软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007394]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

抗量子的高效区块链认证存储方案 *

张川[1], 任旭豪[1], 邓 天[1], 王亚杰[1], 李春海[3], 吴桐[2], 王励成[1]

[1](北京理工大学 网络空间安全学院, 北京 100081)
[2](北京科技大学 计算机与通信工程学院, 北京 100081)
[3](桂林电子科技大学 信息与通信学院, 广西 桂林 541004)
通讯作者: 吴桐, E-mail: tongw@ustb.edu.cn

摘 要: 随着区块链技术的广泛应用, 认证存储作为其核心组件, 承担着确保数据完整性和一致性的重要作用。在传统区块链系统中, 认证存储通过一系列密码算法来验证交易和维护账本状态的完整性。然而, 量子计算机的出现使得现有区块链认证存储技术面临被破解的威胁, 使得区块链面临数据泄露和完整性受损的风险。当前最先进的认证存储技术主要基于双线性 Diffie-Hellman 假设构造的, 该构造难以抵抗量子攻击。为提高认证存储的安全性和效率, 本文引入一种无状态哈希签名技术, 提出抗量子的区块链认证存储方案 EQAS。该方案通过将数据存储和数据认证解耦, 利用随机森林链来高效地生成承诺证明, 同时通过超树结构来执行高效认证。安全性分析表明, EQAS 可以抵御量子算法的攻击。通过与其他认证存储方案的对比, 实验结果验证了 EQAS 方案的高效性, 展现出其在处理区块链认证存储任务时的卓越性能。

关键词: 区块链;抗量子;认证存储;无状态哈希签名
中图法分类号: TP311

中文引用格式: 张川,任旭豪,邓淏天,王亚杰,李春海,吴桐,王励成. 抗量子的高效区块链认证存储方案.软件学报. http://www.jos.org.cn/1000-9825/7393.htm
英文引用格式: Zhang C, Ren XH, Deng HT, Wang YJ, Li CH, Wu T, Wang LC. Quantum-resistant and Efficient Blockchain Authentication Storage Scheme. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7394.htm

Quantum-resistant and Efficient Blockchain Authentication Storage Scheme

ZHANG Chuan[1], REN Xu-Hao[1], DENG Hao-Tian[1], WANG Ya-Jie[1], LI Chun-Hai[3], WU Tong[2], WANG Li-Cheng[1]

[1](School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China)
[2](School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100081, China)
[3](School of Information and Communication, Guilin University of Electronic Technology, Guilin, Guangxi 541004, China)

Abstract: With the widespread application of blockchain technology, authenticated storage, as its core component, plays an important role in ensuring data integrity and consistency. In traditional blockchain systems, authenticated storage verifies transactions and maintains the integrity of ledger status through a series of cryptographic algorithms. However, the emergence of quantum computers has made the existing blockchain authenticated storage technology face the threat of being cracked, which makes blockchain face the risk of data leakage and integrity damage. The most advanced authenticated storage technology is mainly based on the bilinear Diffie-Hellman assumption, which may become vulnerable to quantum attacks. In order to improve the security and efficiency of authenticated storage, this paper introduces a stateless hash signature technology and proposes a quantum-resistant blockchain authenticated storage scheme EQAS. This scheme decouples data storage and data authentication, uses random forest chains to efficiently generate commitment proofs, and performs efficient authentication through a hypertree structure. Security analysis shows that EQAS can resist attacks from quantum algorithms. By comparing with other authenticated storage schemes, experimental results verify the efficiency of the EQAS scheme and show its excellent performance in processing blockchain authenticated storage tasks.

Key words: blockchain; quantum-resistant; authentication storage; stateless hash signature

* 基金项目: 国家重点研发计划项目(2022YFB2702700); 国家自然科学基金项目(62232002, 62202051); 北京理工大学青年教师学术启动计划; 中国科协青年人才托举工程(2023QNRC001)

收稿时间: 2024-07-01; 修改时间: 2024-09-05; 采用时间: 2024-12-30; jos 在线出版时间: 2025-01-20

在数字化时代，区块链技术以其独特的去中心化特性和不可篡改性，已经在金融服务、供应链管理、智能合约等多个领域展现出其革命性的潜力[1][2]。作为区块链架构中的重要组成部分，认证存储技术在确保数据完整性和一致性方面发挥着关键作用。在传统区块链中，认证存储依赖于一系列加密算法，这些算法负责验证每笔交易的有效性，保障账本状态的不可篡改性，并为网络中的每个参与者提供透明的数据访问[3][4]。

尽管区块链认证存储技术在确保数据安全和信任方面取得了显著成就，但量子计算机的迅速发展给现有的区块链认证存储技术带来了威胁。当前主流的认证存储技术，如基于 RSA 或椭圆曲线密码体制的签名算法，依赖于大数分解和离散对数等困难问题。然而，随着量子计算技术的进步，Shor 算法等量子算法[8][19]能够有效分解大整数，进而破解基于这些数学困难问题的加密机制。具体来说，现有的基于签名的认证存储技术在量子攻击下会变得脆弱，这意味着攻击者可以伪造认证信息，插入恶意数据，系统将无法检测到数据的篡改，最终导致区块链系统的数据完整性和一致性丧失。此外，基于哈希的认证存储技术同样面临威胁，例如，量子计算能够通过 Grover 算法[17][18]加速哈希碰撞的搜索，使得哈希函数不再具备足够的抗碰撞能力，这导致区块链中的数据认证不再可靠[20]。因此，为了保持区块链系统的长期安全性和可靠性，必须对现有的认证存储技术进行改进，以确保能够抵御量子计算时代的安全威胁。

现已有一些区块链认证存储工作，例如简化支付验证（SPV）[9][10]允许比特币轻节点通过下载区块头和使用 Merkle 树证明来验证交易，而无需下载整个区块链。SPV 轻节点依赖全节点提供区块头和交易证明，以节省存储和带宽，适用于资源有限的设备；以太坊中的 Merkle Patricia Trie (MPT)结构，每个叶子节点存储一个值，内部节点则存储其子节点内容的加密哈希值，根节点哈希用作区块链状态的承诺，以确保数据完整性和可验证性；Li 等人[3][4]提出高效的区块链认证存储系统（LVMT），采用多层设计来支持无限的键值对，并存储版本号而非哈希值，避免了昂贵的椭圆曲线乘法操作，显著减少了 I/O 放大效应。Zhang 等人[11]提出了一种用于区块链系统的新型基于列的学习存储，来降低区块链存储大小并提高系统吞吐量。然而，上述工作都是基于如离散对数、双线性 Diffie-Hellman 假设等困难问题构造的，这些问题随着量子计算机的发展都是可以被破解的，这导致认证存储机制面临着安全风险[13]。

密码学界提出了多种抗量子方案，包括基于格理论、代码理论和多变量方程的密码系统。这些方案虽在理论上具有良好的抗量子能力，但普遍存在一些问题。例如，基于格的加密方案虽然被广泛认为是抗量子密码的强有力候选者，但其计算复杂度较高，密钥和签名的尺寸相对较大，在区块链等需要高效存储和快速验证的场景中并不理想。基于代码和多变量方程的密码方案同样面临类似的挑战，它们的实现通常需要较高的计算和存储开销，导致在实际应用中效率较低。与这些复杂的方案相比，基于哈希的签名方案因其结构简单、效率高、实现容易而逐渐成为抗量子密码的重点研究方向，这使其特别适合区块链中需要快速验证大量交易和状态数据的场景。此外，哈希签名方案的无状态设计能够很好地适应区块链系统，减少全节点的存储负担和计算复杂度，特别是在轻节点与全节点频繁交互的认证存储场景中，可以显著提高系统的可扩展性和性能。如今，基于哈希的签名方案是两个 RFCs 中正式定义的第一个抗量子签名方案[14][15]，而本文采用的是 NIST 后量子加密标准化项目第二轮中的九个签名提案之一的 SPHINCS+[15]的优化方案[26]。鉴于此，本文提出了抗量子的高效认证存储方案 EQAS，主要贡献包括：

（1）结合基于无状态哈希签名与区块链认证存储，首次提出抗量子的区块链认证存储机制，在实现高效的认证存储的同时，还可以抵御量子计算时代的安全威胁；

（2）创新性地将随机森林算法动态化，设计出 DFORC，以优化其在区块链场景中的应用。EQAS 将数据存储和数据认证解耦，通过动态随机森林链来高效地生成承诺证明，使用超树作为认证树来执行高效认证；

（3）对 EQAS 的安全性和效率进行了理论分析和实验，在安全性方面证明了 EQAS 可以有效抵御量子攻击，在效率方面 EQAS 表现出较高的性能。

本文第 1 节介绍区块链认证存储和抗量子攻击的相关工作。第 2 节介绍本文所需的基础知识,包括认证存储的概念和基于哈希的签名。第 3 节介绍 EQAS 方案概述，包括系统模型、工作流程和威胁模型。第 4 节介绍 EQAS 的详细设计过程。第 5 节通过安全性分析证明了 EQAS 的量子安全。第 6 节通过实验验证了所提方案的有效性。最后总结全文。

1 相关工作

1.1 认证存储

最初，在分布式系统外包存储的背景下，认证存储主要关注如何在服务器不可信的情况下，确保存储数据的完整性和可用性。随着云计算的兴起，云存储技术成为研究的热点，它通过提供动态、可扩展的存储资源，满足了大数据时代对存储的需求。云存储的关键技术包括数据加密、访问控制和数据去重等，以确保数据的安全性和隐私性[30]。区块链技术的引入为认证存储带来了新的变革[31]。区块链的不可篡改性和去中心化

特性，使得它在身份认证、数据完整性验证等方面展现出巨大潜力。例如，基于区块链和密码累加器的自我主权身份认证方案，通过精简区块链数据存储和降低链上数据交互频率，提高了系统性能和可扩展性，同时还能再次隐藏已披露的用户数据，增强了系统的安全性。

在区块链中，认证存储通常采用 Merkle Patricia Trie (MPT)结构[21]，这是一种特定形式的 Merkle 树。MPT 中的每个叶子节点存储一个值，根节点到叶子节点的路径对应于存储值的键。每个内部节点存储其所有子节点的加密哈希值。MPT 的根哈希作为区块链状态的承诺进行认证。然而，这种认证需要付出沉重的性能代价。修改状态中的键值对需要更新 MPT 从相应的叶子节点到根节点的路径中所有节点的哈希值。如果没有缓存，每个状态更新操作放大为 $O(\log(n))$ 个 I/O 操作，其中 n 是存储的大小。然而，即使是简单的支付交易也至少包含两个账本状态更新，即分别扣除和增加发送方和接收方的余额。为改善 MPT 结构，mLSM 提出多层 MPT[22]。最近的更新位于最低层，较低层的键值对将定期合并到较高层。LMPT 维护三个 MPT，一个包含旧状态的大 MPT 和两个包含最近状态变化的小 MPT[23]。LMPT 周期性地将小的 MPT 合并为大的 MPT。对于 mLSM 和 LMPT，所有 MPT 的 Merkle 根的拼接成为账本状态的承诺。因为最近访问的状态存储到具有较小高度的 MPT 中，并且 MPT 的合并可以在后台线程中进行，所以这些技术减少了关键路径上的磁盘 I/O 操作。然而，mLSM 论文只包含其概念设计，没有实现和评估[22]。

此外，基于预存节点的认证存储模型 RainBlock[24]，在区块链系统中引入了三种不同的节点来加速交易的执行：存储预取器、执行交易的矿工和存储节点。在执行交易时，矿工从多个预取器获取所需数据，并将更新发送到多个存储节点，每个存储节点在内存中维护一个 MPT 分片。RainBlock 将本地存储 I/O 更改为网络分布式存储 I/O。为了减少网络存储的读取延迟，RainBlock 引入了 I/O 预取器，并要求矿工在广播区块时附加所有访问的键值对和见证人(MPT 节点)。RainBlock 报告每笔交易的平均证明大小为 4KB，通过优化将证明大小减少了 95%，因此每笔交易的额外网络存储约为 200B，相当于一笔交易的两倍。然而，网络的低效使用给高性能区块链系统带来了瓶颈[25]。此外，RainBlock 在数据可用性方面也容易受到攻击。由于内存存储成本很高，因此 RainBlock 中的副本数量远少于以太坊。

然而，这些方案均基于离散对数难题和大数分解等数学困难问题构造的。随着量子计算技术的飞速发展，这些难题可能被攻克，从而使得现有的认证存储机制可能面临极大的安全风险，进而威胁到数据的完整性与真实性。因此，开发能够抵御量子计算攻击的区块链认证存储方案变得尤为迫切，以确保系统安全稳固。

1.2 抗量子签名

在本节主要讨论基于哈希的数字签名算法：SPHINCS。它是为了在后量子时代保持安全性而设计的。随着密码学的发展，SPHINCS 经历了几个版本的迭代，包括 SPHINCS+和最近提出的 SPHINCS-α。

SPHINCS 最初在 EUROCRYPT 2015 上被提出[27]，它利用了哈希函数的抗量子特性来构建一个无状态的数字签名方案。这种无状态的设计意味着签名者不需要维护任何关于以往签名的状态信息，这在某些应用场景中是非常有用的。SPHINCS+ [15]是 SPHINCS 的一个改进版本，它在保持了原始设计的安全特性的同时，通过引入一些新的技术和方法，提高了算法的效率和安全性。SPHINCS+提出了三种不同的签名方案，分别是 SPHINCS+-SHAKE256、SPHINCS+-SHA-256 和 SPHINCS+-Haraka，这些方案通过使用不同的哈希函数实例化来实现。SPHINCS+的设计目标是减少签名的大小，同时保持较高的安全性。在第二轮提交中，SPHINCS+引入了简单和健壮两种变体，简单变体通过纯随机预言机实例化来实现，相比健壮变体在速度上有约三倍的提升，但在安全性上可能存在一定的折衷。SPHINCS-α [26]是在 SPHINCS+ 的基础上进一步优化的签名方案。它通过改进 Winternitz 一次性签名方案和引入一种新的几次签名方案 FORC，提高了签名的效率。SPHINCS-α 在签名大小和签名时间上相比于 SPHINCS+有所减少，特别是在追求小签名大小的参数设置下，签名大小和签名时间减少了 8%左右。此外，SPHINCS-α 在快速签名操作的参数设置下也展现出了更好的性能。

2 基础知识

本文所提方法主要是结合区块链的认证存储与基于哈希的签名（包括平衡 WOTS+，随机森林链和超树），下面就相关概念和基本知识予以介绍。

2.1 区块链的认证存储

在标准的公有区块链系统中，区块链节点可以分为两种类型:全节点和轻节点。全节点将同步和执行所有的交易，并相应地维护区块链账本状态。轻节点(客户端)只同步区块头，不同步交易和区块链账本状态。当全节点提出新的区块时，执行区块内的所有交易，并将执行完毕的账本状态的承诺放在区块头中。因此，区块链全节点在交易执行中维护回写缓存，并在执行完块中的所有交易后刷新存储。因此，认证存储需要提供两个算法:

- $Get(k)\to v$：获取给定键k的值v。
- $Set(\{(k,v)_i\},e)\to comm$：将键值对$(k,v)$刷新到块号为$e$的存储中，获得变更后账本状态承诺。

当轻节点想要获得键的对应值时，它向全节点查询，全节点返回与值相关的账本承诺和证明。轻节点将检查承诺是否为有效承诺，并验证证明的正确性。因此，认证存储需要提供两种算法来证明和验证:

- $Respond(k)\to(v,\pi,comm)$：根据最新的承诺，用证明$\pi$回答键$k$的值$v$。
- $Verify(k,v,\pi,comm)\to 0/1$：验证来自全节点的响应。

2.2 哈希函数

可调整的哈希函数：可调整哈希函数以公共参数P、调整T和消息 m 作为输入。其中，公共参数可以看作一个密钥函数，而调整可以看作盐值或者随机数。基于 SPHINCS+[15]，本文采用了后续的简化表示法。对于输入长度为$k\lambda$的可调整哈希函数，有$Th_k: Th_k=\{0,1\}^{\lambda}\times\{0,1\}^{256}\times\{0,1\}^{k\lambda}\to\{0,1\}^{\lambda}$。扩展哈希函数定义为$F\triangleq Th_1, H\triangleq Th_2$。

伪随机函数和消息摘要：在本文中，伪随机函数PRF用于生成伪随机密钥，其定义为$PRF:\{0,1\}^{\lambda}\times\{0,1\}^{256}\to\{0,1\}^{\lambda}$，而另一个伪随机函数$PRF_{msg}$则用于生成对消息进行压缩的随机性，其定义为$PRF_{msg}:\{0,1\}^{\lambda}\times\{0,1\}^{\lambda}\times\{0,1\}^{*}\to\{0,1\}^{\lambda}$。此外，本文利用一个能够处理任意长度消息的密钥哈希函数H_{msg}来压缩要签名的消息，其定义为$H_{msg}=\{0,1\}^{\lambda}\times\{0,1\}^{\lambda}\times\{0,1\}^{\lambda}\times\{0,1\}^{*}\to\{0,1\}^{\lambda}$。

2.3 平衡 WOTS+

WOTS+方案在编码过程中引入校验和，防止攻击者在给定有效的消息和签名对的情况下有效伪造签名。给定(σ,m)，攻击者可以通过计算$H^{m_i'}(sk_i)=F^{m_i'-m_i}(H^{m_i}(sk_i))$来伪造任何满足$\forall i(m_i\le m_i')$的签名$m'$，$H$是哈希函数。校验和解决了这个问题：$m_i$中的任何更改都会导致校验和的变化。因此，对手不能伪造(m',C')（C是校验和)。

Size-optimal 编码[26]。在 WOTS+中，编码函数将校验和附加到原始消息的末尾，消息大小固定为l，并且尽可能构造较小的编码长度。然而，WOTS+编码方法存在一个问题，即低消息空间利用率，因为校验和占用了一部分。为了最大化消息空间的利用，平衡 WOTS+首先将编码的大小固定为l，为了容纳尽可能大的消息空间，这种编码方法确保编码过程中所有码字的总和为定值，从而消除了对校验和的需要。为了实现平衡 WOTS+，令

$$D_{n,m}=|\{v\in[w]^n\}:\sum_{i=1}^{n}v_i=m|, \tag{1}$$

其中$D_{n,m}$表示前n个码字的总和为m的大小，定义初始值为

$$\begin{aligned}&D_{1,m}=1, m\in\{0,1,\cdots,w-1\}\\&D_{n,m}=0, 2\le n\in\mathbb{Z}, m\in\mathbb{Z}^{-}\end{aligned} \tag{2}$$

和递归关系

$$D_{n,m}=\sum_{i=0}^{w-1}D_{n-1,m-i}, 2\le n\in\mathbb{Z}, m\in\{0,1,\cdots,l(w-1)\}. \tag{3}$$

接下来解释一下公式(3)的递归关系。要计算$D_{n,m}$，首先考虑最后一个求和的值，可以是$\{0,1,\cdots,w-1\}$中的任意值。假设将该值设为i，则前$n-1$个元素的和必须为$m-i$。因此，可以将问题“n个元素的和为m”变成“$n-1$个元素的和为$m-i$”。通过这种推导，我们可以简单地累加$D_{n-1,m-i}$来计算$D_{n,m}$。根据这种方法，具体的平衡 WOTS+签名算法如下：

算法 1：bw_sig

输入：l, w

输出：SIG_{bw}

1. 设d是大小为l的数组;

2. $m = \lfloor l(w-1)/2 \rfloor$;
3. for $i = l$ to 1 do
4. for $j = 0$ to $\min\{w-1, m\}$ do
5. if $x \geq D_{i-1,s-j}$ then
6. $x = x - D_{i-1,s-j}$;
7. else
8. $d[i] = j$;
9. break;
10. endif
11. endfor
12. $m = m - d[i]$;
13. endfor
14. $SIG_{bw}[i] = F^{d[i]}(sk_i)$;
15. return SIG_{bw}

其中，私钥 sk 由 l 个随机数据块组成，即 $sk = (sk_1, sk_2, \cdots, sk_l), sk_i \in \{0,1\}^{\lambda}$。公钥 $\{pk_i\}_{i=1}^{l}$ 是通过对私钥中的每个块迭代应用哈希函数 $w-1$ 次而生成的，即 $pk_i = F^{w-1}(sk_i)$。最后，通过可调整的哈希函数 Th_l（详见 2.2 节）将这些块合并为公钥 pk，即 $pk = Th_l(pk_1, pk_2, \cdots, pk_l)$。对于包含 l 个块的签名 SIG_{bw}，验证者将哈希函数应用于每个块 $w-1-d[i]$ 次，即 $pk_i' = F^{w-1-d[i]}(SIG_{bw}[i])$。最后，再次使用 Th_l 将这些块合并为 pk'，即 $pk' = Th_l(pk_1', pk_2', \cdots, pk_l')$。如果 $pk = pk'$，验证者接受此签名。

2.4 超树

Merkle 树。 为了对 $2^{h'}$ 条消息进行签名，签名者生成 $2^{h'}$ 个 WOTS+密钥对，并将这 $2^{h'}$ 个公钥作为叶子节点构建二叉树。具体而言，对于每对叶子节点，签名者重复应用哈希函数生成对应的父节点，直到到达树的根节点，即这个树的公钥。在进行签名时，签名者选择一个 WOTS+叶子节点，并发布 WOTS+签名以及从叶子节点到根节点的路径上所有节点的兄弟节点，这就构成了“认证路径”（详见图 1）。验证者首先从签名中获得 WOTS+公钥，然后使用认证路径上的节点及其兄弟节点来重建根节点。

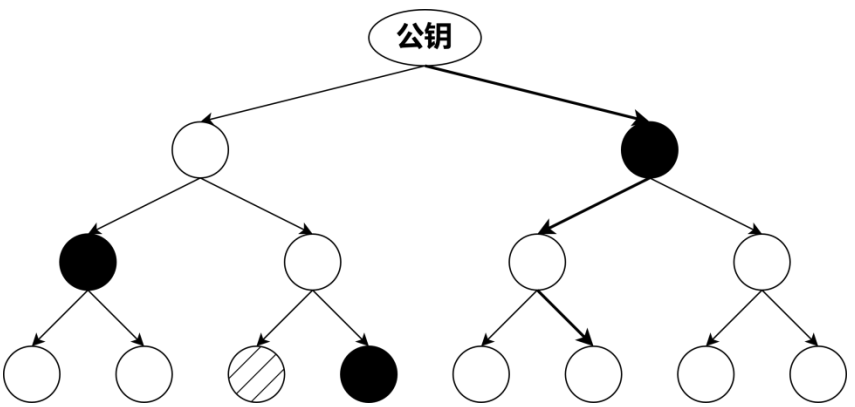

图 1 黑色表示第三个叶子节点的认证路径

请注意，为了生成能够验证 $2^{h'}$ 条消息的根节点，需要对所有叶子节点进行哈希。如果想要在单个 Merkle 树中签名所有消息，这在实际中不可行。因此，接下来使用超树来解决这个问题，如图 2 所示。超树[27]可以用作认证树，由 d 层组成，底层的叶子节点专门用于签名消息(在本文中，签名消息是 DFORC 公钥)。另一方

面，更高层的叶子节点用来对子树的根节点进行签名。从超树最底层的叶子节点到最顶部的根节点，WOTS+签名和认证路径构成了一条完整的认证路径。重要的是，每个中间树的所有叶子节点都是确定生成的 WOTS+公钥，不依赖于下面的任何树。这意味着超树实际上是虚拟存在的，在计算过程中无需生成完整的超树。此外，在密钥生成过程中，只计算最顶部的子树来生成公钥。定义超树的高度为h，中间层数为 d，$h' = h / d$。

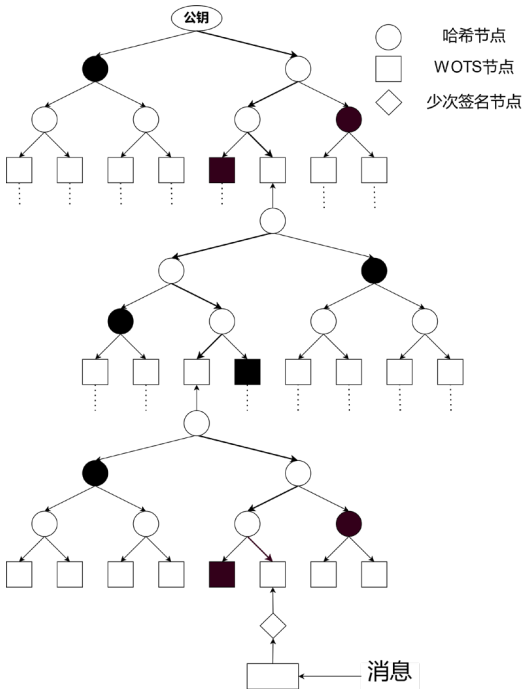

图 2 超树结构： $h = 9, h' = 3, d = 3$

2.5 随机森林链

随机森林链（Forest of Random Chains, FORC）是少次签名的一种变体。Winternitz 方法可以利用哈希函数将多位消息编码到单个块中。基于这一观点，本文选择将哈希链合并到 FORS 签名[27]的叶子节点中。w 表示 Winternitz 参数，表示哈希链的长度。图 3 所示的结构是由 k, a 和 w' 参数构成的。

密钥生成。公钥由k棵树的根节点通过哈希函数 Th_k 生成，每棵树的叶子节点上挂着 $t = 2^a$ 条链。首先随机生成 t 个块，然后对每个块迭代应用 w' 次哈希函数 F 以构建根节点，其中 F 详见 2.2 节。最后，将端点块视为叶子节点并逐步哈希到 Merkle 树的根节点。

签名和验证。给定 $k(a+\log w')$ 位的消息，将其分成 k 个块，每个块由 $(a+\log w')$ 位组成。每个块表示叶子节点的索引及其在 k 个 Merkle 树中的位置。签名由所选节点及其各自的认证路径（详见图 1）组成。验证者使用认证路径重建每个根节点，然后使用 Th_k 重建公钥。

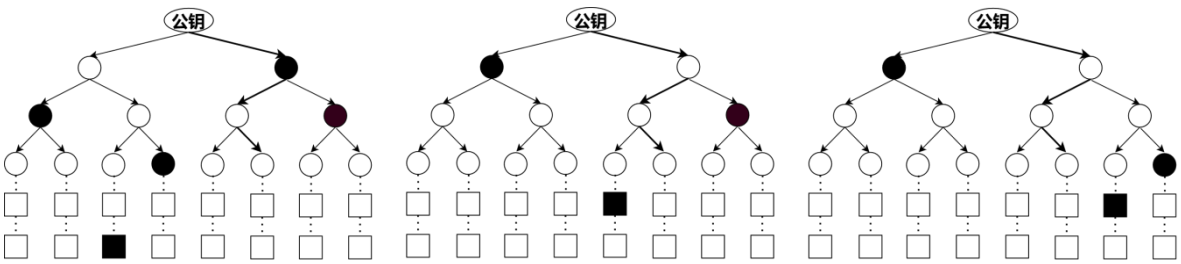

图 3 对于消息 010 10 110 01 100 11，k=3，a=3 和 $w' = 2$ 的 FORC 签名的说明

3 EQAS 方案概述

本节从 3 方面介绍 EQAS，分别是系统模型、工作流程和威胁模型。在系统模型部分介绍轻节点和全节点两个实体的功能；在工作流程部分通过实例来介绍认证存储的全过程；在威胁模型部分定义了每个实体的行为和攻击手段。为了便于阅读，在表 1 中给出相关符号的含义。

表 1 符号表

符号	含义
n	以字节为单位的安全参数
w	Winternitz 参数
h	超树的高度
d	超树的层数
k	DFORC 中树的数量
t	DFORC 树的叶子节点数
w'	DFORC 哈希链节点数

3.1 系统模型

EQAS 的系统架构主要由两实体组成：轻节点和全节点。

（1）轻节点：轻节点负责验证区块链上交易和状态的正确性，只存储区块头，极大减少了存储空间和计算资源的需求。为了验证数据的真实性，轻节点向全节点请求交易证明和状态承诺，并独立验证返回的结果。这种验证过程减少了对轻节点的存储负担，使其能有效参与系统，同时也提高了系统的可扩展性。

（2）全节点：全节点存储区块链的完整数据，处理所有交易的验证与记录，负责生成和维护最新的区块链状态，并为轻节点提供数据验证服务。当轻节点请求某笔交易或某个区块头的状态证明时，全节点返回相应的证明数据。全节点通过生成交易证明、Merkle 根哈希等，保证数据的完整性和一致性。

在这些实体之间，轻节点发送请求后，全节点生成证明并发送回轻节点。轻节点随后独立验证该证明并更新其本地状态信息。

3.2 工作流程

在区块链完整的认证存储工作流程中，用户首先发起并签名交易，然后将交易广播至网络。全节点接收交易并进行验证，随后将有效交易临时存储在内存中的交易池。接着，全节点会为新区块创建打包这些交易，生成区块哈希，并通过共识机制提交新区块。在新区块被网络中其他全节点验证和接受后，全节点将区块中的交易数据及其状态变更写入区块链数据库，并生成当前账本状态的承诺（如 Merkle 根哈希），以确保数据的不可篡改性和可验证性。

此时，轻节点将请求验证交易或状态的证明。具体交互过程如下：

1、轻节点发出请求：轻节点不存储完整的区块链数据，当需要验证某笔交易或状态时，它会向全节点发送数据请求（如某一交易的存在性证明或某一状态的最新承诺），包括交易的标识符或状态的哈希。

2、全节点生成证明：全节点接收到请求后，会根据轻节点的请求，生成相应的交易证明或状态承诺证明。具体过程包括从区块链中检索到对应的交易或状态数据，生成 Merkle 树的验证路径，并提供最新的区块头信息作为证明。

3、全节点返回证明：全节点将生成的证明数据返回给轻节点，这些证明包括：交易的存在性证明，如 Merkle 树的验证路径，轻节点可以通过此路径验证交易是否确实存在于某个区块中；状态的承诺证明：全节点返回最新的状态承诺信息，以证明该状态自上次提交以来没有被篡改。

4、轻节点验证证明：轻节点接收到证明后，首先使用全节点返回的最新区块头信息来验证交易或状态的完整性。通过验证 Merkle 路径或状态承诺，轻节点可以独立确定所请求的数据是否真实可靠，并更新其本地的区块链状态。

5、验证结果：轻节点在验证证明后，如果验证成功，则更新其本地状态，并将其视为有效交易或状态。如果验证失败，轻节点将丢弃返回的结果，并可能尝试从其他全节点获取正确的证明数据。

通过这种交互机制，轻节点可以在不下载整个区块链数据的情况下，确保交易和状态的真实性和完整性。

3.3 威胁模型

在抗量子区块链认证存储系统的威胁模型构建中，我们假设轻节点为诚实验证者，并特别考虑了全节点

作为潜在的攻击者。这一假设基于以下几个方面的考虑：

1. 轻节点与全节点在资源能力上的差异：轻节点通常是资源受限的设备，它们不存储完整的区块链数据，仅保存区块头信息，并依赖全节点提供交易证明和状态承诺。由于轻节点的主要任务是验证由全节点提供的证明数据，且不具备篡改交易或生成伪造数据的能力，它们的唯一目标是确保收到的证明数据的真实性和完整性，因此我们将其设定为诚实的验证者。

2. 全节点在网络中的关键角色：全节点存储了整个区块链的交易数据和状态信息，并承担着生成和维护这些信息的责任。在此过程中，全节点不仅可以影响账本的状态，还负责为轻节点提供数据服务。这种角色使全节点有可能成为攻击的焦点。例如，一个不诚实的全节点可能通过量子攻击手段伪造交易证明、篡改状态数据或向轻节点返回错误信息，试图破坏系统的完整性和安全性。因此，全节点在威胁模型中被设定为潜在的攻击者。

由于全节点具备量子计算能力，能够对区块链认证存储构成直接威胁。该量子敌手可以利用量子攻击算法来破坏区块链系统的安全性。基于以上分析，全节点可能采用以下攻击手段：

1、Shor 算法[8][9]：该算法能够高效分解大整数，从而破解 RSA 等基于大数分解问题的公钥密码体系。攻击者可以利用这一算法来伪造全节点生成的数字签名，进而插入恶意数据，轻节点将难以察觉篡改行为。

2、Grover 算法[17]：Grover 算法可以加速搜索哈希碰撞，削弱基于哈希的认证存储机制的安全性。这种量子加速的攻击手段可能导致哈希承诺机制失效，进而影响轻节点对全节点返回数据的验证结果。

3、数据伪造和篡改：作为攻击者，不可信的全节点可以通过伪造签名，进而伪造认证信息或插入恶意数据，使系统无法检测到数据被篡改，最终破坏区块链系统的完整性和真实性。

结合文献[15]的安全定义，我们形式化定义本文所提出的抗量子安全，如下所示：

定义 1 (PQ-EU-CMA) 设 $SIG=(kg,sign,vf)$ 为数字签名方案。本文定义量子对手 $\mathcal{A}$ 的成功概率为：

$$Succ_{SIGN}^{EU-CMA}(\mathcal{A})=\Pr[vf(pk,M^{*},\sigma^{*})=1\wedge M^{*}\notin\{M_i\}_{i=1}^{q_s}\mid(sk,pk)\leftarrow kg();(M^{*},\sigma^{*})\leftarrow\mathcal{A}^{sign(sk;)}], \tag{4}$$

其中 $M_1,\cdots M_{q_s}$ 是 $\mathcal{A}$ 提交给签名 Oracle 的消息。本文将 SIGN 的 PQ-EU-CMA 不安全性定义为：计算时间为 ξ，查询复杂度为 q_s 的所有量子对手的最大成功概率为

$$In\,\mathrm{Sec}^{PQ-EU-CMA}(SIGN;\xi;q_s)=\max_{\mathcal{A}}\{Succ_{SIGN}^{EU-CMA}(\mathcal{A})\}. \tag{5}$$

4 抗量子的高效区块链认证存储方案 EQAS

在本节中，EQAS 利用键值对数据库作为后端，并维护一个键值映射元组 (KV,VM,ID)，其中 KV 存储键值对，VM 存储键的版本号，ID 存储键对应的索引。当区块链处理一个区块时，该区块包含键值对 $(key,value)$，其中每个键值对都具有一个版本号，起始值为 0。接下来，首先介绍 FORC 的变体 DFORC，然后介绍 EQAS 的设计细节。EQAS 框架如图 4 所示。

4.1 动态随机森林链（DFORC）

典型的 FORC 结构是静态的。一旦 FORC 构造完成，私钥在未达到签名限制前不会进行更新，FORC 结构也不会被修改。此外，由于 FORC 结构依赖于私钥的构建，若私钥保持不变，则该结构也将始终保持不变。然而，在公有区块链中，这种静态 FORC 结构无法用于生成承诺，因为承诺会随着键值对的更新而改变。因此，如何动态地产生 FORC 承诺成为一个挑战。为了解决这个问题，本文提出了 DFORC(动态随机森林链)。在 DFORC 中，我们只更改树上的节点，而哈希链上存储的私钥保持不变。当新键值对选择同一棵树上相同索引时，我们需要处理与该索引对应的叶子节点，然后对叶子节点执行哈希操作，哈希迭代的次数取决于选择该索引的次数。然后更新从该叶子节点到根节点路径上的节点，逐步生成一个新的根节点，即承诺。

在传统区块链认证存储中，MPT 是一种常用的数据结构，用于生成交易的存在性证明和状态承诺。然而，MPT 结构的一个关键瓶颈在于每次状态更新时，不仅需要更新涉及的路径节点的哈希值，还需要进行大量的磁盘 I/O 操作。这是因为 MPT 的数据存储通常会涉及到磁盘读写，而读取路径上各个节点进行更新会导致 I/O 操作的频繁发生。本文提出的动态随机森林链（DFORC）依然需要更新路径上的节点，但与传统的 MPT 结构不同的是，DFORC 在设计上避免了额外的磁盘读写操作。DFORC 将状态存储结构优化为多条独立的哈希链，这些哈希链构成一个随机森林结构，允许每次状态更新仅限于相关的哈希链，而不涉及其他无关链上的节点。这种设计虽然仍需对哈希路径上的节点进行更新，但通过缓存机制或内存中的节点管理，DFORC 大幅减少了对磁盘的频繁读写需求。具体而言，当状态更新发生时，DFORC 只需要对当前状态变更涉及的那条哈

希链进行更新操作，而无需频繁读取和写入磁盘。通过避免大量无关节点的读写操作，DFORC 有效地降低了磁盘 I/O 瓶颈问题。这不仅显著提高了更新效率，还减轻了全节点的存储压力。

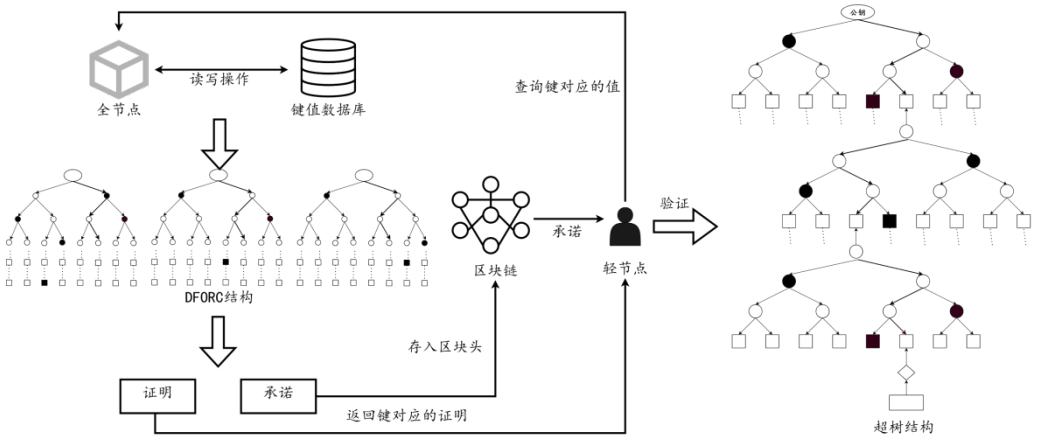

图 4 EQAS 框架

4.2 初始化

首先计算随机生成器 R，消息摘要 md' 及其在 DFORC 中的相应索引，以便稍后用于生成承诺和证明，具体过程参见算法 2。生成 EQAS 的 $SK.seed$、$PK.seed$、$SK.prf$ 需要调用三次随机数生成器(算法 2 第 1 行)。然后，基于 key 和 $SK.prf$ 生成 R，定义为：

$$R = PRF(SK.prf, Optrand.key), \tag{6}$$

其中，$Optrand$ 是 n 字节字符串，初始化为 0，可以选择性地用 1 随机覆盖来进行设置。通过添加额外的随机性，R 具有非确定性，这有助于抵御侧信道攻击。然后，我们推导出使用的索引 idx，以及 md：

$$(md \,\|\, idx) = H_{msg}(R, PK.seed, PK.root, key), \tag{7}$$

其中，H_{msg} 详见 2.2 节，$PK.root$ 来源于超树的顶层根节点（见算法 7 中 1 行至 14 行），md 的长度为 $m = \lfloor (ka+7) \rfloor + \lfloor (h - h/d + 7) \rfloor + \lfloor (h/d + 7) \rfloor bits$。根据选择的参数（算法 2 的 5 行至 7 行）将 $(md \,\|\, idx)$ 拆分为 md'、树地址 idx_tree 和叶子索引 idx_leaf，其中需要 m_1 字节用于 key 的消息摘要 md'，需要 m_2 字节用于树地址和 m_3 字节用于叶子节点索引，使 $m = m_1 + m_2 + m_3$。接着将 md' 拆分成 k 个 $\log(t)$ 位的字符来生成索引数组 $idx[k]$，其中 k 是 DFORC 树的数量，t 是叶子节点数量。例如 $idx[k] = [2,5,7,\ldots]$，这 k 个索引中的每个都对应 DFORC 中每个 sk。具体来说，第一个索引选择第一个树中的第 2 个 sk；第二个选择第二棵树中的第 5 个 sk；以此类推，最后一个索引选择最后一棵树中的第 $idx[k-1]$ 个 sk。随后，将与 key 对应的索引添加到 ID 中。

算法 2：初始化

输入：key

输出：md', idx_tree, idx_leaf

1. $SK.seed, SK.prf, PK.seed \leftarrow Random(n)$;
2. $PK.root \leftarrow ht\_sig()$（见算法 7 中 1 行至 14 行）；
3. 根据公式（6）生成 R；
4. 根据公式（7）生成 $(md \,\|\, idx)$;
5. md' 为 $(md \,\|\, idx)$ 中前 $\lfloor ka+7 \rfloor$ bits;
6. idx_tree 为 $(md \,\|\, idx)$ 中接下来的 $\lfloor (h - h/d + 7) \rfloor$ bits;
7. idx_leaf 为 $(md \,\|\, idx)$ 后面的 $\lfloor h/d + 7 \rfloor$ bits;

8. return md',idx_tree,idx_leaf

4.3 承诺和证明

4.3.1 *ADRS*方案

*ADRS*方案采用一组操作地址的方法，具体的所有操作方法见文献[32]，所有节点在每次哈希调用后更新地址。对于不同的用例，有五种类型的地址：第一种类型用于平衡 WOTS+ (BW)方案中的哈希函数，另一种用于压缩 BW 公钥，第三种类型用于超树中 Merkle 树的哈希函数，以及另一种用于 DFORC 中每棵树的哈希，最后一种类型用于压缩 DFORC 的根节点。如图 5 所示，layer address 表示节点在超树中的高度，对于底层的树，layer address=0；tree address 表示从最左边的树索引 0 开始，节点在超树中的位置；type 表示地址的类型：BW 哈希地址为 0，BW 公钥压缩为 1，哈希树地址为 2，DFORC 地址为 3，DFORC 树的根节点压缩为 4；hash address 表示键值对的地址；tree height 表示节点在 DFORC 中的高度；tree index 通过公式（8）来进行计算得出。

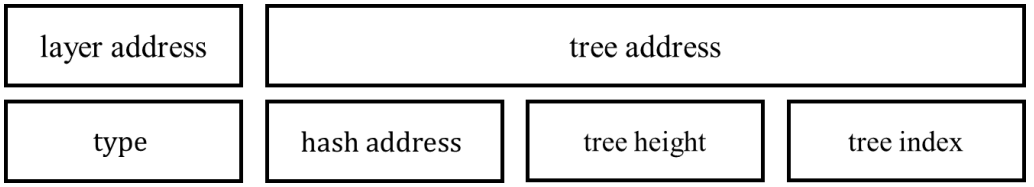

图 5 地址结构

为了便于在本文其余部分提供的伪代码中管理类型，我们分配了特定的常量，以便于引用。这些常量是: $BW\_hash, BW\_pk, DFORC\_tree$ 和 $DFORC\_roots$ 。

4.3.2 生成承诺

$dforc\_Keygen(PK.seed, SK.seed, ADRS, idx, s) \rightarrow dforc_{sk}, dforc_{pk}$:该算法用来生成 DFORC 的公私钥对，以公私钥种子 $PK.seed, SK.seed$ 、地址 *ADRS* 、索引 *idx* 和开始索引 *s* 作为输入，结合树的索引和高度，依次生成树中的节点。具体来说，首先根据 $ADRS.setTreeHeight()$ 和 $ADRS.setTreeIdx()$ 来生成树的索引和高度，然后通过公式(9)生成私钥，随后通过对每一棵树进行遍历，逐层生成节点，过程中利用了 *ADRS* 方案来控制树的索引和高度，通过循环的方式生成树的每一层节点，最终合成根节点并返回密钥。完整的过程见算法 3，其中 Th_k 和 *H* 的说明见 2.2 节，有关 *ADRS* 的操作参考文献[32]。

ADRS 中 $tree\_index$ 部分计算为:

$$tree\_index = i * t + idx[i]. \tag{8}$$

私钥利用 $SK.seed$ 和 *ADRS* 通过 *PRF* 来生成：

$$dforc_{sk} = PRF(SK.seed, ADRS). \tag{9}$$

然后，我们迭代地将 *F* 应用于每个私钥 *w'* 次，将每次哈希的结果添加到 $leaf[q][w'+1] = [F^2, F^1, sk[q]]$，表示第 *q* 个叶子节点上的哈希链上节点，其中 *F* 见 2.2 节的说明。每个哈希值作为哈希链上节点，最后哈希的节点作为叶子节点，接着从叶子节点开始，根据算法 3 中的 3 行至 18 行来生成每层的节点，直到根节点。

算法 3： $dforc\_Keygen$

输入： $PK.seed, SK.seed, ADRS, idx\_tree, s$

输出： $dforc_{sk}, dforc_{pk}$

1. $ADRS.setTreeHeight(0); ADRS.setTreeIdx(idx\_tree);$
2. 根据公式（9）生成私钥 $dforc\_sk$;
3. for $i \in [0, k-1]$ do
4. for $q = 0$ to $a-1$ do
5. for $j = 0$ to w' do
6. $leaf[q][j] = F^j(PK.seed, ADRS, sk);$

7. endfor
8. $node[i][q] = leaf[w']$;
9. endfor
10. $ADRS.setTreeHeight(1)$;
11. $ADRS.setTreeIdx(s+i)$;
12. while $(Stack.top.height == node.height)$ do
13. $ADRS.setTreeIdx((ADRS.getTreeidx()-1)/2)$;
14. $node = H(PK.seed, ADRS, (Stack.pop() \| node))$;
15. $ADRS.setTreeHeight = (ADRS.getTreeHeight()+1)$;
16. endwhile
17. $Stack.push(node)$;
18. $root[i] = Stack.pop()$;
19. endfor
20. $ADRS.setType(DFORC\_roots)$;
21. $dforc_{pk} = Th_k(PK.seed, ADRS, root[0] \| \cdots \| root[k-1])$;
22. return $dforc_{sk}, dforc_{pk}$

4.3.3 生成证明

$dforc\_sig(PK.seed, ADRS, SK.seed, idx, md') \to SIG_{dforc}$：全节点通过该算法来生成对应的证明。该算法输入 md'、$SK.seed$、$PK.seed$、$ADRS$ 和 idx，输出 DFORC 签名 SIG_{dforc}，即 $proof$。该 $proof$ 是长度为 $k(\log(t)+1)$ 的字符串数组，其中 k 是 Merkle 树的数量，t 是叶子节点的数量，包含从叶子节点到根节点的认证路径上的节点。每个认证路径的元素由 n 字节的私钥和树中节点组合而成，认证路径见图 1 所示。在实际生成过程中，认证路径的每个节点都使用 $SK.seed$ 和 $ADRS$ 通过伪随机函数 PRF 进行计算。对于每个 key，算法通过遍历 Merkle 树来生成相应的认证路径，并将这些路径与签名组合，输出完整的 $proof$。具体过程见算法 4，有关 $ADRS$ 的操作和部分函数 ($SIG_{XMSS}.getAUTH$) 参考文献[32]。

算法 4：$dforc\_sig$

输入：$PK.seed, SK.seed, ADRS, idx, md'$

输出：SIG_{dforc}

1. for $i = 0$ to $k-1$ do
2. $ADRS.setTreeHeight(0)$;
3. $ADRS.setTreeIdx(i*t + idx[i])$;
4. $SIG_{dforc}[i] = PRF(SK.seed, ADRS)$;
5. for $j = 0$ to $a-1$ do
6. $s = (idx/2^j) \oplus 1$;
7. $AUTH[j] = SIG_{XMSS}.getAUTH(j)$;
8. endfor
9. $SIG_{dforc} = SIG_{dforc}[i] \| AUTH[j]$;
10. endfor
11. return SIG_{dforc}

4.4 超树构造

在本节中，我们详细阐述超树的构造过程。由于超树中的叶子节点都是平衡 WOTS+ (BW)的公钥，所以首先介绍 BW，然后基于 BW 来构造超树。

4.4.1 BW 签名

首先需要调用 $w \times l$ 次 F（见 2.2 节）来生成相应的 bw_{pk}。当计算 $F^w(sk_i)$ 时，私钥的地址在每次迭代都

会发生变化。每次迭代过程中，地址 *ADRS* 和公钥种子 *PK.seed* 被结合使用，用于计算新的哈希值，并且更新后的地址会对应当前节点在哈希链中的位置，并将其用作下一步计算的输入。这样设计确保每个哈希步骤的不可预测性，增强了签名方案的抗量子攻击能力。该过程详细描述于算法 5，通过对哈希和私钥地址的更新，输出哈希链中的每一次迭代的哈希值，i 表示开始索引，有关 *ADRS* 的操作参考文献[32]。

算法 5： *chain*

输入： $PK.seed, ADRS, r, s, i$

输出： r 迭代 s 次后的哈希值

1. if $s = 0$ then
2. return r;
3. endif
4. if $(i + s) > w - 1$ then
5. return $NULL$;
6. endif
7. $tmp = chain(PK.seed, ADRS, r, s-1, i)$;
8. $ADRS.setHashAddr(i + s - 1)$;
9. $tmp = F(PK.seed, ADRS, tmp)$;
10. return tmp

$bw\_Keygen(PK.seed, ADRS, SK.seed) \to (bw_{pk}, bw_{sk})$：全节点利用该算法用于生成 BW 公私钥对，私钥中的每个 n 字节字符串都由 *SK.seed* 和地址 *ADRS* 生成，相应的公钥通过对私钥进行 w 次哈希计算生成。具体而言，结合 *ADRS* 地址对每个私钥进行哈希计算，生成对应的公钥，这个过程中，私钥地址根据哈希链的结构进行更新，根据链函数(算法 5)计算哈希链每次迭代的结果。算法 6 展示了 BW 密钥对的生成步骤，有关 *ADRS* 的操作参考文献[32]。通过 *ADRS.setChainAddr*() 和 *ADRS.setHashAddr*() 来调用对应地址，并对每个私钥执行哈希操作，生成 l 个 BW 密钥对，之后计算 bw_{pk} 如下所示：

$$bw_{pk} = Th_l(PK.seed, ADRS, bw_{pk}[0] \| \cdots \| bw_{pk}[l-1]). \tag{10}$$

算法 6： *bw_Keygen*

输入： $PK.seed, SK.seed, ADRS$

输出： bw_{sk}, bw_{pk}

1. for $i = 0$ to l do
2. $ADRS.setChainAddr(i)$;
3. $ADRS.setHashAddr(0)$;
4. $bw_{sk}[i] = PRF(SK.seed, ADRS)$;
5. endfor
6. for $j = 0$ to l do
7. $ADRS.setChainAddr(j)$;
8. $ADRS.setHashAddr(0)$;
9. $tmp[j] = chain(bw_{sk}[j], 0, w-1, PK.seed, ADRS)$;
10. endfor
11. 根据公式（10）生成 bw_{pk}；
10. return bw_{sk}, bw_{pk}

$bw\_sig(PK.seed, ADRS, SK.seed) \to SIG_{bw}$:根据 Size-optimal 编码（见 2.3 节），我们用 BW 私钥进行签名生成 SIG_{bw}。具体过程见算法 1（详见 2.3 节）。

4.4.2 超树签名

本节将说明 BW 如何与 Merkle 树结合生成超树(HT)，从而产生固定输入长度的签名方案。HT 的公钥位于顶层的根节点，下面将解释计算超树签名的生成过程。

$ht\_sig(PK.seed, SK.seed, ADRS, idx\_tree, idx\_leaf) \rightarrow SIG_{ht}$:轻节点利用该算法用于生成 HT 签名。在 ht_sig 算法中，输入包括 $SK.seed$、$PK.seed$、idx 和 $ADRS$，其中，idx 用于选择对消息签名的超树叶子节点。该算法首先通过设置相关参数生成认证路径 $AUTH[j]$，然后利用哈希函数依次从底部叶子节点向上生成树的节点哈希值，哈希函数 H 见 2.2 节，最后输出签名 SIG_{ht}。注意，idx 是作为两个单独的参数传递的：一个参数处理树索引，而另一个参数表示该树中的叶子索引。具体来说，算法通过 $ADRS.setTreeIdx$ 函数调用树节点的索引地址，并逐层生成每一层的认证路径节点。根据认证路径中的相邻节点（通过$AUTH[j]$获取），递归计算出节点，最终合并得到树根节点 $root$。详细过程见算法 7, 其中 h' 表示 HT 中 Merkle 树的高度，有关 $ADRS$ 和部分函数操作，例如 $SIG_{XMSS}, ADRS.set$ 等函数，具体请参考文献[32]。

算法 7：ht_sig

输入：$PK.seed, SK.seed, ADRS, idx\_tree, idx\_leaf$

输出：$SIG_{ht}, root$

1. $ADRS.setLayerAddr(0)$;
2. $ADRS.setTreeAddr(idx\_tree)$;
3. for $j = 0$ to h' do
4. $AUTH[j] = SIG_{XMSS}.getAUTH(j)$;
5. $ADRS.setTreeHeight(j+1)$;
6. if $(idx / 2^j)\%2 == 0$ do
7. $ADRS.setTreeIdx(ADRS.getTreeIdx() / 2)$;
8. $node[1] = H(PK.seed, ADRS, (node[0] \| AUTH[j]))$;
9. else
10. $ADRS.setTreeIdx(ADRS.getTreeIdx() - 1) / 2)$;
11. $node[1] = H(PK.seed, ADRS, (AUTH[j] \| node[0]))$;
12. endif
13. endfor
14. $root = node[0]$;
15. for $j = 1$ to $d - 1$ do
16. $idx\_leaf = idx\_tree$ 中前 $\lfloor h / d \rfloor$ bits;
17. $idx\_tree = idx\_tree$ 中后 $\lfloor h - (j+1) * (h / d) \rfloor$ bits;
18. $ADRS.setLayerAddr(j)$;
19. $ADRS.setTreeAddr(idx\_tree)$;
20. $ADRS.setType(BW\_hash)$;
21. $ADRS.setKeypairAddr(idx)$;
22. 根据算法 1 生成签名 SIG_{bw};
23. $tmp = SIG_{bw} \| AUTH$； $SIG_{ht}[] = null$;
24. $SIG_{ht} = SIG_{ht} \| tmp$;
25. endfor
26. return $SIG_{ht}, root$

4.5 认证

在本节中，我们验证承诺的有效性和证明的正确性。只有当两者都通过认证时，对应的 *key* 才通过认证。详细的认证过程如下所示。

$proof\_ver(SIG_{dforc}, PK.seed, ADRS, md', dforc_{pk}) \to (True / False)$:算法用于验证由 $dforc\_sig$ 提供的签名，输入为 SIG_{dforc} 、公钥种子 $PK.seed$ 、私钥种子 $SK.seed$ 、地址 $ADRS$ 和消息摘要 md' 。首先，算法通过消息摘要 md' 计算出与消息对应的叶子节点索引，并找到签名中的私钥 $SIG_{dforc}.getsk(i)$ 。接着，通过 F （详见 2.2 节）生成节点，并根据 $SIG_{dforc}.getAUTH(i)$ 提供的认证路径，从叶子节点向上依次计算每层的节点，直至生成树的根节点，即 $dforc_{pk}'$ 。最终，将计算得到的 $dforc_{pk}'$ 与公钥 $dforc_{pk}$ 进行比对，如果两者匹配则返回 $True$ ，否则返回 $False$ ，表示签名无效。该过程在算法 8 中给出，其中 $SIG_{dforc}.getsk(i)$ 和 $SIG_{dforc}.getAUTH(i)$ ，前者返回存储在签名中的第 i 个私钥，后者返回存储在签名中的第 i 个认证路径, 其中 Th_k 和 H 的说明见 2.2 节，有关 $ADRS$ 的操作参考文献[32]。

算法 8： $proof\_verify$

输入： $SIG_{dforc}, PK.seed, ADRS, md', dforc_{pk}$

输出： $True / False$

1. for $i = 0$ to $k-1$ do
2. $idx[i] = \lfloor i * \log(t) \rfloor to \lfloor (i+1) * \log(t) - 1 \rfloor$ bits of md';
3. $dforc_{sk} = SIG_{dforc}.getsk(i)$;
4. $ADRS.setTreeHeight(0)$;
5. $ADRS.setTreeidx(i * t + idx[i])$;
6. $node[0] = F(PK.seed, ADRS, dforc_{sk})$;
7. $auth[i] = SIG_{dforc}.getAUTH(i)$;
8. $ADRS.setTreeIdx(i * t + idx[i])$;
9. for $j = 0$ to $a-1$ do
10. $ADRS.setTreeHeight(j+1)$;
11. if $idx[j]\%2 == 0$ then
12. $ADRS.setTreeIdx(ADRS.getTreeIdx() / 2)$;
13. $node[1] = H(PK.seed, ADRS, (node[0] \| auth[0]))$;
14. else
15. $ADRS.setTreeIdx((ADRS.getTreeIdx() - 1) / 2)$;
16. $node[1] = H(PK.seed, ADRS, (auth[j] \| node[0]))$;
17. endif
18. $node[0] = node[1]$;
19. endfor
20. $root[i] = node[0]$;
21. endfor
22. $dforc.setType(DFORC\_roots)$;
23. $dforc.setKeyPairAddr(ADRS.getKeyPairAddr())$;
24. $dforc_{pk}' = Th_k(PK.seed, dforc_{pk}, ADRS, root[0] \| \cdots \| root[k-1])$;
25. if $dforc_{pk}' == dforc_{pk}$ then
26. return $True$;
27. else
28. return $False$;
29. endif

$com\_ver(SIG\_ht, PK.seed, root, ADRS, idx\_tree, idx\_leaf, md') \to (True / False)$**:** 算法用于验证 HT 的根节点是否与提供的承诺一致。首先，算法通过索引 idx_leaf 找到消息对应的叶子节点，并开始使用 ht_sig 中提供的签名值和认证路径中的哈希值，逐层重构 HT。通过对 $SIG_{ht}.getsig()$ 返回的签名值进行递归哈希计算，算法

逐层构建出树的内部节点，最终得到树的根节点。若计算出的根节点 *root'* 与提供的 *root* 一致，则返回 *True*，表明承诺验证成功；否则，返回 *False*，表示验证失败。具体流程见算法 9，有关 *ADRS* 的操作参考文献[32]。

算法 9： com_ver

输入： $SIG\_ht, PK.seed, root, ADRS, idx\_tree, idx\_leaf, md'$

输出： $True / False$

1. $ADRS.setLayerAddr(0)$;
2. $ADRS.setTreeAddr(idx\_tree)$;
3. for $j = 0$ to $d - 1$ do
4. idx_leaf 为 idx_tree 中前 $\lfloor h / d \rfloor$ bits;
5. idx_tree' 为 idx_tree 中后 $\lfloor h - (j+1) * (h/d) \rfloor$ bits;
6. $tmp = SIG_{ht}.getsig(j)$;
7. $ADRS.setLayerAddr(j)$;
8. 根据算法 7 的 1 行至 14 行生成 $root'$;
9. endfor
10. if $root' == root$ then
11. return $True$;
12. else
13. return $False$;
14. endif

4.6 承诺和证明更新

当有键值对加入时，首先查询该键值对是否存在于元组 *KV* 中。现在我们来讨论两种情况:

情况 1：若不存在，承诺值保持不变，但证明改变。换言之，每当验证新的键值对时，承诺值与前一个相同，但生成的证明不同，因为键值对选择的 DFORC 节点发生了变化。随后，根据算法 1 至 9 来进行验证。

情况 2：如果存在，则修改承诺值和证明，并将来自元组 *VM* 对应的版本号增加 1。根据元组*ID*中对应的索引，对应的叶子节点进行 $VM[k]$ 次哈希，并根据 Merkle 树调整从叶子节点到根节点的路径，生成新的承诺值和证明。接下来，根据算法 1 至 9 对进行重新认证。请注意，当验证证明的正确性时，还需要对 $ID[key]$ 进 $VM[k]$ 次哈希，以获得重建 DFORC 的新叶子节点。

5 安全性分析

在本节中，我们通过以下定理来证明方案的量子安全。需要注意的是，*F* 和 *H* 只是 *Th* 的替代哈希，其中 *F* 对应于消息长度 *n*，*H* 对应于消息长度 $2n$。

定理. 对于方案中的参数 *n*、*w*、*h*、*d*、*m*、*t*、*k*，如果满足以下条件，那么 EQAS 是 PQ-EU-CMA 安全：

- *Th* (也是 *F* 和 *H*)针对不同的调整，具有后量子单函数多目标碰撞抗性。
- *F* 针对不同的调整，具有后量子单函数多目标决策第二原像抗性。
- *PRF* 和 PRF_{msg} 是后量子伪随机函数族。
- H_{msg} 具有后量子交错目标子集抗性。

更具体地说，

$$\begin{aligned} &In\mathrm{Sec}^{PQ-EU-CMA}(EQAS;\xi,q_s) \\ &\le In\mathrm{Sec}^{PQ-PRF}(PRF;\xi,q_1) + In\mathrm{Sec}^{PQ-PRF}(PRF_{msg};\xi,q_s) \\ &+In\mathrm{Sec}^{PQ-ITSR}(H_{msg};\xi,q_s) + In\mathrm{Sec}^{PQ-SM-TCR}(T_h;\xi,q_2) \\ &+3\cdot In\mathrm{Sec}^{PQ-SM-TCR}(F;\xi,q_3) + In\mathrm{Sec}^{PQ-SM-DSPR}(F;\xi,q_3), \end{aligned} \tag{11}$$

其中，上述公式涉及到的 PQ-PRF, PQ-ITSR 和 PQ-SM-TCR 的概念参考文献[15]。首先使用固定签名数量(以及 Oracle 查询数量)的通用攻击来估计上述不安全级别。然后，计算对手的计算复杂度，以获得在 EU-CMA 博弈中获胜的成功概率。

证明. 本节的安全证明借鉴了文献[15]中的证明方法，为了证明 EQAS 方案的 EU-CMA 安全性，我们通过引入五个博弈（GAME.0 到 GAME.4）逐步构建敌手成功的概率界限。每个博弈代表了签名方案的一个不同近似，从而使我们能够对签名方案的安全性进行详细分析。在 GAME.0 中，敌手面临的是一个真实的签名场

景。这里，敌手可以提出查询，得到合法的签名，试图找到漏洞以伪造签名。在 GAME.1 中，我们用随机数替换 *PRF* （伪随机函数）的输出，用于生成 BW 和 DFORC 私钥。我们需要在不让敌手察觉的情况下完成这一替换。具体做法是将私钥生成的每个秘密值随机替换为相同大小的随机值，但保证这些随机值与公钥的计算保持一致，即：公钥根据这些随机值生成，使得签名过程看起来仍然有效。为了使敌手无法察觉，这里的“伪签名”设计仍然保证签名对消息 m 的验证通过，因为我们在使用随机值替换时，确保这些值满足签名验证的结构。敌手无法区分 GAME.1 与 GAME.0，因为伪签名的生成与真实签名生成无异，唯一的不同在于使用了随机替代的密钥。敌手在 GAME.0 和 GAME.1 之间成功概率的差异如下：

$$|Succ_{GAME.1}(\mathcal{A}) - Succ_{GAME.0}(\mathcal{A})| \leq In\mathrm{Sec}^{PQ-PRF}(PRF;\xi,q_1), \tag{12}$$

其中，$InSec^{PQ-PRF}$ 表示 *PRF* 的不安全性度量，ξ 表示敌手在博弈中的查询空间大小，q_1 为敌手的查询次数。

在 GAME.2 中，我们将用于消息哈希的 PRF_{msg} 替换为一个真正的随机函数。在消息 m 通过的 PRF_{msg} 后，得到一个随机值，这个随机值用于生成签名，要确保这个随机值与之前生成的私钥值对应。伪签名的设计需要使得消息哈希结果可验证。具体方法是利用哈希值的随机性生成伪签名，但这些伪签名在结构上与真实签名无异，因此签名验证算法对这些伪签名的验证仍然通过。由于 PRF_{msg} 的输出在 GAME.1 中已被替换为随机数，因此敌手无法通过多次查询来区分这些伪签名，因为它们与真实签名完全一致。敌手在 GAME.1 和 GAME.2 之间的成功概率差异为：

$$|Succ_{GAME.2}(\mathcal{A}) - Succ_{GAME.1}(\mathcal{A})| \leq In\mathrm{Sec}^{PQ-PRF}(PRF_{msg};\xi,q_s), \tag{13}$$

其中，$InSec^{PQ-PRF}$ 表示 PRF_{msg} 的不安全性度量，q_s 为消息查询次数。

在 GAME.3 中，攻击者无法直接找到签名的第二原像，而是试图找到签名结构中的弱点来伪造签名。具体而言，攻击者利用签名 Oracle 查询到的合法签名，尝试推断出签名中哈希函数的结构，并找到一种方式生成一个有效的伪造签名。在这里，我们确保签名对所有使用的密钥都是随机的。这意味着签名的生成不仅依赖于随机密钥，还涉及到随机化的结构，使得敌手无法利用之前的已知信息重现签名过程。伪签名在此步骤中是通过随机化签名的各个部分来完成的，通过引入额外的随机值来扰乱签名结构，使得敌手难以通过已知的签名信息推测签名的生成过程。这个阶段的伪造主要依赖于通过伪造签名本身的结构，使得签名能够在验证中通过。如果敌手能够在此阶段识别出伪签名，则说明我们的密钥随机化过程被破坏。但在理想情况下，伪签名与真实签名无法区分。因此，敌手在 GAME.3 和 GAME.2 之间的成功概率差异为：

$$|Succ_{GAME.3}(\mathcal{A}) - Succ_{GAME.2}(\mathcal{A})| \leq In\mathrm{Sec}^{PQ-ITSR}(H_{msg};\xi,q_s), \tag{14}$$

其中，$InSec^{PQ-ITSR}$ 表示哈希函数 H_{msg} 的抗扰性度量。

在 GAME.4 中，与 GAME.3 不同，攻击者的目标是找到一个与已知签名结构相同的“第二原像”，可以伪造出一个新的消息-签名对，使得该签名对通过验证。通过将哈希函数替换为理想的随机函数 Oracle，并在生成签名过程中引入随机扰动，使得攻击者难以找到第二原像。具体来说，生成伪签名时引入一个随机值r，将该随机值与消息 m 结合生成一个伪秘密值，然后使用伪秘密值通过哈希函数生成签名，使得签名与公钥的关系保持一致。通过确保伪签名在验证过程中能够通过哈希和承诺机制的检验，避免敌手发现伪造的过程。此时的敌手可能会尝试通过攻击树的构建算法来推测私钥，但树结构的抗碰撞性（TCR）和抗冲突性确保了此类攻击难以成功。敌手在 GAME.3 和 GAME.4 之间的成功概率差异：

$$|Succ_{GAME.4}(\mathcal{A}) - Succ_{GAME.3}(\mathcal{A})| \leq In\mathrm{Sec}^{PQ-SM-TCR}(Th;\xi,q_2), \tag{15}$$

其中，$InSec^{PQ-SM-TCR}$ 表示可调整哈希函数 *Th* 的抗碰撞性度量。

最终，敌手在整个签名系统中成功伪造签名的概率可以通过以下式子表达，其中结合了多次抗冲突性和抗第二原像攻击的安全性：

$$Succ_{GAME.4} \leq 3 \cdot In\mathrm{Sec}^{PQ-SM-TCR}(F;\xi,q_3) + In\mathrm{Sec}^{PQ-SM-DSPR}(F;\xi,q_3), \tag{16}$$

这表明在理想的安全环境中，敌手成功伪造签名的概率是被严格限定的。

通过逐步的游戏变换和设计有效的伪签名方案，使得敌手无法分辨我们在游戏中的签名是否来自真实签名过程。EQAS 的安全性是基于用到的上述哈希算法，例如在证明和承诺生成的过程中，该阶段的安全性主要依赖于 Th、F、PRF 和 H_{msg}，验证承诺主要依赖于 F 和 PRF，因此本文方案的安全性是由这些哈希函数来保证的，通过定理的证明我们可知基于这些哈希函数的数字签名方案是具有量子安全的，因此，本文方案可以抵御量子攻击，具有量子安全性。

6 实验分析

在设计 EQAS 方案时，安全性和效率之间的取舍是关键考量。系统的安全性主要依赖于哈希签名方案的安全参数选择，而效率则涉及计算、存储和通信的优化。接下来将解释如何选择合适的参数来进行实验，然后对方案进行理论分析和性能分析。

6.1 参数选择

哈希签名的安全性直接依赖于哈希函数的输出长度n，决定了抗量子攻击的安全等级。然而，选择较大的哈希长度虽然提升了安全性，却不可避免地增加了签名的大小和生成验证的计算成本。为了在保证安全性的同时减少签名长度，我们引入了 Winternitz 参数 w，通过增大 w 值来减少签名中的哈希链数量。这样可以加快签名生成和验证的速度，但也会使签名稍大，导致通信和存储开销的上升。此外，较大的 Winternitz 参数也会在一定程度上降低签名的安全性。因此，在不同的安全需求下调整 w 值和哈希函数的输出长度，权衡了签名的大小与计算复杂度之间的关系。DFORC（动态随机森林链）的设计目的是在不牺牲安全性的前提下，最大化提高系统的存储和认证效率。树的高度 h 与层数 d 直接影响着存储的效率。我们通过设置合理的 h 和 d 值，确保在认证过程中，只有少数的路径节点需要更新，这大幅减少了存储的读写开销。然而，较高的树高度虽然减少了更新所需的哈希链长度，却增加了每次认证时的验证路径长度。为了在减少存储开销和加快验证之间取得平衡，我们通过动态调整超树结构中的层数来确保系统的高效性。此外，DFORC 在设计中还需要平衡认证速度与存储开销之间的取舍。通过降低 Merkle 树的高度，我们减少了每次认证需要遍历的节点数量，从而提高了认证的速度。然而，这也意味着需要更多的存储空间来容纳更多的节点。因此，我们采用了多层结构，每层的树高度适中，既优化了认证速度，又确保存储开销在合理范围内。较低的树高度减少了认证路径上的计算复杂度，但多层设计也增加了实现的复杂性。参数 k 和 t 决定了 DFORC 的性能和安全性。在 DFORC 中，树的叶子节点数量必须是 2 的幂次方，而 k 可以自由选择。较小的t通常会导致更小、更快的签名。然而，对于给定的安全级别，较小的 t 需要较大的 k，这会增加签名大小并减慢签名速度。因此，平衡这两个参数非常重要。消息摘要长度 md 是 H_{msg} 的输出长度，以字节为单位，长度为 $\left(k\log t+7\right)/8+\left(h-h/d+7\right)/8+\left(h/d+7\right)/8$ 字节。BW 密钥对中的哈希链数决定了 BW 签名的大小。

EQAS 的部分参数集设置如表 2 所示（证明大小单位为 bit）。重要的是，本文提出的参数集不是仅仅通过在输出中搜索最小或最快的选项来获得的。这是对于 256 位安全级别，签名的大小大约为 15 KB。然而，单个签名的计算需要 20 多分钟，尽管这种权衡可能对一些特定的应用程序很有吸引力，但许多应用程序不太可能愿意接受如此大量的签名时间。相反，本文建议的参数集认为是“非极端的”，这意味着在我们的实现中，签名时间不超过几秒钟。因此，本文选择 SHA2-128f-simple 算法来实现我们的方案，使用具有 128 位安全性的哈希函数 SHA-256，使用快速参数(更快的计算，更大的签名大小)，具有简单的形式(即在哈希时不使用位掩码)。基于上述设置，我们用 Python 编程语言实现了 EQAS。

表 2 EQAS 的参数设置

n	h	d	logt	k	w	w'	证明大小
128	66	22	7	23	13	2	17088
128	64	8	13	11	32	2	7856
192	66	22	7	37	15	2	35664
192	64	8	12	19	38	2	16224
256	64	16	8	48	16	2	49856
256	63	9	12	27	46	2	29792

6.2 理论分析

本节概述每个操作所需的哈希函数调用次数。因为在估计速度时可以忽略不计，我们省略了对H_{msg}的单个调用和PRF_{msg}用于生成承诺和签名。表 3 总结了对F、H、PRF和Th_k的调用次数。私钥和公钥的大小以及签名和认证路径可以从算法过程推断出来，如表 4 所示。

表 3 F、H、PRF和Th_k的调用次数

	F	H	PRF	Th_k
密钥生成	$2^{h/d}wl+ktw'$	$2^{h/d}-1$	$2^{h/d}l$	$2^{h/d}$
承诺生成	kt	$k(t-1)$	kt	$d(2^{h/d})$
认证路径	$d(2^{h/d})wl$	$d(2^{h/d}-1)$	$d(2^{h/d})l$	————————
认证	$k+dwl$	$k\log t+h$	————————	d

表 4 密钥、签名和认证路径大小

	私钥	公钥	签名	认证路径
存储大小	$4n$	$2n$	$nk(\log t+1)$	$n(h+dl+1)$

6.3 性能分析

6.3.1 实验设置和指标

EQAS 框架用于解决区块链认证存储的安全问题。为了更好地展示 EQAS 的性能，本文进行了实验来展示关键算法的运行时间，所有操作均在 CPU 为 Intel(R) Core(TM) i5-8300H、GPU 为 NVIDIA GeForce GTX 1050、内存为 8GB 的笔记本电脑上进行。操作系统为 64 位 Windows 10, Python 编译器为 Visual Studio Code。EQAS 使用 rocksdb[29]作为后端键值数据库。安全参数为 128 位， $w=16,k=33,t=64,w'=2$ 。此外，本文在国内真实的区块链 Fisco 平台上对提出的 EQAS 方案进行了测试，通过 FISCO BCOS 技术文档将其部署在本地，同时在平台上部署智能合约来进行交易。

在评估 EQAS 的时间开销时，交易总数量 T 、轻节点数量 U 和查询次数 q 是三个关键变量。交易总数量直接影响区块链账本的大小和全节点需要处理的数据量。随着交易数量的增加，全节点必须执行更多的交易验证、账本更新和区块构建工作，这将导致更多的计算和存储操作，从而增加了时间开销。轻节点数量的增加意味着有更多的网络参与者需要从全节点获取数据和证明。每个轻节点的查询请求都需要全节点进行响应，包括数据检索和证明生成，这可能会对全节点造成额外的计算和网络通信负担，尤其在高并发查询的情况下，可能导致响应延迟增加。查询次数是指轻节点向全节点请求数据验证的频率。频繁的查询会使得全节点需要不断地进行数据检索和证明生成，这不仅增加了全节点的计算负载，还可能因为 I/O 操作的增加而影响存储系统的性能。在高查询负载下，认证存储的时间开销会显著增加。

每个认证存储方案的性能是基于生成证明时间和验证时间来评估的。用于比较的认证存储方案如下：

1、SPHINCS+[15]：这是一个无状态的基于哈希的签名框架，它在速度、签名大小和安全性方面相较于传统技术有显著优势。

2、LVMT[4]：这是一种新型的区块链存储结构，旨在通过减少磁盘 I/O 操作的数量来提高区块链系统的性能。LVMT 通过使用多级认证多点评估树和一系列仅附加的 Merkle 树，显著减少了读写操作的放大效应。

3、EQAS：在本文中第 4 节。

6.3.2 实验结果

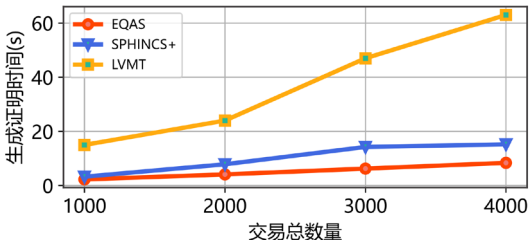

图 6(a) 不同交易总量下证明生成时间

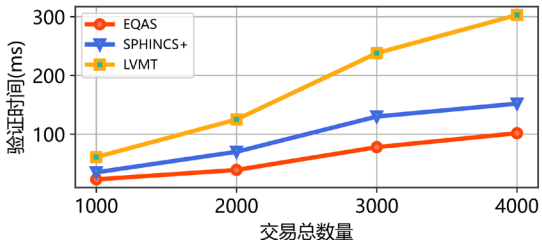

图 6(b) 不同交易总量下验证时间

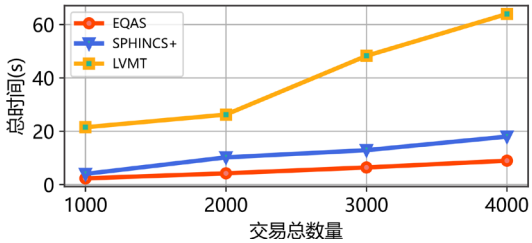

图 6(c) 不同交易总量下总时间开销

图 6 在固定查询次数下，交易总量对认证存储时间开销的影响

图 6 显示了在交易总量固定在 1000 至 4000 的范围内，固定查询次数 $q=100$ 对认证存储每个关键过程的影响。图 6(a)显示，随着交易总量增加，EQAS 的证明生成时间明显优于 SPHINCS+和 LVMT，尤其在交易总量达到 4000 时，LVMT 生成时间接近 60 秒，而 EQAS 保持在较低水平；图 6(b)表明，EQAS 在验证时间上也显著优于其他方案，随着交易总量增加，EQAS 的验证时间增长缓慢，而 SPHINCS+和 LVMT 的验证时间显著上升；图 6(c)则展示了在固定查询次数下，不同交易总量对认证存储时间开销的影响，其中 EQAS 的总时间开销增速最缓慢，进一步证实了其在高交易量场景中的高效性。整体来说，时间开销与交易总量之间存在正相关关系，这是因为随着查询次数的增加，全节点必须处理更多的请求，这可能导致 CPU 和内存的使用率提高，I/O 操作的频率增加，从而增加了整体的时间开销。

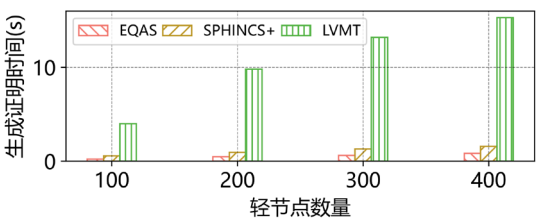

图 7(a) 不同轻节点数量下证明生成时间

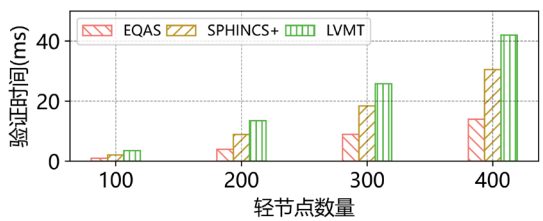

图 7(b) 不同轻节点数量下验证时间

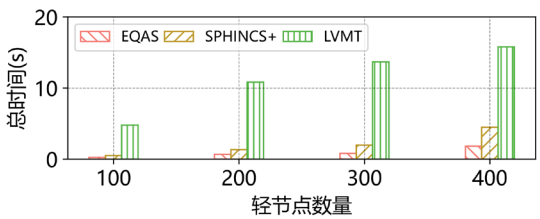

图 7(c) 不同轻节点数量下总时间开销

图(7) 在固定查询次数下，轻节点数量对认证存储时间开销的影响

图 7 展示了在不同轻节点数量 $U=100,200,300,400$ 和固定查询次数 $q=100$ 的情况下，签名生成时间的变化情况。图 7(a)展示了在不同轻节点数量下的证明生成时间，EQAS 在轻节点数量增加时表现出极低的生成时

间，明显优于 SPHINCS+和 LVMT，特别是在轻节点数量达到 400 时，LVMT 的生成时间大幅增加，而 EQAS 几乎不受影响；图 7(b)显示了验证时间的变化，随着轻节点数量的增加，EQAS 的验证时间增长非常缓慢，始终保持在低水平，而 SPHINCS+和 LVMT 的验证时间随着轻节点数量的增加呈显著上升趋势；图 7(c)反映了在固定查询次数下，轻节点数量对总时间开销的影响，EQAS 在轻节点数量增加时总时间开销最小，进一步证实了其在多轻节点场景中的高效性。整体来说，时间开销与轻节点数量之间存在正相关关系，这是因为随着轻节点数量的增加，全节点需要生成更多的证明以响应更多的查询，这增加了计算和网络通信的负载，因此时间开销增加。

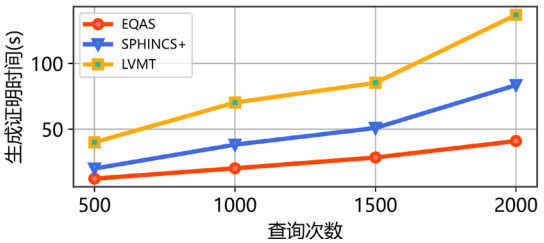

图 8(a) 不同查询数量下证明生成时间

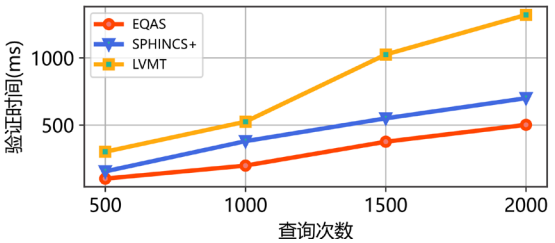

图 8(b) 不同查询数量下验证时间

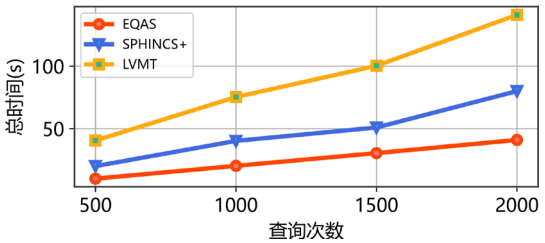

图 8(c) 不同查询数量下总时间开销

图(8) 在固定交易总量下，查询数量对认证存储时间开销的影响

图 8 展示了在不同查询次数 $q=100,200,300,400$ 和不同交易总数量 $T=1000$ 下，对签名生成时间的影响。图 8(a)展示了在不同查询次数下的证明生成时间，EQAS 随着查询次数的增加，生成时间保持在较低水平，显著优于 SPHINCS+和 LVMT；尤其是在查询次数达到 2000 次时，LVMT 的生成时间大幅增加，而 EQAS 的增长趋势相对平缓。图 8(b)显示了验证时间的变化，随着查询次数的增加，EQAS 在验证时间上始终保持领先，呈现出较小的增长幅度，而 SPHINCS+和 LVMT 的验证时间随查询次数的增加呈现更为明显的上升趋势。图 8(c)反映了在固定交易总量下，查询次数对总时间开销的影响，EQAS 在查询次数增加时的总时间开销明显低于其他两种方案，体现了其在高查询次数场景中的高效性。总的来说，随着查询次数的增加，时间开销普遍呈现上升趋势。这是因为随着查询次数的增加，全节点必须处理更多的请求，这增加了计算和 I/O 操作的负载，从而可能导致响应时间延长。此外，在更高的交易总量下，账本数据更大，查询可能需要检索和处理更多的信息，这增加了每次查询的复杂性和所需时间。

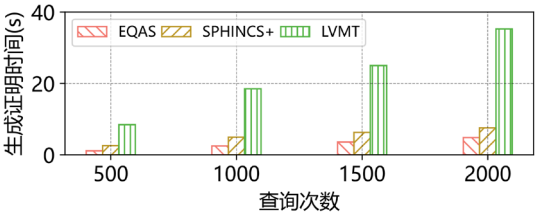

图 9(a) 不同查询数量下证明生成时间

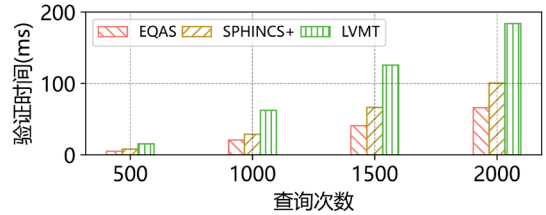

图 9(b) 不同查询数量下验证时间

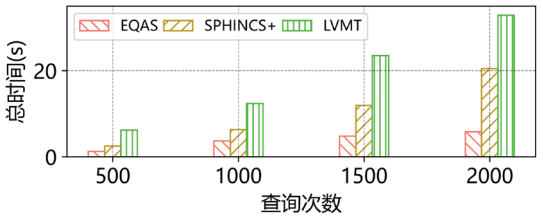

图 9(c) 不同查询数量下总时间开销

图 9 在固定轻节点数量下，查询次数对认证存储时间开销的影响

图 9 分别展示了不同轻节点数量 $U=100,200,300,400$ 下，查询次数 $q=100$ 对签名生成时间的影响。图 9(a)显示了不同查询次数下的证明生成时间，随着查询次数的增加，EQAS 的生成时间保持在较低水平，而 SPHINCS+和 LVMT 的生成时间则随着查询次数的增加呈现出明显的上升趋势，尤其是 LVMT 在查询次数达到 2000 时，生成时间大幅增加，明显高于 EQAS。图 9(b)展示了验证时间的变化，EQAS 在不同查询次数下的验证时间均保持在较低水平，与 SPHINCS+和 LVMT 相比呈现出明显的优势。图 9(c)显示了总时间开销，EQAS 的总时间开销在不同查询次数下增长缓慢，始终低于其他两种方案，显示了其在高查询次数场景中的高效性。随着轻节点数量的增加，每个查询可能需要从全节点获取更多的数据。此外，在高轻节点数量下，全节点可能需要同时处理更多的查询请求，因此时间开销随着查询和轻节点数量成正比。

综上所述，本文提出的 EQAS 方案总的认证存储时间开销平均为 40 秒左右，然而，在以太坊中交易通常需要等待大约 12 个区块才能被确认，每个区块间隔约 15 秒，总计约 180s。由此可见，EQAS 方案的时间开销相对较小。

7 总 结

本文提出了一种高效且抗量子的区块链认证存储方案——EQAS，以应对量子计算对认证存储技术带来的安全威胁。通过将基于无状态哈希签名技术引入区块链认证存储系统，本文有效提高了系统的抗量子能力。具体来说，本文首次将基于无状态哈希签名技术应用于区块链认证存储中，并通过解耦数据存储与认证过程，利用随机森林链和超树结构实现高效的数据认证。安全性分析和实验结果均表明，EQAS 能够抵御量子攻击且具有高效的认证存储过程，为区块链技术提供了一个安全、高效的认证存储解决方案。

未来的研究可以从以下几个方面展开：首先，针对 EQAS 签名长度较大的问题，可以尝试进一步优化基于哈希的签名方案，降低存储和通信成本。其次，随着区块链技术的发展，系统的扩展性和可操作性至关重要，未来可以研究如何在多链甚至跨链环境中应用抗量子认证存储技术，确保跨链交易的安全性。最后，结合分布式计算与隐私保护技术的快速发展，探索如何在保持抗量子安全性的同时，实现更高效的数据共享和验证机制，将是值得深入研究的方向。

References:

[1] Farah M B, Ahmed Y, Mahmoud H, et al. A survey on blockchain technology in the maritime industry: challenges and future perspectives[J]. Future Generation Computer Systems, 2024,157: 618-637. DOI: https://doi.org/10.1016/j.future.2024.03.046.

[2] Chen J, Yang H, He K, et al. Current status and prospects of blockchain expansion technology[J]. Ruan Jian Xue Bao, 2024,35(02):828-851. DOI: 10.13328/j.cnki.jos.006954.

[3] Li C, Beillahi S M, Yang G, et al. LVMT: An efficient authenticated storage for blockchain[C]//17th USENIX Symposium on Operating Systems Design and Implementation. 2023: 135-153.

[4] Li C, Beillahi S M, Yang G, et al. LVMT: An Efficient Authenticated Storage for Blockchain[J]. ACM Transactions on Storage, 2024, 20(3): 1-34. DOI: 10.1145/3664818.

[5] Yang Z, Zolanvari M, Jain R. A survey of important issues in quantum computing and communications[J]. IEEE Communications Surveys & Tutorials, 2023,25(02):1059-1094. DOI: 10.1109/COMST.2023.3254481.

[6] Lu Y, Sigov A, Ratkin L, et al. Quantum computing and industrial information integration: A review[J]. Journal of Industrial Information Integration, 2023: 100511. DOI: 10.1016/J.JII.2023.100511.

[7] Li L, Wu XT, Wu DL. “Nezha” flies into the sky and into the sea, the dawn of quantum computing has appeared[N]. Jiefang Xuebao, 2024-06-14(002). DOI: 10.28410/n.cnki.njfrb.2024.002631.

[8] Monz T, Nigg D, Martinez E A, et al. Realization of a scalable Shor algorithm[J]. Science, 2016, 351(6277): 1068-1070.

[9] Faridi A, Siddiqui F. Improving SPV-based cryptocurrency wallet[C]//Cybernetics, Cognition and Machine Learning Applications: Proceedings of ICCCMLA 2019. Springer Singapore, 2020: 127-137.

[10] Zhao Y, Niu B, Li P, et al. A novel enhanced lightweight node for blockchain[C]//Blockchain and Trustworthy Systems: First International Conference. 2020: 137-149. DOI: https://doi.org/10.1007/978-981-15-2777-7_12.

[11] Zhang C, Xu C, Hu H, et al. COLE: A Column-based Learned Storage for Blockchain Systems[C]//22nd USENIX Conference on File and Storage Technologies. 2024: 329-345.

[12] Truger F, Barzen J, Bechtold M, et al. Warm-starting and quantum computing: A systematic mapping study[J]. ACM Computing Surveys, 2024, 56(9): 1-31. DOI: 10.1145/3652510.

[13] Pillai S E V S, Polimetla K. Analyzing the Impact of Quantum Cryptography on Network Security[C]//2024 International Conference on Integrated Circuits and Communication Systems. 2024: 1-6.

[14] Hülsing A, Butin D, Gazdag S, et al. XMSS: eXtended Merkle signature scheme[R]. Internet Research Task Force. 2018. DOI: 10.17487/RFC8391.

[15] McGrew D, Curcio M, Fluhrer S. RFC 8554: Leighton-Micali hash-based signatures[J]. 2019. DOI: https://doi.org/10.17487/RFC8554.

[16] Bernstein D J, Hülsing A, Kölbl S, et al. The SPHINCS+ signature framework[C]//Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. 2019: 2129-2146. DOI: 10.1145/3319535.3363229.

[17] Grover L K. A fast quantum mechanical algorithm for database search[C]//Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. 1996: 212-219. DOI: 10.1145/237814.237866.

[18] Gao F, Hu R, Yin L, et al. Quantum Grover Search-inspired Global Maximum Power Point Tracking for Photovoltaic Systems under Partial Shading Conditions[J]. IEEE Transactions on Sustainable Energy, 2024, 15(03):1601-1613.

[19] Willsch D, Willsch M, Jin F, et al. Large-scale simulation of Shor’s quantum factoring algorithm[J]. Mathematics, 2023, 11(19): 4222.

[20] Hasanova H, Baek U, Shin M, et al. A survey on blockchain cybersecurity vulnerabilities and possible countermeasures[J]. International Journal of Network Management, 2019, 29(2): 2060. DOI: 10.1002/NEM.2060.

[21] de Ocáriz Borde H S. An overview of trees in blockchain technology: merkle trees and merkle patricia tries[J]. University of Cambridge: Cambridge, UK, 2022.

[22] Raju P, Ponnapalli S, Kaminsky E, et al. mLSM: Making authenticated storage faster in ethereum[C]//10th USENIX Workshop on Hot Topics in Storage and File Systems. 2018.

[23] Choi J A, Beillahi S M, Li P, et al. LMPTs: Eliminating storage bottlenecks for processing blockchain transactions[C]//2022 IEEE International Conference on Blockchain and Cryptocurrency. 2022: 1-9. DOI: 10.1109/ICBC54727.2022.9805484.

[24] Ponnapalli S, Shah A, Banerjee S, et al. RainBlock: Faster transaction processing in public blockchains[C]//2021 USENIX Annual Technical Conference. 2021: 333-347.

[25] Han Y, Li C, Li P, et al. Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems[C]//Proceedings of the 11th ACM Symposium on Cloud Computing. 2020: 238-252. DOI: 10.1145/3419111.3421283.

[26] Zhang K, Cui H, Yu Y. SPHINCS-α: A Compact Stateless Hash-Based Signature Scheme[J]. Cryptology ePrint Archive, 2022.

[27] Bernstein D J, Hopwood D, Hülsing A, et al. SPHINCS: practical stateless hash-based signatures[C]//Annual international conference on the theory and applications of cryptographic techniques. 2015: 368-397.

[28] Li T, Wang H, He D, et al. Designated-verifier aggregate signature scheme with sensitive data privacy protection for permissioned blockchain-assisted IIoT[J]. IEEE Transactions on Information Forensics and Security, 2023, 18:4640-4651. DOI: 10.1109/TIFS.2023.3297327.

[29] Team F D E. RocksDB: A persistent key-value store for flash and RAM storage. 2022.

[30] Karniavoura F, Magoutis K. Decision-making approaches for performance QoS in distributed storage systems: A survey[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(8): 1906-1919. DOI: 10.1109/TPDS.2019.2893940.

[31] Cai ZH, Lin JY, Liu F. Blockchain storage:technologies and challenges[J]. Chinese Journal of Network and Information Security, 2020, 6(5): 11-20.

[32] Jean-Philippe A, Daniel J.B, Warb B, et al. SPHINCS+ Submission to the NIST post-quantum project, v.3. 2020. https://di-mgt.com.au/pqc-08-sphincs-example.html.

附中文参考文献:

[2] 陈晶,杨浩,何琨,等.区块链扩展技术现状与展望[J].软件学报,2024,35(02):828-851. DOI: 10.13328/j.cnki.jos.006954.

[7] 李蕾,吴晓彤,吴丹璐.“哪吒”上天入海，量子计算曙光已现[N].解放日报,2024-06-14(002). DOI: 10.28410/n.cnki.njfrb.2024.002631.

[31] 蔡振华, 林嘉韵, 刘芳. 区块链存储：技术与挑战. 网络与信息安全学报[J], 2020, 6(5): 11-20.