利用事实高

度 h_0 的节点需要 $2^{h_0+1}-1$ 个计算单位，我们看到期望的子树需要

$$a_0(2^{(i-1)h+1}-1)+a_1(2\times 2^{(i-1)h+2}-1)+\cdots+a_h(2\times 2^{ih+1}-1)$$

计算单位来放置那些同样的圆形区域。花费等于 $2\times 2^{(i-1)h}\cdot a-z$，其中 z 表示 a 的二进制表达式中非零数字的个数。

在时间 $2^{(i-1)h}a-1$，共有 $2\times 2^{(i-1)h}a-2$ 个计算单位应用到 DESIRE_i 里（延迟一轮分解）。注意到在 EXIST_i 失去更多的圆形区域之前可能有多于 $2^{(i-1)h}-1$ 轮通过，我们看到在此间隔中圆形区域的最大数目必须在间隔结束时实现。在这个时间点，期望的子树精确计算从现有的树中删除的圆形区域，加上任何额外的圆形区域它可以用剩余的 $2\times 2^{ih}-2+z-2$ 个计算单位计算。下一个圆形区域，（一个叶子）花费 $2^{ih}-1$ 留下 $z-3$ 个计算单位。即使所有的单位在新的圆形区域结果里，总的额外数仍小于等于 $1+z-3$。因为 $z\leqslant h$，额外的圆形区域的数量限制在 $h-2$，如所期望的，方程(28)被证明。

4.3　log 时空的 Merkle 树遍历

本节对 4.1 节经典的遍历算法提出一点看法。我们看到上面经典的算法，树哈希算法的至多 H 个实例可能是同时活跃的，且每个高度比 H 少。概念化它们作为并行运算的 H 个过程，对于“尾节点”的树哈希算法每个也需要一定的空间，并接收每一轮计算两个哈希值的预算，显然能够完成 $2^{h+1}-1$ 个哈希计算需要超过 2^h 个可用轮数。

因为树哈希使用的栈可能至多包含 $h+1$ 个节点值，所以我们只能保证 $1+2+\cdots+H$ 的空间限制。这么多尾节点的可能性正是经典算法中 $\Omega(H^2/2)$ 空间复杂性的缘由。

考虑更大的 h，哈希树计算需要多轮才能完成，它似乎一次保存这么多的中间节点有些浪费。我们的想法是调度不同的并行树哈希计算，所以在任意给定的轮 $s\in\{0,\cdots,2^H-1\}$，相关的栈大部分是空的。我们选择了对较低 h 通常有利计算即将出现的认证节点 AUTH_h 的一个调度，（因为它们不久就需要），但略推迟哈希树算法的一个新例子的开始，直到不包含高度小于 h 的尾节点的所有栈 STACK_i 部分完成。

这种延迟来自观察，在一般情况，如果在相同高度不同的树哈希栈里的两个节点的计算是连续地计算，而非平行，将使用更少的空间。非正式地，我们所说的延迟是在开始的新栈里计算“压缩的尾巴”。我们需要证明不再明显的事实，即将出现的需要的节点总是及时的准备好。

1. 新的遍历算法

本节描述新的调度算法。相较于经典遍历算法，唯一的区别是如何使得 $2H$ 个哈希函数赋值预算分配给潜在的 H 个并行的树哈希过程。

定义 $\mathrm{STACK}_h.\mathrm{low}$ 是 STACK_h 里最低点的高度，除了两种情况：如果栈是空的 $\mathrm{STACK}_h.\mathrm{low}$ 被定义为 h，且如果树哈希算法完成 $\mathrm{STACK}_h.\mathrm{low}$ 被定义为 ∞。

使用压缩尾巴的想法，有不止一种方法利用这种储存优势的调度算法。我们在这提出的不是最佳的，但它的描述简单。在 4.5 节讨论更多的实际改进。

此版本可简要地描述如下。

在经典遍历中计算即将出现的需要认证的节点，但各栈得不到平等的重视。每一个

树哈希实例可赋予为未被启动，部分完成或者已完成。

我们的调度更倾向于最低 h 值完成的 STACK_h，除非另一个栈有一个更低的尾节点。我们通过定义 $l_{\min}$ 是 h 值 $\mathrm{STACK}_h.\mathrm{low}$ 的最小值来表达这种偏好，然后将我们的注意力集中在最小层 h 以获得这个最小值（设置 $\mathrm{STACK}_h.\mathrm{low}=\infty$，对于完成的栈有效地跳过它们）。

换句话说，在我们开始一个新的 STACK_h 树哈希计算之前，所有的栈必须完成没有尾节点的高度是 h 或者更少的阶段。在算法 6 中对最后的算法进行了概述。

算法 6 对数 Merkle 树遍历。

（1）设 $s=0$。

（2）输出：

- 对每一个 $h\in[0, H-1]$ 输出 AUTH_h。

（3）更新 AUTH 节点。

对所有的 h 使得 2^h 等分 $s+1$：

- 设 AUTH_h 是 STACK 的唯一节点。
- 设始节点 $=(s+1+2^h)\oplus 2^h$。
- STACK. initialize(始节点，h)。

（4）建立栈。

重复下列步骤 $2H-1$ 次：

- 设 $l_{\min}$ 是 $\mathrm{STACK}_h.\mathrm{low}$ 的最小值。
- 设 focus 至少是 h，所以 $\mathrm{STACK}_h.\mathrm{low}=l_{\min}$。
- $\mathrm{STACK}_{\mathrm{focus}}.\mathrm{update}$。

（5）循环。

- 设 $s=s+1$。
- 如果 $s<2^H$ 返回第 2 步。

2. 正确性和分析

本节说明 $2H-1$ 个计算预算是在它作为一个需要的验证节点之前足够完成每一个 STACK_h 计算。我们还指出哈希值需要的空间小于 $3H$。

准时计算节点。如上所述，算法确切的分配每一轮 $2H-1$ 个计算单位的预算来花费更新 h 个栈。这里，一个计算单元一次调用 LEAFCALC，或者一个哈希值的计算。由于叶的计算复杂性我们没有将任何额外的费用表示成模型。

为了证明这一点，我们设定一个特定的高度 h，并且我们设想从创立时间 STACK_h 开始和即将出现的需要完成的验证节点（这里记为 NEED_h）完成的结束时间。由于复杂的调度算法使得这些尚不清楚。我们的方法证明 NEED_h 能按时完成，这说明在此期间的总预算超过了在 NEED_h 之前的计算期间中所有节点总的计算成本。

节点 NEED_h 自己的花费仅为 $2^{h+1}-1$ 个单位，假定一个易于处理的数量即在创建时间 STACK_h 和 NEED_h 必须完成的时间之间存在 2^h 轮。然而，需要一个非平凡的计算，因为除了 NEED_h 所需的资源，在此期间一些其他的节点竞争 $2H2^h$ 个可用的计算单位的

总预算。在较低的层这些节点包含所有将来需要的节点 $NEED_i$，$(i<h)$。最后可能部分的值给节点 $NEED_i$，$i>h$，通过完成的时间 $NEED_h$ 以便它的栈不含有较低的节点。

在时间间隔内计算这些需要的节点数量是容易的，且我们知道每一个的成本。由于较高栈的值，我们至少知道涉及高度 h 的较低节点的成本少于 $2^{h+1}-1$（高度 h 节点的总成本）。表 1 列出了这些数量和成本。

表 1　在 $NEED_h$ 的 2^h 轮里建立节点

节点类型	数 量	每一个的成本	节点类型	数 量	每一个的成本
$NEED_h$	1	$2^{h+1}-1$	⋮	⋮	⋮
$NEED_{h-1}$	2	2^h-1	$NEED_0$	2^h	1
⋮	⋮	⋮	TAIL	1	$\leqslant 2^{h+1}-2$
$NEED_k$	2^{h-k}	$2^{k+1}-1$			

我们继续统计在时间间隔内产生的总成本。注意开始的 $NEED_0$ 行需要总共 2^{h+1} 个计算单位。对于节点表中的其他行，每个节点的成本乘以一个给定类型的节点数量均小于 2^{h+1}。存在 $h+1$ 个这样的行，所以在图表中表示的所有节点的总成本是

$$\mathrm{TotalCost}_h < (h+2)2^h \tag{32}$$

当高度 $h \leqslant H-2$ 时，显然总成本小于 $2H2^H$。对剩下的 $h=H-1$ 情况，这也是正确的。因为在这种情况下没有尾节点。

我们得出这样的结论，如期望的，能按时充足的为 $NEED_h$ 准备每轮 $2H-1$ 个单位的预算对任意的 $0 \leqslant h < H$。

空间限制在 $3H$。我们的动机导致相对复杂的调度使用尽可能少的空间。为了证明这一点，我们简单地添加了每一类节点的数量。我们知道总是存在 H 个节点 $AUTH_h$。设 $C<H$ 是完成的节点 $NEED_h$ 的数量。

$$\#\mathrm{AUTH}_i + \#\mathrm{NEED}_i = H + C \tag{33}$$

最后必须考虑在 $STACK_h$ 里尾节点的数量。至此，我们观察到因为 $STACK_h$ 从未变得活跃直到在“高”栈中的所有节点的高度至少是 H，不可能有两个不同的栈，每个都包含一个相同高度的节点。此外，看一下树哈希算法，我们知道至多一个高度对一个栈有两个节点值。总之，在每一个高度（$0 \leqslant h \leqslant H-3$）里至多有一个尾节点，每一个未完成的栈里有一个额外的尾节点。因而

$$\#\mathrm{TAIL} \leqslant H-2+(H-C) \tag{34}$$

添加所有类型的节点得到：

$$\#\mathrm{AUTH}_i + \#\mathrm{NEED}_i + \#\mathrm{TAIL} \leqslant 3H-2 \tag{35}$$

这证明了断言。最多存在 $3H-2$ 个储存节点。

4.4　渐进最优结果

一个有趣的最优结果指出遍历算法从未被时间 $O(\log(N))$ 和空间 $O(\log(N))$ 击败。很显然树哈希算法至少需要 $H-2$ 个节点，所以我们的任务基本是说明如果空间被限制

是 $\log(N)$ 的常数倍，那么计算复杂性一定是 $\Omega(\log(N))$。我们必须清楚这个定理没有量化常数。显然，随着空间的增大，计算时间可减少。

定理 1 假设有一个 Merkle 树遍历算法的空间限制在 $\alpha\log(N)$。那么存在一些常数 β 使得需要的时间至少是 $\beta\log(N)$。

定理简单地指出它是不可能在不增加超出对数时间复杂性来减少空间复杂性，使得空间复杂性低于对数。

这个技术的描述证明将在下面的小节中找到，但在这里我们简单地介绍这个方法。对于这个证明我们只考虑右节点。我们把所有的右节点分成两个组：必须计算的($2^{h+1}-1$ 个成本)节点，和在以前计算中保存的节点。这个证明假定亚对数时间复杂性并推出一个矛盾。

在第 2 类中节点越多，遍历执行越快。然而，为了减少到亚对数时间复杂性并保存这样巨大数量的节点，即保存的节点值的平均数将超过线性数量。下面提到的精确技术证明使用一个特定的子树序列来制造矛盾。

我们现在开始定理 1 的技术证明。这是一个反证法。我们假设时间复杂性是亚对数的，且说明这与空间复杂性 $O(\log(N))$ 是不相符的。我们产生一个矛盾的策略是找到时间和空间的平均消耗量的线性组合的限制。

符号。定理是一个渐进的描述，所以对于较大的 H，我们考虑高度 $H=\log(N)$ 的树。我们需要考虑高度 k 的子树的 L 层，其中 $kL=H$。在主要的树里，这些子树的根将是高度 $k,2k,3k,\cdots,H$。如果它的根是高度 $(i+1)k$ 我们称子树在第 i 层。这个子树的概念类似在 4.2 节中使用的。

注意我们只需要考虑右节点来完成我们的讨论。回想一下在一个完整的树遍历中每个唯一的右节点作为最终输出的认证数据的一部分。这提示我们将右节点分成三类。

(1) 在密钥生成后已存在的：自由节点。

(2) 明确计算的节点(如在树哈希里)：计算节点。

(3) 来自另外节点的计算保留(如来自另外节点的树哈希)：保存节点。

注意类型(2)的节点需要的计算工作，而类型(1)和类型(3)的节点需要一定的储存期。为了方便这些节点的描述，我们需要进一步的符号说明。设 a_i 表示包含至少一个非根的可计算的(右)节点的层 i 子树的数量。同样地，设 b_i 表示包含零计算节点的层 i 子树的数量。仅通过计算层 i 子树的总数我们有关系

$$a_i + b_i = N/2^{(i+1)k} \tag{36}$$

计算成本。让我们标记可计算的节点的一些成本。存在 a_i 个子树含有两类型的节点，其中高度至少是 ik。每个这样的节点将花费至少 $2^{ik+1}-1$ 次运算去计算。取整，在第 i 层对于节点的花费我们找到一个简单的较低限制。

$$\text{Cost} > \sum_{i=0}^{L-1}(a_i 2^{ik}) \tag{37}$$

储存成本。让我们标记留存节点的一些使用寿命。当考虑消耗的平均空间时，空间×轮数的计算单位是自然。一般地，一个储存的节点 S，来自于可计算节点 C 的计算结果，