中国科学 : 信息科学 2026 年 第 56 卷 第 9 期: 2417–2435
SCIENTIA SINICA Informationis
论文

《中国科学》杂志社
SCIENCE CHINA PRESS
CrossMark click for updates

基于分层 Merkle 山脉的高效可信数据溯源方法

张学旺[1*], 张俊杰[1], 林金朝[2]

1. 重庆邮电大学计算机科学与技术学院 (示范性软件学院), 重庆 400065
2. 重庆邮电大学光电信息感测与传输技术重庆市重点实验室, 重庆 400065
* 通信作者. E-mail: zhangxw@cqupt.edu.cn

收稿日期: 2025–09–27; 修回日期: 2025–12–02; 接受日期: 2026–02–24; 网络出版日期: 2026–09–02

国家自然科学基金项目 (批准号: U21A20447) 和重庆市研究生创新项目 (批准号: CYS25472) 资助

摘要 可信数据溯源是可信数据空间可信管控的关键环节. 针对区块链溯源场景中数据的高并发写入与验证效率瓶颈, 提出一种基于分层 Merkle 山脉的高效可信数据溯源方法. 该方法利用星际文件系统 (interplanetary file system, IPFS) 存储原始溯源数据, 并在 Merkle 树与 Merkle 山脉的基础上设计 “局部平衡子树 + 全局 Merkle 山脉” 的双层索引结构, 将溯源数据按阈值分段构建, 优化数据插入及完整性证明验证效率. 实验测试了不同数据规模和分段阈值下各个阶段的时间开销. 理论分析及实验结果表明, 所提方法的数据批量插入效率对比传统 Merkle 山脉具有显著提升, 数据批量完整性验证延迟随着数据分布的集中也相比传统 Merkle 山脉表现出更大优势, 在大规模数据场景下具有更高的溯源效率.

关键词 可信数据溯源, 分层 Merkle 山脉, Merkle 树, 区块链, 完整性验证

1 引言

为贯彻国家数据治理与数据要素流通战略, 国家数据局在《可信数据空间发展行动计划 (2024~2028 年)》[1] 中强调要引导和支持可信数据空间发展, 促进数据要素合规高效流通使用, 加快构建以数据为关键要素的数字经济. 行动计划明确将可信管控能力作为核心能力, 要求支持对空间内主体身份、数据资源、产品服务等开展可信认证, 支持对数据流通利用全过程动态管控, 支持实时存证和结果追溯. 在该政策的推动下, 构建既符合安全要求又具备高效写入与验证的可信数据溯源技术体系成为建设可信数据空间的重要方向. 区块链技术凭借其分布式账本、不可篡改和可追溯的特性, 为构建可信的数据溯源体系提供了理想基础, 随着区块链在商业、数字资产、供应链 [2] 等领域的广泛应用, 对数据的动态写入效率、验证实时性与可扩展管理需求日益迫切. Merkle 树作为高效的数据验证结构, 因其抗篡改特性与简洁证明机制, 被广泛应用于区块链 [3]、供应链溯源、数据完整性验证等领域. 其核心机制是通过对底层数据单元进行哈希运算并逐层聚合, 生成唯一的根哈希值, 为构建可信溯源体系提供技术基础. 然而, 传统 Merkle 树在应对高频次、大规模、强动态性的溯源场景时面临严峻挑战.

(1) 动态写入性能瓶颈. 频繁的事件更新导致持续的数据插入, 传统 Merkle 树依赖自底向上的全量重构机制, 频繁的插入引发路径节点哈希的重复计算, 造成显著的写入延迟, 难以满足实时记录需求.

引用格式: 张学旺, 张俊杰, 林金朝. 基于分层 Merkle 山脉的高效可信数据溯源方法. 中国科学: 信息科学, 2026, 56: 2417–2435, doi: 10.1360/SSI-2025-0411
Zhang X W, Zhang J J, Lin J Z. Efficient trusted data traceability method based on a segmented Merkle mountain range. Sci Sin Inform, 2026, 56: 2417–2435, doi: 10.1360/SSI-2025-0411

© 2026《中国科学》杂志社

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

(2) 验证效率随规模劣化. 随着数据规模的增大, 单个数据的溯源树高度剧增, 进行完整性验证时, 需要遍历从叶节点到根节点的长路径, 哈希计算量巨大, 导致验证延迟提高且难以支持高频校验.

(3) 横向扩展能力不足. 单一全局 Merkle 树结构缺乏分段管理机制, 当数据规模膨胀时, 树高度急剧增加, 节点访问局部性恶化, 不仅制约并行计算优化空间, 更为跨系统锚定带来效率瓶颈, 限制系统整体可扩展性.

现有的数据溯源研究沿着多个方向展开, 然而这些方法在应对高并发写入与验证的平衡问题上仍存在不同程度的局限, 本文将这些数据溯源研究归纳为下列四类.

第一类数据溯源研究侧重于通过优化系统架构与业务流程来提升溯源性能. Wu 等 [4] 在区块链供应链溯源中通过分块并行搜索机制提升查询效率, 但其优化集中于查询环节, 并未对高频数据写入带来的延迟提出底层优化策略. Lu 等 [5] 在物联网传感器数据存储中引入区块链安全协议, 重点保障了数据完整性与抗篡改性, 但其工作重心在于安全协议设计, 未对数据动态插入和实时查询所依赖的索引结构性能做出改进. 这类研究从系统层面解决了特定问题, 但未触及由底层数据结构引发的性能瓶颈本源.

第二类数据溯源研究致力于对 Merkle 树结构本身进行改良或寻求替代方案. Escobar 等 [6] 针对 Merkle 树与工作量证明的能耗优化进行了研究, 其目标在于降低区块链整体能耗, 但未解决动态增量写入时频繁重构哈希路径带来的性能瓶颈问题. Mizrahi 等 [7] 提出了事务感知的 Merkle 树结构, 通过调整树节点组织来缩短区块链交易的验证路径, 但其优化基于特定的事务模型, 并未考虑溯源场景中高频数据写入的动态性需求. Shadab 等 [8] 设计的硬件友好型 Bonsai Merkle Tree 算法在 FPGA 平台上实现了高带宽和低延迟, 展现了显著的性能潜力, 但该方案受限于专用硬件平台, 其通用性与可推广性不足. Faraj 等 [9] 则系统回顾了多种加密和验证机制以增强溯源过程的安全性, 但同样忽视了高并发条件下的实际效率问题. 值得注意的是, Xie 等 [10] 提出的平衡 Merkle 树通过维持树结构平衡来优化性能, 然而其本质仍是单一的全局树结构, 任何数据的插入都可能引发全局性的平衡调整, 随着数据规模持续增长, 其动态维护的开销会随之增长.

第三类数据溯源研究引入了 Merkle 山脉 (Merkle mountain range, MMR) 及其变种等更具弹性的数据结构. MMR 因其支持增量构造、追加写入友好的特性被广泛应用于大数据审计与区块链环境. Bünz 等 [11] 在轻客户端协议中、Gudgeon 等 [12] 在二层区块链协议的综述中均肯定了 MMR 的价值. 后续研究如 Perazzo 等 [13] 设计了面向物联网的轻量级客户端. 然而现有基于 MMR 的研究仍普遍关注结构层面的构造简化或特定应用的适配. Bonneau 等 [14] 虽证明了 MMR 在加密累加器更新频率上的最优性, 但对动态更新过程中的路径重构代价考虑不足; Krishnan 等 [15] 将 MMR 用于教育认证生态系统, 但其焦点在于与深度学习模型的结合应用. Liang 等 [16] 提出的分叉 Merkle 山脉通过增加分支因子来降低树高, 但其作为单一的全局结构未能从根本上解决数据规模增长时验证路径延长与更新成本高的问题.

第四类数据溯源研究聚焦于利用联盟链与链下存储的协同设计. 联盟链通过其准入机制和灵活共识, 在保障数据共享安全与审计可靠性方面展现出优势. 典型实践如 Hyperledger Fabric 提供了链下数据处理与链上根哈希锚定的机制. 研究者们在此基础上, 从群组密钥协商 [17]、联邦学习 [18]、同态加密 [19]、属性访问控制 [20]、数据安全共享 [21,22] 以及链下数据索引 [23] 等多个角度进行了深入探索, 旨在降低链上负载、提升扩展性并增强隐私保护. 然而这类研究大多假定链下数据索引是给定的或采用传统结构, 其优化核心在于区块链层与存储层的交互, 并未专门针对链下索引结构本身在高频写入与验证效率上的问题进行优化.

针对上述背景与挑战, 本文提出基于分层 Merkle 山脉的高效可信数据溯源方法 (efficient trusted data traceability method based on segmented Merkle mountain range, ETDT-SMMR), 创新点如下.

(1) 提出分层 Merkle 山脉 (segmented Merkle mountain range, SMMR). 在 Merkle 树与 Merkle 山脉的基础上, 设计了由 “局部平衡 Merkle 子树” (local balanced tree, LBT) 与 “全局 Merkle 山脉” 组成的双层混合索引结构. LBT 层通过分段阈值将全局数据划分为可并行管理的子集, 每个 LBT 段采用平衡二叉树结构优化存储与更新效率; MMR 层采用山峰进位合并机制实现 LBT 根哈希的高效聚合, 降低单次段根追加的时间成本, 在确保抗篡改特性的同时避免全量重构.

(2) 设计具体的 SMMR 算法并开展性能评估. 针对 SMMR 构建、数据增量插入、完整性证明生成与验证等核心操作设计并实现算法. 从高效插入和验证的角度探讨分段阈值的选取, 并在不同段阈值与数据规模下测试

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

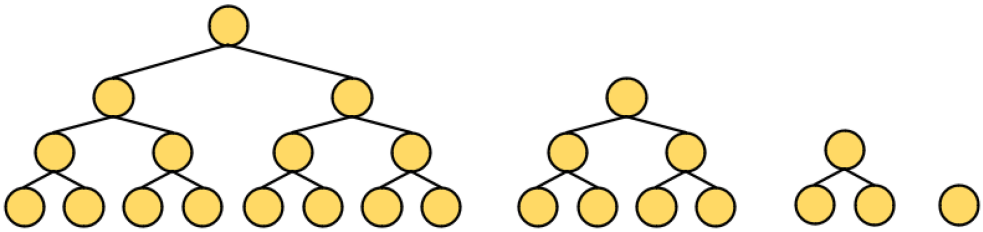


图 1　(网络版彩图) MMR 山脉.

Figure 1　(Color online) Merkle mountain range.

SMMR 的各项性能.

(3) 结合 Hyperledger Fabric 联盟链, 通过链下 SMMR 索引与链上根锚定的解耦设计, 在保障可审计性的同时最小化链上负载. 链下 SMMR 索引周期性地将 MMR 根哈希提交至排序节点, 链上智能合约自动化执行 MMR 根锚定. 数据完整性验证仅需获取链上全局根进行最终比对, 使资源受限的边缘设备可独立完成证明生成和验证.

2　相关技术

2.1　Merkle 树

Merkle 树是一种树状哈希结构, 其核心在于将原始数据分块后, 通过逐层哈希自底向上构建出唯一的根哈希值 [24, 25], 这一机制确保了对数据的任何微小更改都能在对比根哈希时被迅速检测到 [26]. 在供应链与边缘计算环境中, Merkle 树被广泛用于构建安全的数据审计机制与访问控制链路, 有效保障设备交互过程中产生数据的可验证性 [27, 28]. 在保障数据隐私方面, Merkle 树通过仅暴露部分哈希路径即可完成验证操作, 从而避免泄露原始数据内容, 在满足最小可证明原则的同时提升了系统安全性 [29, 30]. 近年来, 该结构被广泛应用于区块链、数据溯源、防篡改存储与可追溯系统中 [31, 32], 凭借其轻量化和可组合性优势, Merkle 树已成为工业级可信溯源架构的基础支撑结构之一.

2.2　Merkle 山脉

Merkle 山脉是由 Peter Todd 提出的一种改良的 Merkle 树结构 [33], 其结构本质上是多个完全二叉 Merkle 树的有序堆叠. Merkle 山脉的核心优势在于其支持高效的数据动态追加, 新元素可以增量式地添加到结构末尾, 而无需重构整个树或改变现有节点, 如图 1 所示. 除此之外, Merkle 山脉保留了 Merkle 树的核心密码学特性, 通过对所有叶子节点数据执行抗碰撞哈希函数进行逐层迭代计算, 最终生成一个唯一的根哈希值, 任何叶子节点数据的修改都会导致其所有祖先路径上的哈希值发生变化. 这种高效的数据插入能力与可靠的完整性验证机制相结合, 使得 Merkle 山脉被广泛应用于数据溯源领域.

3　ETDT-SMMR 方法

3.1　方法描述

ETDT-SMMR 如图 2 所示, 该方法分为存储与查询验证两大核心过程. 存储过程为图中 1~4 (图中实线步骤). 第 1 步将原始数据存储到 IPFS 中并得到对应的内容标识符 (content identifier, CID); 第 2 步将内容标识符与哈希绑定后在当前分段内构建局部平衡子树, 若分段未满则增量插入, 若分段已满则批量重构 LBT 并生成段根哈希; 第 3 步将段根哈希输出至 MMR 山峰列表, 执行山峰合并进位操作, 实时检测并合并高度相同的相邻山峰, 维持山峰高度严格递减; 第 4 步将最新全局根哈希周期性锚定至 Hyperledger Fabric 区块链. 查询验证过程包含图中 5~9 (图中虚线步骤); 第 5 步查询目标数据内容标识符; 第 6 步收集目标数据到段根的路径兄弟哈希

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

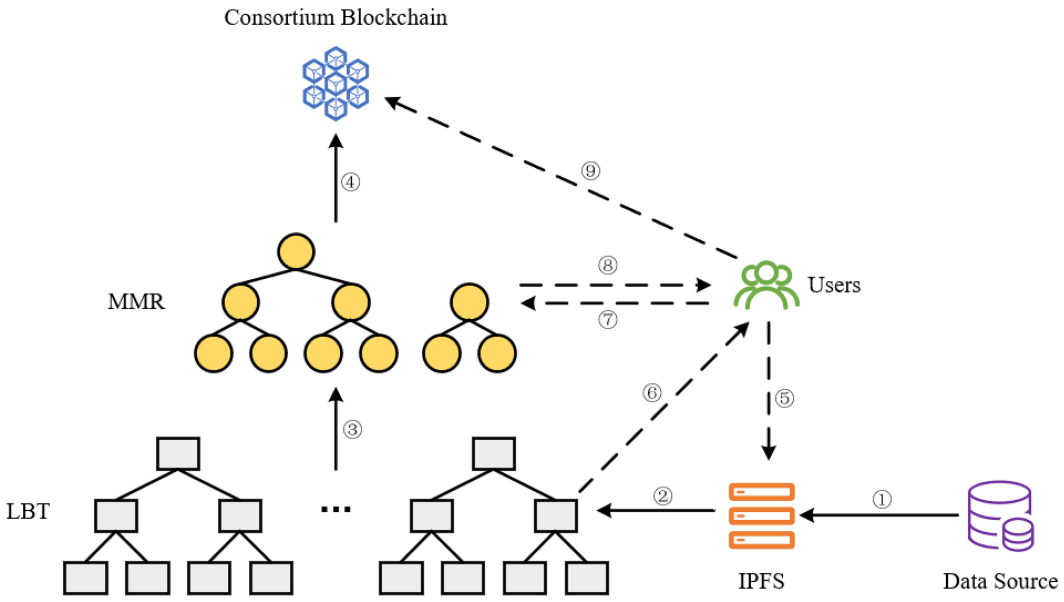


图 2　(网络版彩图) ETDT-SMMR 溯源流程.

Figure 2　(Color online) Provenance process of ETDT-SMMR.

链, 生成 LBT 证明; 第 7 步定位段根在 MMR 中的山峰位置, 收集验证所需兄弟山峰哈希集, 生成 MMR 证明; 第 8 步执行完整性验证, 先利用 LBT 证明重建段根哈希, 再结合 MMR 证明重建全局根哈希; 第 9 步从联盟链智能合约获取最新锚定全局根进行比对, 若一致则返回验证通过, 否则验证失败. 在第 9 步中, 对于刚刚插入还未经历一次完整锚定周期的最新数据, 其关联的根哈希可能还未上链, 对于此类数据的验证请求, 首先将其置为 “待验证” 状态, 负责该批次数据的可信溯源服务节点会将其已生成但还未锚定的全局根哈希附上时间戳并用其私钥进行数字签名, 生成一个具有时间约束力的预锚定证明. 验证方获得预锚定证明后, 可通过验证服务节点的公钥来确认该根哈希的真实性与时效性, 从而完成验证.

分层 Merkle 山脉将平衡 Merkle 树与 Merkle 山脉有机结合, 在分段策略、重构触发、验证流程优化等方面进行了创新设计, 从而提升溯源效率.

(1) 分段策略. SMMR 在每个段内采用缓冲与增量插入, 当段未满时在该段内进行自底向上的哈希更新, 仅在段满时才触发批量重构. 这种局部化的更新策略将高频写入产生的哈希更新局限于段内, 避免每次写入触发对全局结构的大量重算.

(2) 重构触发. 在 SMMR 中, 重构发生时会对段内所有数据进行统一处理并产生新的段根, 重构的开销可视为被段内所有数据均摊, 从而在频繁小更新与偶发批量重构之间取得更好的平衡.

(3) 验证流程优化. 完整性证明由两部分组成 —— 段内的 LBT 路径与 MMR 层的峰列表, 验证按 “局部重建段根 $\rightarrow$ 全局峰证明” 顺序执行. 这一分层验证顺序既减少了重复哈希计算, 又支持段级别的并行验证, 可在多核或分布式环境中并发处理不同段的局部验证, 从而改善实际验证延迟.

不同方法在结构、数据插入和批量完整性验证上的差异如表 1 所示.

3.2　SMMR 构建

SMMR 采用 “局部平衡子树 + 全局山脉” 的双层结构, LBT 层设定阈值 b 作为段容量上限, 其中, b 为 2 的整数次幂 (关于 b 的具体取值将在第 4.1 小节进行讨论), 将数据流分片至 m 个独立段. 当段内数据量 $k < b$ 时, 采用增量插入策略, 新数据按哈希值大小插入 LBT, 通过左旋、右旋等操作维持子树平衡. 当 $k = b$ 时触发批量重构, 全量计算段内哈希并构建满平衡二叉树, 生成唯一段根哈希. LBT 构建的具体算法如算法 1 所示. 在算法 1 中, 步骤 1~8 为叶子节点初始化过程, 将本段的每条原始记录映射成树的叶节点, 准备后续由叶向上构建. 对

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

表 1　不同方法的差异.

Table 1　Differences between different methods.

	MMR	BMT [10]	KMMR [16]	SMMR
Structure	Multiple complete binary trees	Single complete binary tree	Single complete k-ary tree	Complete binary tree + Merkle mountain range
Data insertion	Global	Global	Global	Local
Batch data verification (decentralized)	Independent	Independent	Independent	Sharing the verification path of Merkle mountain range
Batch data verification (centralized)	Sharing partial path	Sharing partial path	Sharing partial path	Sharing paths to a great extent and requiring only one global verification

数组 R 中的每个记录调用一次哈希函数 H 生成哈希值, 每个节点存放记录的哈希值, 左右指针初始化为 null. 步骤 9∼24 为自底向上合并过程, 将当前层 level 按序号两两合并, 父节点的左右指针指向所合并的子节点, 以便后续生成完整性证明和回溯, 如果当前层节点数为奇数, 最后一个节点无法配对, 此时直接将其上浮到 next_level 列表末尾, 保留该节点的哈希不变, 上浮策略保证了所有叶节点都会被包含到最终树中, 且只有当无法完全配对时才产生一个额外节点上浮, 不会引入过多的单侧倾斜, 当 level 数组只剩下一个节点时, 该节点即为段内子树的根, 迭代终止.

算法 1 构建 LBT.

输入: 数据记录数组 $R[1 \ldots b]$, 哈希函数 H.

```
1: level ⇐ empty list;      /*level 存放当前层节点 (初始为叶节点列表)*/
2: for i = 1 to b do
3:     node ⇐ new TreeNode();      /* 为第 i 条记录创建叶节点 */
4:     node.hash ⇐ H(R[i]);
5:     node.left ⇐ null;
6:     node.right ⇐ null;
7:     append(level, node);
8: end for
9: while length(level)>1 do
10:    next_level ⇐ empty list;     /* 当前层节点数 >1 则向上合并生成父层, next_level 存放父层节点 */
11:    for j = 1 to floor(length(level)/2) do
12:        node.left ⇐ level[2j − 1];
13:        node.right ⇐ level[2j];
14:        parent ⇐ new TreeNode();     /* 每次处理一对相邻节点构成父节点 */
15:        parent.hash ⇐ H(node.left.hash‖node.right.hash);
16:        parent.left ⇐ node.left;
17:        parent.right ⇐ node.right;
18:        append(next_level, parent);
19:    end for
20:    if length(level) mod 2 = 1 then
21:        append(next_level, level[last]);     /* 将未配对的最后一个节点上浮 */
22:    end if
23:    level ⇐ next_level;
24: end while
25: r ⇐ level[1].hash;     /* 循环结束时, level 中仅剩节点的哈希即为 LBT 段根哈希 */
```

输出: LBT 子树根哈希 r.

LBT 层构建完成后, 各 LBT 段根节点作为 Merkle 山脉的叶子节点进行山峰合并. 各段根哈希按生成顺序追加至峰列表, 若相邻山峰高度相同, 则合并为高度加 1 的新峰, 并递归检测更高层是否需要合并. 该过程确保

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

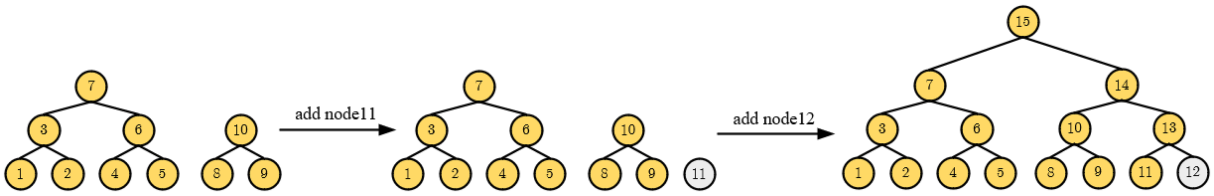


图 3　(网络版彩图) MMR 山峰合并.

Figure 3　(Color online) Merging of MMR peaks.

峰列表高度严格递减, MMR 山峰节点合并过程如图 3 所示. 向 MMR 峰列表添加节点 11 时, 由于列表中不存在相同高度的山峰, 因此不会触发合并. 再向列表中添加节点 12 时会触发以下合并操作. 首先, 节点 11 和 12 的高度相同, 因此合并形成新的根节点 13. 然后, 新形成的山峰与之前根节点为 10 的山峰高度相同, 进一步合并形成新的根节点 14. 接着, 以 7 和 14 为根节点的山峰高度相同, 继续合并得到根节点 15. 此时山峰列表里不存在相同高度的山峰, 合并停止. MMR 主干追加与峰合并的算法如算法 2 所示.

算法 2 MMR 主干追加与峰合并.

输入: 当前山峰列表 P (每项为 $\{h_i, H_i\}$, 其中 h_i 是峰高度, H_i 是峰哈希), 待追加段根哈希 r, 哈希函数 H.

1: cur $\Leftarrow (0, r)$;　/*cur 表示新追加的高度为 0, 哈希为 r 的峰 */
2: append(P, cur);
3: **while** length(P) $\geqslant 2$ **do**
4:　$(h_a, H_a) \Leftarrow P[\text{length}(P)-1]$;　/* 取倒数第 2 个峰 */
5:　$(h_b, H_b) \Leftarrow P[\text{length}(P)]$;　/* 取最后 1 个峰 */
6:　**if** $h_a = h_b$ **then**
7:　　parent.hash $\Leftarrow H(H_a || H_b)$;　/* 当相邻两峰高度相等时进行合并, 计算合并后父峰哈希 */
8:　　parent $\Leftarrow (h_a + 1$, parent.hash);
9:　　pop(P);　/* 弹出倒数第 2 个峰 */
10:　　pop(P);　/* 弹出最后 1 个峰 */
11:　　push(P, parent);
12:　**else**
13:　　break;
14:　**end if**
15: **end while**
16: **if** length(P)=0 **then**
17:　$R \Leftarrow$ null;
18: **else**
19:　$R \Leftarrow P[1]$.hash;
20:　**for** $i = 2$ to length(P) **do**
21:　　$R \Leftarrow H(R || P[i]$.hash);　/* 逐峰折叠计算全局根 */
22:　**end for**
23: **end if**

输出: 更新后的峰列表 P, 全局 MMR 根哈希 R.

在算法 2 中, 步骤 1 和 2 将新段根哈希 r 封装为高度为 0 的峰并追加到峰列表 P 末尾. MMR 结构的每次追加都会引入一座新峰, 其高度从 0 开始, 追加至末尾保证新的峰节点总在最右侧, 符合 MMR 的增量拼接逻辑. 追加前, 峰列表满足高度严格递减; 追加后, 列表末端仍是最低高度. 步骤 3∼15 为合并同高峰的过程, 检查列表末尾两座峰高度是否相同, 若相同, 则通过拼接它们的哈希值并哈希生成更高一级的父峰, 弹出原两峰, 追加父峰, 新父峰可能与它左侧的峰再次高度相同, 算法继续迭代, 直到末尾两峰高度不同或列表长度小于 2. 步骤 16∼23 从最左侧峰的哈希开始, 按顺序将剩余峰依次拼接来计算全局根.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

3.3　数据增量插入

在添加新数据时, 数据首先被追加至当前活动段缓冲队列. 插入操作在段级上采用两种模式: 增量插入与批量重构. 当活动段未满时执行增量插入; 当活动段达到分段阈值 b 时触发一次批量重构并生成确定性的段根哈希.

增量插入时, 若当前活动段尚无局部平衡子树, 则先对段内已有记录构造一棵初始的平衡 Merkle 子树; 否则将新记录的叶哈希直接插入到 LBT 的末端叶位置. 插入时沿插入路径自底向上递归比较并维护哈希次序, 必要时动态执行左旋或右旋操作以维持子树的平衡性; 在完成局部旋转与父链哈希重算后, 返回并更新该段当前的根哈希. 该模式中所有更新均局限于所属段内的节点, 保证了哈希更新的局部性与高频写入下的低开销.

当段缓冲恰积累 b 条记录时, 算法进入批量重构流程. 首先对段内 b 条记录一次性计算叶哈希, 随后调用算法 1 对这些叶哈希进行自底向上的全量重构, 生成新的局部子树及其确定性段根. 重构完成后, 将新的 LBT 持久化到历史段列表, 并清空活动段缓冲.

仅在批量重构完成并产生确定性段根 r 后, 才将该段根提交到 MMR 层. 将 r 作为高度为 0 的新峰追加到峰列表 P, 随后递归检查峰列表末端相邻峰的高度. 若末端两峰高度相同, 则弹出并拼接其哈希并生成高度加一的父峰, 再将新峰压入列表; 重复该合并过程直至末端两峰高度相同. 全部合并完成后同步更新 SMMR 的全局根. 数据增量插入的算法如算法 3 所示.

算法 3 数据增量插入.

输入: 待插入记录 x, 当前段内记录列表 S, LBT 子树结构 T 及其根哈希 r, MMR 主干峰列表 P, 分段阈值 b, 哈希函数 H.

1: append(S, x);
2: **if** length$(S) < b$ **then**
　　/* 若段未满, 采用增量插入方式更新 LBT*/
3:　leaf $\Leftarrow$ new TreeNode();
4:　leaf.hash $\Leftarrow H(x)$;
5:　leaf.left $\Leftarrow$ leaf.right $\Leftarrow$ null;
6:　insert_leaf(T, leaf);
7:　node $\Leftarrow$ leaf.parent;　　/* 从叶节点开始向上更新父节点哈希 */
8:　**while** node $\neq$ null **do**
9:　　node.hash $\Leftarrow H$(node.left.hash$\|$node.right.hash);
10:　　node $\Leftarrow$ node.parent;
11:　**end while**
12:　$r \Leftarrow T$.root.hash;　　/* 更新后的段根哈希赋值给 r*/
13: **else**
　　/* 若段已满, 进行整段批量重构 */
14:　node_list$\Leftarrow [H(y)$ for y in $S]$;　　/* 构建当前段内记录的哈希列表 */
15:　$(T, r) \Leftarrow$ Build_LBT(node_list);　　/* 通过算法 1 重新构建 LBT*/
16:　clear(S);
17:　append$(P, (0, r))$;
18:　**while** length$(P) \geqslant 2 \wedge P$[length(P)].height $=$ P[length$(P) - 1$].height **do**
19:　　$(h_1, H_1) \Leftarrow$ pop(P);
20:　　$(h_2, H_2) \Leftarrow$ pop(P);
21:　　append$(P, (h_1 + 1, H(H_2 \| H_1)))$;
22:　**end while**
23: **end if**
24: $R \Leftarrow$ CalcRoot(P);

输出: 更新后的段内记录列表 S, LBT 子树根哈希 r 以及峰列表 P 和 MMR 全局根哈希 R.

在算法 3 中, 步骤 1∼12 为段内数据增量插入, 将新记录 x 追加到段缓冲 S, 当段缓冲未满时, 在已有局部平衡子树末端插入哈希为 $H(x)$ 的新叶, 然后沿着新增叶的父链, 从父节点到根依次重算哈希从而保证子树完整

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

性, 最后将最新的子树根哈希存入 r. 步骤 13~23 为段满重构与全局峰合并, 当段缓冲恰好积累 b 条记录时, 先一次性计算所有记录的叶哈希, 利用算法 1 对这些叶哈希进行重构, 得到新的子树 T 及根哈希 r, 随后清空段缓冲, 将新的段根 r 作为高度 0 的新峰追加到峰列表 P, 只要峰列表末尾两峰高度相同, 就弹出它们并拼接哈希生成高度加 1 的父峰, 替换原两峰, 重复直到末尾两峰高度各异. 步骤 24 进行 SMMR 的全局根同步.

3.4　SMMR 数据验证

3.4.1　完整性证明生成

在 SMMR 中, 通过分段定位、路径重建与时空快照为任意指定数据构建包含 LBT 段内路径与 MMR 全局状态的完整性证明. 在生成证明的过程中, 首先根据数据内容标识符检索对应分段的原始记录列表, 若为历史段直接读取持久化存储数据, 若为活动段则获取当前缓冲队列, 接着从目标数据对应的叶子节点出发, 沿父节点指针回溯至段根并得到路径兄弟哈希链与方向序列, 以构成段内证明部分. 为消除动态更新导致的峰列表漂移, 系统在每次写入时都会保存一份全局山脉结构中各山峰根哈希的快照, 利用与当前记录对应的写入时刻上的山峰列表就可以直接提取出全局证明部分. 将段内兄弟哈希及方向与山脉峰列表组合起来, 就能得到完整的 SMMR 证明. 生成完整性证明的算法如算法 4 所示. 在算法 4 中, 步骤 1 和 2 定位目标数据的段号与段内偏移, 对全局索引 n 整除和取余计算出它属于第 k 段, 以及在该段中的位置 j. 步骤 3~11 生成段内证明, 在第 k 段的平衡 Merkle 子树 T_k 中找到第 j 个叶节点, 然后从叶节点出发, 沿父链向上遍历, 每到一层就获取该节点的兄弟节点及其左/右位置信息, 记录哈希与位置后追加到段内 Proof 列表 proof_L, 直至到达根. 步骤 12~18 生成 MMR 峰证明, 通过比较哈希值定位段 k 的根哈希在峰列表的索引, 遍历峰列表, 排除位于目标段根的那一项, 将所有其他峰的哈希值按顺序追加到 proof_M.

算法 4 完整性证明生成.

输入: 数据索引 n, 分段阈值 b, 各段的局部平衡 Merkle 子树集合 T_k, MMR 主干峰列表 P, 哈希函数 H.

1: $k \Leftarrow \lfloor n/b \rfloor$;　/* 计算目标记录所属的段编号 */
2: $j \Leftarrow n-(k-1)*b$;　/* 计算段内索引 */
3: $T \Leftarrow T_k$;
4: leaf $\Leftarrow T$.leaf[j];
5: $\text{proof}_L \Leftarrow$ [(leaf.hash, null)];　/* 初始化段内证明列表 */
6: node $\Leftarrow$ leaf;
7: **while** node.parent $\neq$ null **do**
8:　sibling $\Leftarrow$ node.sibling;　/* 取当前节点的兄弟节点 */
9:　proof_L.append((sibling.hash, sibling.position));　/* 记录兄弟哈希及其位置 */
10:　node $\Leftarrow$ node.parent;
11: **end while**
12: idx $\Leftarrow$ find_peak_index(P, $(0, T_k)$);　/* 在峰列表 P 中找到包含该段根的峰索引 */
13: $\text{proof}_M \Leftarrow$ [];
14: **for** $i = 1$ to length(P) **do**
　/* 遍历所有峰构建 MMR 证明 (除包含目标段根的峰)*/
15:　**if** i $\neq$ idx **then**
16:　proof_M.append($H(P[i])$);
17:　**end if**
18: **end for**

输出: 段内 LBT 证明 proof_L, MMR 峰证明 proof_M.

3.4.2　完整性验证

进行完整性验证时, 验证者首先使用接收到的段内证明重建局部子树根哈希, 将其与存储的段根哈希进行比对, 确保目标事数据在段内未被篡改. 如图 4 所示, 假设要对 h2 节点进行验证, 则需收集图中虚线节点的哈希值来计算段哈希根.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

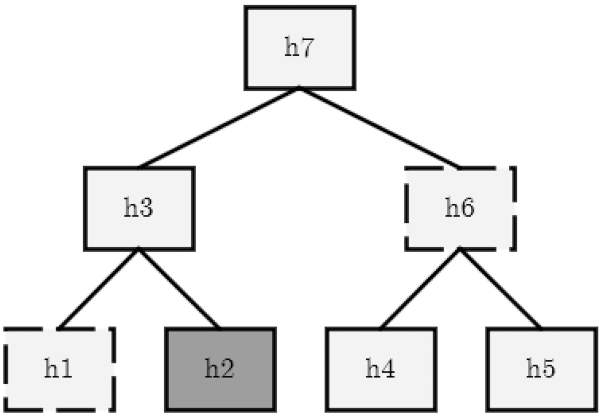


图 4　LBT 层验证.
Figure 4　Verification of LBT.

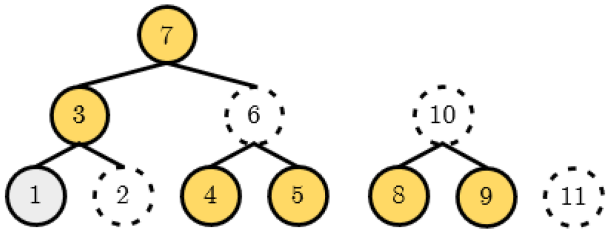


图 5　(网络版彩图) MMR 层验证.
Figure 5　(Color online) Verification of MMR.

LBT 层验证通过后, 将已验证的段根哈希及其 MMR 兄弟峰哈希投入 MMR 验证过程, 通过递归合并或哈希折叠还原全局根哈希, 与链上锚定的全局根进行比较. 当两者完全一致时, 就能确认整条记录在各个分段和全局层面均保持了完整性且未被篡改. 如图 5 所示, h2 节点通过 LBT 层验证后, 其 LBT 层段根节点对应 MMR 层中的 1 号节点, 验证时需依次收集图中虚线节点的哈希值来重构全局根. 完整性验证的算法如算法 5 所示. 在算法 5 中, 步骤 1∼9 为段内根的重构, 从 proof_L 的第 1 项取得待验证记录的原始哈希值 h_0, 然后对 proof_L 中剩余每项 (h_s, pos) 进行判断, 若 $\text{pos} =$ "L", 表示兄弟节点在左侧, 拼接为 $H(h_s \| h)$; 若 $\text{pos} =$ "R", 表示兄弟在右侧, 拼接为 $H(h \| h_s)$, 循环结束后, h 即为重建得到的段内 Merkle 根 r. 步骤 10∼24 为全局根的重构与验证, 首先合并段根与峰证明, 构造一份完整的峰列表 peaks, 按原有 MMR 峰序进行遍历, 在索引等于 sid 时插入重建的段根 r, 否则依序取出 proof_M 的哈希值进行填充, 然后折叠峰列表, 按从左至右的方式进行两两哈希的拼接并最终得到全局根 R, 将得到的全局根 R 与可信根 R_e 进行对比, 若相等则验证成功, 返回 True, 否则验证失败, 返回 False.

3.5　安全性分析

为确保所提 ETDT-SMMR 方法能够为数据溯源提供可信保障, 本节从密码学基础与系统架构层面对方法抵御常见攻击的能力进行形式化分析. 分析基于以下安全假设. 所采用的密码学哈希函数具有抗碰撞性 (本文选取的哈希函数为 SHA-256) 且底层联盟链是不可篡改的.

考虑一个具有如下能力的敌手. 能够截获并篡改在 IPFS 网络中存储的原始数据或在传输过程中的验证信息; 能够尝试伪造或篡改 SMMR 结构中的节点哈希值; 可能发起重放攻击, 即使用旧版本的数据和证明来冒充最新状态. 针对上述威胁, ETDT-SMMR 方法通过其内在机制提供了多层次的安全保障.

首先, 针对数据篡改攻击, 方法将原始数据的 IPFS 内容标识符 (CID) 作为 SMMR 叶节点的输入, 由于 CID 是数据内容的密码学哈希, 任何对原始数据的修改都会导致 CID 变化, 从而在后续完整性验证中无法重构出与链上锚定根哈希相匹配的 Merkle 路径, 使得任何篡改行为均可被检测.

其次, 对于索引结构篡改攻击, 即敌手试图伪造或修改 SMMR 中的中间节点或根哈希, 方法依赖 Merkle 树结构的内在特性, 任何中间节点的变动都会沿路径传递至全局根哈希. 由于全局根哈希被定期锚定于不可篡改的联盟链上, 敌手无法在不被察觉的情况下伪造出能与链上历史记录吻合的完整认证路径, 从而保证索引结构的完整性.

针对时序一致性, ETDT-SMMR 通过联盟链上按时间顺序锚定的全局根哈希序列予以保证. 验证者必须从链上获取与数据写入时刻相对应的特定历史根进行匹配, 而非始终使用最新根. 敌手无法为过时数据伪造与当前链上状态一致的证明, 从而有效保障验证结果的时效性.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

算法 5 完整性验证.

输入: 数据索引 n, 分段阈值 b, 段内证明列表 proof_L, MMR 峰证明列表 proof_M, 目标段根在峰列表中的位置索引 sid, 可信的全局 MMR 根哈希 R_e, 哈希函数 H.

$h \Leftarrow \text{proof}_L[1].h$;　/* 初始化 h 为 proof_L 的首项 */
for (h_s, pos) in proof_L **do**
　/* 从叶到段根遍历段内证明元素 */
　if pos = "L" **then**
　　$h \Leftarrow H(h_s||h)$;
　else
　　$h \Leftarrow H(h||h_s)$;
　end if
end for
$r \Leftarrow h$;
peaks $\Leftarrow$ [];
$i \Leftarrow 1$;
for $i = 1$ to length($\text{proof}_M + 1$) **do**
　/* 在 proof_M 中插入重建段根 r 到正确的 sid 位置构造完整峰序列 */
　if i = sid **then**
　　peaks.append(r);
　else
　　peaks.append($\text{proof}_M[i]$);
　　$i \Leftarrow i + 1$;
　end if
end for
$R \Leftarrow$ peaks[1];
for $j = 2$ to length(peaks) **do**
　/* 从最左峰开始折叠重建全局根 R*/
　$R \Leftarrow H(R\|\text{peaks}[j])$;
end for
return($R = R_e$);

输出: 验证结果 True (通过) 或 False (不通过).

此外, MMR 结构的增量更新特性为系统提供了前向安全能力. 当新增数据段并产生新全局根时, 历史根仍被永久记录于链上. 这意味着敌手即使破坏当前活跃数据, 也无法抹去或篡改已被锚定的历史溯源记录, 保证历史状态的完整性.

ETDT-SMMR 的安全性基于以下密码学原理.

(1) 哈希函数的抗碰撞性, 确保数据与 CID、节点与其哈希之间的唯一绑定, 是篡改检测的核心.

(2) Merkle 树和 Merkle 山脉的认证路径, 对于任意叶子节点, 仅需提供一定数量的兄弟节点哈希值即可高效验证其完整性, 而无需暴露整棵树.

(3) 区块链的不可篡改性, 联盟链作为一个分布式的信任锚点, 确保全局根哈希一旦被记录就无法被回溯性修改, 为整个链下索引结构提供最终的安全保障.

通过将 IPFS 的内容可寻址性、分层 Merkle 山脉的可验证性以及联盟链的不可篡改性与时序性相结合, ETDT-SMMR 方法从理论上能够构建一条从单条数据到可信锚点的完整信任链, 有效防御对数据完整性的攻击, 为实现高效可信的数据溯源奠定坚实的安全基础.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

4　性能分析

4.1　分段阈值选取

当前大数据背景下的可信数据溯源面临数据量大、数据到达速度快的挑战 [34], 可信数据溯源一方面要支持低延迟, 可并发的流式写入, 另一方面要能够在验证阶段高效、快速响应 [35]. 比如: 在水生态数据溯源场景中, 流域监测系统由多源传感器与周期性人工采样共同构成, 对数据实时性要求很高 [36], 数据既包含高频的连续上报, 也常在污染事件或集中取样时出现突发的批量上传. 此外, 验证请求往往具有空间或时间聚集性, 某一时间窗或某一水域段可能需要同时对一批记录进行完整性验证 (例如事后溯源与集中质检). 基于上述双重要求, SMMR 的分段策略考虑动态增量写入的低延迟和批量验证时的高效.

标准 MMR 插入一次数据可能触发多次合并操作, MMR 的高度为 $\log_2 N$ (N 为叶子节点数, 即数据总量), 因此最坏情况下一次插入操作的时间复杂度为 $T_{\mathrm{MMR}}^{\mathrm{insert}} = O(\log_2 N)$.

SMMR 将数据分段, 每段大小为 b, 每个段独立构建一个平衡 Merkle 子树 (LBT), 段根按顺序追加到 MMR 中. 插入操作包括: 插入数据到 LBT, 将事件插入段内的平衡二叉树, 最坏情况下的时间复杂度为 $O(\log_2 b)$; 插入段根到 MMR, 当段满时 (每 b 次插入触发一次), 插入一个段根到 MMR 的时间复杂度为 $O(\log_2 m)$, 其中 $m = \lceil N/b \rceil$, 表示分段的数量, 即 MMR 层的叶子节点总数.

因此, 对于单次事件插入的最坏情况, 其摊销时间复杂度如下:

$$T_{\mathrm{SMMR}}^{\mathrm{insert}} = O(\log_2 b) + O\left(\frac{\log_2 m}{b}\right), \tag{1}$$

由于 $m = \lceil N/b \rceil$, 有 $\log_2 m = \log_2(N/b) = \log_2 N - \log_2 b$, 将其代入式 (1) 得

$$T_{\mathrm{SMMR}}^{\mathrm{insert}} = O\left(\log_2 b\right) + O\left(\frac{\log_2 N}{b}\right), \tag{2}$$

接着定义时间成本函数 I_{SMMR}, 如下:

$$I_{\mathrm{SMMR}} = c_1 \log_2 b + c_2 \frac{\log_2 N}{b}, \tag{3}$$

其中 c_1 和 c_2 为大于 0 的常数, 分别反映 LBT 和 MMR 中处理单个节点的哈希计算开销.

为了找到最优阈值 b^* 使得 I_{SMMR} 最小, 计算 I_{SMMR} 的导数, 如下:

$$I'_{\mathrm{SMMR}} = \frac{c_1}{b \ln 2} - c_2 \frac{\log_2 N}{b^2}. \tag{4}$$

令 $I'_{\mathrm{SMMR}} = 0$, 得

$$b^* = \frac{c_2 \ln 2}{c_1} \log_2 N. \tag{5}$$

对式 (4) 求导, 得 I_{SMMR} 的二阶导

$$I''_{\mathrm{SMMR}} = -\frac{c_1}{b^2 \ln 2} + 2c_2 \frac{\log_2 N}{b^3}. \tag{6}$$

将 b^* 代入式 (6), 得

$$I''_{\mathrm{SMMR}} = \frac{(c_1)^3}{(c_2)^2 (\ln 2)^3 (\log_2 N)^2} > 0, \tag{7}$$

因此, 当 $b = b^*$ 时, I_{SMMR} 取得最小值.

将 b^* 代入 I_{SMMR} 并化简得

$$I_{\mathrm{SMMR}} = c_1 \log_2(\log_2 N) + c_1 \log_2\left(\frac{c_2}{c_1}\right) + c_1, \tag{8}$$

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

表 2　不同数据规模下的耗时及提升率.

Table 2　Time consumption and enhancement rates under different data scales.

N	$\log_2 N$	b	MMR	SMMR	Enhancement rate (%)
2^{10}	10	8	$10c$	$4.25c$	57.5
2^{15}	15	8	$15c$	$4.88c$	67.5
2^{20}	20	16	$20c$	$5.25c$	73.8
2^{25}	25	16	$25c$	$5.56c$	77.8
2^{30}	30	16	$30c$	$5.88c$	80.4
2^{35}	35	32	$35c$	$6.09c$	82.6
2^{40}	40	32	$40c$	$6.25c$	84.4

因此, 当 $b = b^*$ 时, SMMR 插入事件的摊销复杂度 $T^{\text{insert}}_{\text{SMMR}} = O(\log_2\log_2 N)$. 由于 $O(\log_2\log_2 N)$ 渐近小于 $O(\log_2 N)$, SMMR 在插入效率上更具优势. 在 SMMR 中, 为保证 LBT 的满二叉结构并尽可能消除空节点的填充开销, 调整 b 的取值为与 b^* 最接近/次接近的 2 的整数次幂.

式 (3) 定义了 SMMR 插入操作的时间成本函数 I_{SMMR}, 在算法实现层面, LBT 层和 MMR 层通常采用相同的哈希函数, 因此可将常数 c_1 和 c_2 视为相等, 即 $c_1 = c_2 = c$, 由此定义标准 MMR 插入操作的时间成本函数 $I_{\text{MMR}} = c\log_2 N$, 进而得到在插入操作上 SMMR 相比标准 MMR 的提升率, 如下所示:

$$\frac{I_{\text{MMR}} - I_{\text{SMMR}}}{I_{\text{MMR}}} \times 100\% = \left(1 - \frac{1}{b} - \frac{\log_2 b}{\log_2 N}\right) \times 100\%, \tag{9}$$

将不同数值的 N 代入式 (9) 得到如表 2 所示不同数据规模下 SMMR 和 MMR 的耗时及 SMMR 相比标准 MMR 在插入操作上的提升率, 随着数据规模的增大, SMMR 在插入操作耗时上显示出渐进优势.

4.2　完整性验证效率分析

本小节在 4.1 小节的基础上从单条数据完整性验证和批量数据完整性验证角度分析对比标准 MMR 和 SMMR 的效率. 定义 V_{MMR} 为标准 MMR 的验证时间成本函数, V_{SMMR} 为 SMMR 的验证时间成本函数, x 为验证的数据量, t 为 x 条数据涉及的段数, x 和 t 有如下约束:

$$\begin{cases} x, & x \geqslant 1, \\ t, & 1 \leqslant t \leqslant x, \end{cases} \tag{10}$$

对于单条数据验证, 即 $x = 1, t = 1$. 标准 MMR 的验证路径长度与高度成正比, 时间成本函数为 $V_{\text{MMR}} = c\log_2 N$. SMMR 验证包括 LBT 段内路径和 MMR 路径, 时间成本函数如下:

$$V_{\text{SMMR}} = c\log_2 b + c\log_2 m = c\log_2(b \times m) = c\log_2 N, \tag{11}$$

因此, 在验证单条数据完整性时, SMMR 与 MMR 的耗时相当.

对于批量数据验证, 考虑 2 种极端情况. 最坏情况, 所有数据位于不同段; 最好情况, 所有数据位于同一段.

在最坏情况下, 数据分布在不同段, 即 $1 < x \leqslant m$, $t = x$. 标准 MMR 每个数据项需要独立验证路径, 总时间为 $V_{\text{MMR}} = cx\log_2 N$; SMMR 每个数据项需要 LBT 段内验证, 每个段根需要 MMR 验证, 总时间如下所示:

$$V_{\text{SMMR}} = cx\log_2 b + ct\log_2 m = cx\log_2(b \times m) = cx\log_2 N, \tag{12}$$

因此, 在最坏情况下, SMMR 与标准 MMR 耗时也相当.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

最好情况下, 所有数据位于同一段, 即 $1 < x \leqslant b$, $t = 1$. 标准 MMR 验证 x 条数据所需时间仍然为 $V_{\mathrm{MMR}} = cx\log_2 N$; SMMR 所有数据在同一个 LBT 段内验证, 且只需验证一个 MMR 中的段根, 总时间如下所示:

$$V_{\mathrm{SMMR}} = cx\log_2 b + c\log_2 m = cx\log_2(\log_2 N) + c\log_2\frac{N}{b}, \tag{13}$$

因此, 在最好情况下, SMMR 批量验证所需时间相比标准 MMR 具有较大优势.

综上所述, 在数据批量验证的场景下, 当数据具有局部性, 即数据更多地分布在较少的 LBT 段时, SMMR 相比标准 MMR 具有优势, 且局部性越强, 优势越大.

5　实验及结果分析

5.1　实验环境

算法性能实验在一台处理器为 Intel(R) Core(TM) i7-9700, 内存为 16G 的 64 位 Windows10 操作系统主机上进行, 实验平台为 Pycharm, 使用 Python 语言进行代码的编写和测试, 实验使用 OpenDataLab/IPOD 公开数据集 [37]; 区块链网络性能实验在 8G 内存, 8 核处理器的 Ubuntu20.04 系统中搭建 Hyperledger Fabric v2.3 进行, 测试结果通过 Hyperledger Caliper 生成.

5.2　构造时间开销对比

本小节针对 Merkle 树、文献 [10] 提出的平衡 Merkle 树 (balanced Merkle tree, BMT), SMMR 进行构造时间开销对比. 分别测试在不同数据规模下进行构造所需要消耗的时间, 实验结果如图 6 所示, 由于不同方法时间开销差距较大, 因此在实验结果柱状图中采用断点处理.

由图 6 可知, 在同等数据量下, 构造 SMMR 所消耗的时间约为 Merkle 树的 4 倍, 这是因为 SMMR 需要进行额外的子树平衡操作并构造 Merkle 山脉, 所以相比 Merkle 树会有较大的时间开销. 同时, 相比 BMT, SMMR 由于在 LBT 层采用分段结构, 因此能够消除平衡操作随数据量增大所带来的额外开销. 如图所示, 随着数据量的增加, SMMR 的构造时间能够稳定保持在 Merkle 树的 4 倍左右, 而 BMT 每插入一条数据就需要对整棵树重新平衡, 随着数据量的增大, 平衡操作所消耗的时间会越来越多. 当数据量为 10000 时, 构造 BMT 的时间开销约为 Merkle 树的 30 倍; 当数据量为 50000 时, 构造 BMT 的时间开销约为 Merkle 树的 55 倍; 当数据量为 1000000 时, 构造 BMT 的时间开销约为 Merkle 树的 95 倍. 因此, 在构造阶段, SMMR 的时间开销相比 BMT 更少、更稳定.

5.3　数据增量插入时间开销对比

在完成 SMMR 结构的构造后, 本节首先评估单条数据插入的时间开销. 在不同数据规模 N 下, 记录向已构建好的 SMMR 结构中插入一条新数据的耗时情况, 由于单次插入时间很短且 LBT 段内平衡时左旋/右旋时间具有随机性, 所以统计重复 100 次单条数据插入的总时长并求平均时间开销, 实验结果如表 3 所示. 平均时间开销 t 的计算公式如下所示:

$$t = \frac{T}{n}, \tag{14}$$

其中 n 表示实验重复次数, T 表示重复 n 次实验的总时间.

表 3 展示了数据量为 1022, 1023, 2046, 2047, 32766, 32767, 65534, 65535 时, 插入一条数据所消耗的时间. 当数据量为 1023, 2047, 32767, 65535, 即 $2^{10}-1$, $2^{11}-1$, $2^{15}-1$, $2^{16}-1$ 时, 插入的数据刚好是当前 LBT 分段的最后一条数据, 会触发当前 LBT 分段的重构; 当数据量为 1022, 2046, 32766, 65534, 即 $2^{10}-2$, $2^{11}-2$, $2^{15}-2$, $2^{16}-2$ 时, 插入新数据仅涉及 LBT 段内平衡, 不会触发段内重构, 此时的时间开销低于触发重构时的时间开销.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

表 3　单条数据插入时间开销.

Table 3　Single record insertion time.

Data scale	b	Reconfiguration	Time consumption (ms)
1022	8	×	0.0102
1023	8	√	0.0317
2046	8	×	0.0117
2047	8	√	0.0345
32766	16	×	0.0154
32767	16	√	0.0626
65534	16	×	0.0171
65535	16	√	0.0656

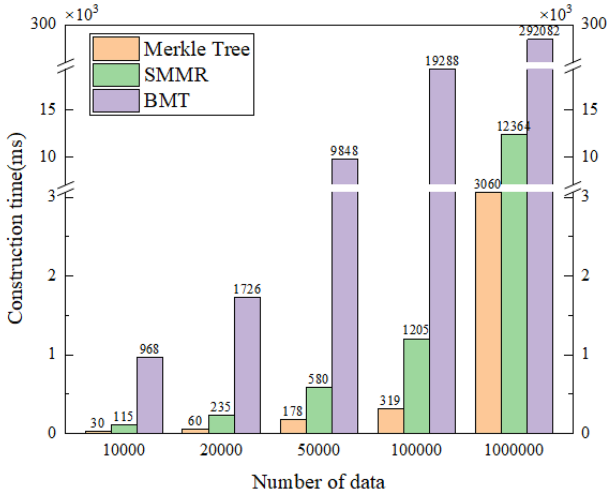


图 6　(网络版彩图) 构造时间开销对比.

Figure 6　(Color online) Experimental results of construction time.

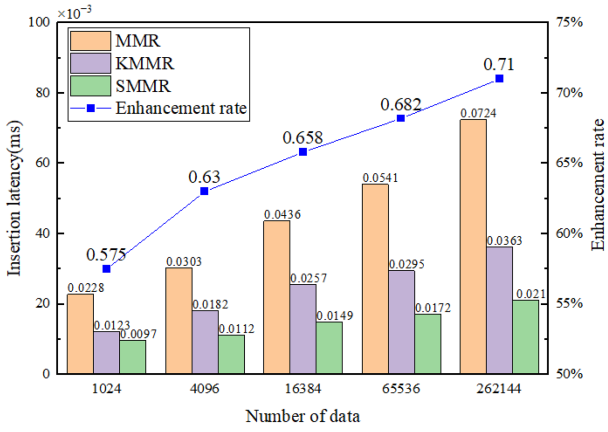


图 7　(网络版彩图) 批量插入时间开销对比及提升率.

Figure 7　(Color online) Experimental results of batch insertion time and enhancement rate.

进一步地, 触发重构的耗时与分段阈值 b 相关, 当数据量为 1023 和 2047 时, 由于分段阈值均为 8, 所以插入的时间开销相近; 当数据量为 32767 和 65535 时, 分段阈值为 16, 其插入时间也相近.

在此基础上, 进一步测试批量插入多条数据时的动态更新效率, 并与 Merkle 山脉和文献 [16] 提出的分叉 Merkle 山脉 (k-ary Merkle mountain range, KMMR) 进行对比. 实验测试了在数据量为 1024 (2^{10}), 4096 (2^{12}), 16384 (2^{14}), 65536 (2^{16}), 262144 (2^{18}) 的条件下, 分别插入当前数据总量一半, 即 2^9, 2^{11}, 2^{13}, 2^{15}, 2^{17} 条数据时, 各方法的平均时间开销 (平均时间开销计算参照式 (14)) 和 SMMR 相比标准 MMR 的提升率, 如图 7 所示. KMMR 通过分叉来降低插入数据的时间开销, 然而 KMMR 作为单一的 K 叉树结构未能降低数据插入的复杂度, 在数据量逐渐增加时仍然面临较高的更新成本, 而 SMMR 通过分层与分段策略降低数据插入的计算复杂度. 由图 7 可知, 当数据量为 1024 时, 连续插入 512 条数据, MMR, KMMR, SMMR 插入一条数据的平均时间开销分别为 0.0228, 0.0123, 0.0097 ms, SMMR 相比 MMR 的提升率为 57.5%; 当数据量为 16384 时, 连续插入 8192 条数据, MMR, KMMR, SMMR 插入一条数据的平均时间开销分别为 0.0436, 0.0257, 0.0149 ms, SMMR 相比 MMR 的提升率为 65.8%; 随着数据量的进一步增大, 当数据量为 262144 时, 连续插入 131072 条数据, MMR, KMMR, SMMR 插入一条数据的平均时间开销分别为 0.0724, 0.0363, 0.0210 ms, SMMR 相比 MMR 的提升率为 71%. 提升率与表 2 的理论结果相符, 随着数据规模的增大, SMMR 表现出渐进的插入性能优势.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

表 4　查询时间开销.

Table 4　Query latency.

Data scale	Ref. [4] (ms)	SMMR (ms)
1000	140	7
2000	190	9
5000	326	16
10000	432	21

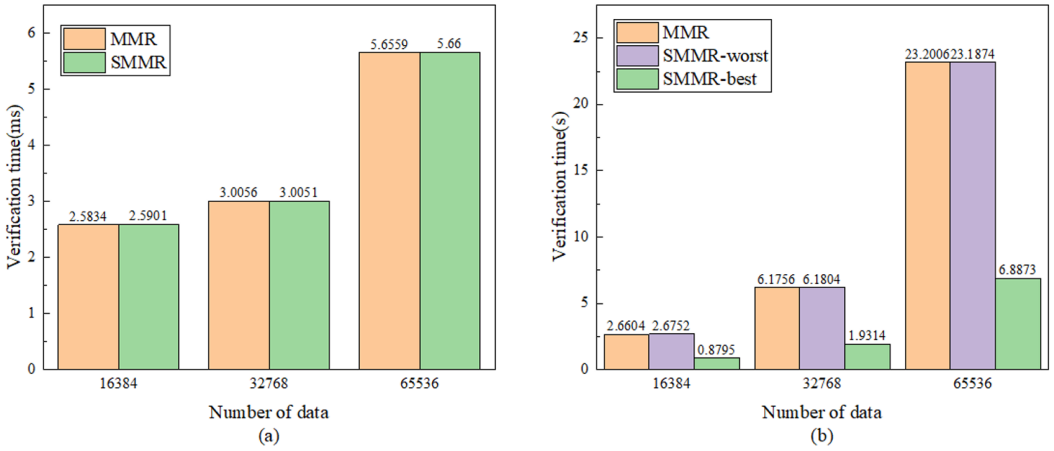


图 8　(网络版彩图) 完整性验证时间开销对比. (a) 单条数据; (b) 批量数据.

Figure 8　(Color online) Experimental results of verification time. (a) Single data; (b) batch data.

5.4　完整性验证时间开销对比

SMMR 的完整性验证过程与查询过程相关联, 因此本小节先针对查询过程的时间开销进行测试. 本小节实验将文献 [4] 提出的并行搜索方法与 SMMR 进行对比, 测试不同数据规模下的查询时间开销, 由于单次查询时间开销较小, 因此设置查询次数为 100 次并最终求平均值 (平均时间开销计算参照式 (14)), 实验结果如表 4 所示. 数据量分别为 1000, 2000, 5000, 10000 时, 文献 [4] 和 SMMR 的查询时间开销变化趋势相同, 但由于文献 [4] 的并行搜索过程在链上进行, 与区块链交互频繁, 而 SMMR 由于在链下进行查询, 因此查询时间开销更低, 大约为文献 [4] 查询时间开销的 $\frac{1}{20}$.

在完整性验证方面, 本节将文献 [38] 提出的标准 MMR 方法和 SMMR 在不同数据规模下的单次验证和批量验证时间开销进行对比. 考虑到方法之间结构的差异, 本部分实验的数据规模时选择为 2 的整数次幂, 分别测试数据规模为 16384, 32768 和 65536 时不同方法的验证时间开销. 由第 4.2 小节可知, 单次验证时 MMR 与 SMMR 的时间开销并无明显差距; 批量验证时, 以数据量为 16384 为例, SMMR 分段阈值 $b = 16$, 共有 1024 个 LBT 段, 批量验证 1024 条数据, 最坏情况为 1024 条数据分散分布在 1024 个 LBT 段, 最好情况为 1024 条数据集中分布在前 64 段. 同理, 当数据量为 32768 和 65536 时, 测试批量验证 2048 和 4096 条数据在分散分布和集中分布情况下的时间开销. 实验结果如图 8 所示.

如图 8(a) 展示了不同数据规模下 MMR 和 SMMR 进行单次验证的时间开销, 单次验证时 MMR 与 SMMR 的耗时相当. 图 8(b) 展示了不同数据规模下 MMR, SMMR 最坏情况、SMMR 最好情况批量验证数据的耗时情况. 数据量为 16384 时, MMR 批量验证 1024 条数据耗时 2.6604 s, 数据分散分布时 SMMR 耗时 2.6752 s, 数据集中分布时 SMMR 耗时 0.8795 s; 数据量为 32768 时, MMR 批量验证 2048 条数据耗时 6.1765 s, 数据分散分布时 SMMR 耗时 6.1804 s, 数据集中分布时 SMMR 耗时 1.9314 s; 数据量为 65536 时, MMR 批量验证 4096 条数据耗时 23.2006 s, 数据分散分布时 SMMR 耗时 23.1874 s, 数据集中分布时 SMMR 耗时 6.8873 s. 图 8(b) 实验

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

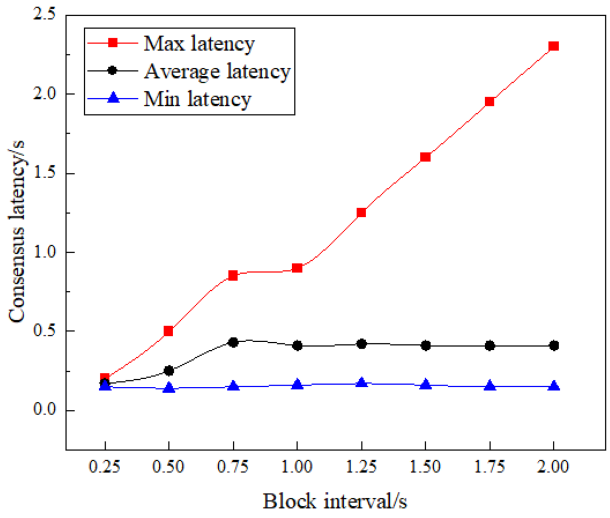


图 9　(网络版彩图) 共识时延.

Figure 9　(Color online) Experimental results of consensus latency.

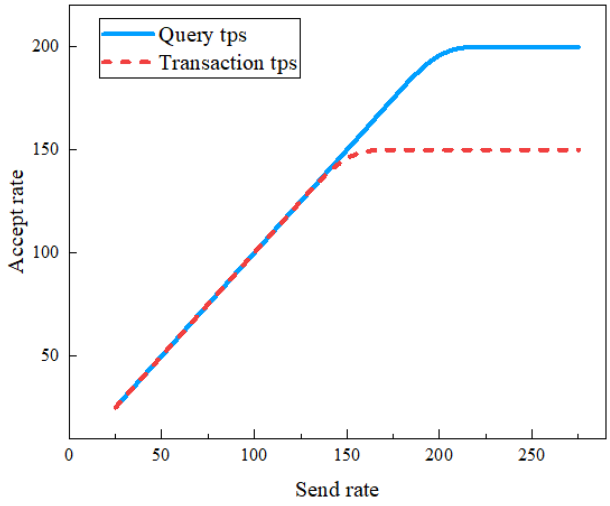


图 10　(网络版彩图) 写入和查询吞吐量.

Figure 10　(Color online) Experimental results of transaction and query transactions per second.

结果表明, SMMR 批量验证在数据分散分布时与 MMR 时间开销相当, 而 SMMR 在数据集中分布时时间开销较 MMR 提升约 70%.

5.5　区块链网络性能测试

本文提出的 ETDT-SMMR 方法最终需与联盟链协同以构建可信闭环, 其性能受链上交互环节影响. 如 3.1 小节所述, 关键链上交互包括 SMMR 全局根哈希的周期性链上锚定与完整性验证时对锚定根的查询. 为评估链上交互开销, 本小节搭建基于 Hyperledger Fabric v2.3 的测试网络, 包含 4 个节点 (分属 2 个组织), 采用 Raft 共识机制进行性能测试与分析. 根哈希锚定作为一笔写交易, 其共识时延影响溯源数据的确认速度, 实验通过调用链码函数提交根哈希, 并记录在不同出块时间下的交易共识时延. 测试结果如图 9 所示, 数据锚定时延与出块时间增长呈正相关, 这与共识机制的工作逻辑相符. 当出块时间设置为 0.75 s 时, 根哈希锚定的平均时延为 0.43 s, 最大时延为 0.85 s. 考虑到 SMMR 方法在每个 LBT 段满时才触发一次锚定, 将此固定时延分摊到单条数据上, 其开销较低, 因此, 联盟链共识流程并不影响所提方法的高效性.

在数据完整性验证过程中, 验证方需要从联盟链查询最新的可信全局根哈希, 此操作是只读查询, 无需全网共识, 但其吞吐量影响方法支持高并发验证的能力. 实验在出块时间为 0.75 s 的稳定网络中, 通过 Hyperledger Caliper 模拟不同发送速率下的查询请求, 结果如图 10 所示. 当写入请求的发送速率在 25~150 TPS 之间时, 实际查询吞吐量随之线性增长, 并在达到 150 TPS 后趋于稳定; 当查询请求的发送速率在 25~200 TPS 之间时, 实际查询吞吐量随之线性增长, 并在达到 200 TPS 后趋于稳定. 由于 SMMR 根查询操作不涉及复杂的共识流程, 其吞吐量高于 SMMR 根锚定的写入吞吐量, 链上环节完全能够胜任大规模溯源场景下海量完整性验证请求的并发负载, 为 SMMR 实现高效验证提供可靠的可信根基.

6　结束语

本文提出一种基于分层 Merkle 山脉的高效可信数据溯源方法, 通过局部平衡子树与全局 Merkle 山脉的双层索引结构, 有效缓解了区块链溯源系统中大规模数据的高并发写入与验证瓶颈. LBT 层以分段阈值管理数据流, 优化数据增量插入效率; MMR 层通过山峰进位机制实现全局根的轻量级聚合, 确保数据时序的可验证性; 同时, 结合联盟链锚定机制构建了链下 – 链上协同的信任闭环. 实验表明, 相较传统 Merkle 山脉, SMMR 在数据

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

增量插入与完整性验证性能上取得显著提升. 在保证方法正确性和安全性的前提下, 探索数据分组策略和 MMR 证明缓存机制以进一步优化性能是下一步研究的主要工作.

参考文献

1 国家数据局. 可信数据空间发展行动计划 (2024–2028 年). 2024. https://www.nda.gov.cn/sjj/xxgk/gknr/ghjh/1125/ff808081-92b8a4f1-0193-612f2a6d-04c4.pdf
2 Liu J G, Zhang H M, Zhen L. Blockchain technology in maritime supply chains: applications, architecture and challenges. Int J Prod Res, 2023, 61: 3547–3563
3 Zhuang C X, Dai Q Y, Zhang Y. BCPPT: a blockchain-based privacy-preserving and traceability identity management scheme for intellectual property. Peer Peer Netw Appl, 2022, 15: 724–738
4 Wu H Q, Jiang S, Cao J N. High-efficiency blockchain-based supply chain traceability. IEEE Trans Intell Transp, 2023, 24: 3748–3758
5 Lu J Q, Shen J, Vijayakumar P, et al. Blockchain-based secure data storage protocol for sensors in the industrial internet of things. IEEE Trans Ind Inform, 2021, 18: 5422–5431
6 Escobar C C, Roy S, Kreidl O P, et al. Toward a green blockchain: engineering Merkle tree and proof of work for energy optimization. IEEE Trans Netw Serv Man, 2022, 19: 3847–3857
7 Mizrahi A, Koren N, Rottenstreich O, et al. Traffic-aware Merkle trees for shortening blockchain transaction proofs. IEEE ACM Trans Network, 2024, 32: 5326–5340
8 Shadab R M, Zou Y, Gandham S, et al. Hmt: a hardware-centric hybrid bonsai Merkle tree algorithm for high-performance authentication. In: Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2022, 1–28
9 Faraj O, Megias D, Garcia-Alfaro J. Security approaches for data provenance in the internet of things: a systematic literature review. ACM Comput Surv, 2025, 57: 1–41
10 Xie P S, Ran Y X, Feng T, et al. Industrial OT network access authorization traceability method based on balanced Merkle tree. J Commun, 2025, 46: 282–294 [谢鹏寿, 冉玉翔, 冯涛, 等. 基于平衡 Merkle 树的工业 OT 网络访问授权溯源方法. 通信学报, 2025, 46: 282–294]
11 Bünz B, Kiffer L, Luu L, et al. Flyclient: super-light clients for cryptocurrencies. In: Proceedings of 2020 IEEE Symposium on Security and Privacy, 2020. 928–946
12 Gudgeon L, Moreno-Sanchez P, Roos S, et al. Sok: layer-two blockchain protocols. In: Proceedings of International Conference on Financial Cryptography and Data Security, 2020. 201–226
13 Perazzo P, Xefraj R. SmartFly: fork-free super-light ethereum classic clients for internet of things. IEEE Int Things, 2024, 11: 15348–15358
14 Bonneau J, Chen J, Christ M, et al. Merkle mountain ranges are optimal: on witness update frequency for cryptographic accumulators. In: Proceedings of Annual International Cryptology Conference, 2025. 170–202
15 Krishnan S, Rajendran S, Zakariah M. A secured accreditation and equivalency certification using Merkle mountain range and transformer based deep learning model for the education ecosystem. Sci Rep, 2025, 15: 22511
16 Liang C F, Zhang J L, Ma S S, et al. Study on data storage and verification methods based on improved Merkle mountain range in IoT scenarios. J King Saud Univ-Com, 2024, 36: 102117
17 Hu X Y, Song X Y, Cheng G, et al. Efficient sharing of privacy-preserving sensing data on consortium blockchain via group key agreement. Comput Commun, 2022, 194: 44–54
18 Wang B, Tian Z, Liu X R, et al. A multi-center federated learning mechanism based on consortium blockchain for data secure sharing. Knowl-Based Syst, 2025, 310: 112962
19 Ali A, Pasha M F, Guerrieri A, et al. A novel homomorphic encryption and consortium blockchain-based hybrid deep learning model for industrial internet of medical things. IEEE Trans Netw Sci Eng, 2023, 10: 2402–2418
20 Shammar E A, Zahary A T, Al-Shargabi A A. An attribute-based access control model for Internet of Things using hyperledger fabric blockchain. Wirel Commun Mob Com, 2022, 2022: 1–25
21 Wang F Q, Cui J, Zhang Q Y, et al. Lightweight and secure data sharing based on proxy re-encryption for blockchain-enabled industrial internet of things. IEEE Int Things, 2023, 11: 14115–14126
22 Zheng P L, Xu Q Q, Zheng Z B, et al. Meepo: multiple execution environments per organization in sharded consortium blockchain. IEEE J Sel Areas Comm, 2022, 40: 3562–3574
23 Lin Q, Gu B B, Nawab F. RollStore: hybrid onchain-offchain data indexing for blockchain applications. IEEE Trans Knowl Data Eng, 2024, 36: 9176–9191
24 Kuznetsov O, Rusnak A, Yezhov A, et al. Merkle trees in blockchain: a study of collision probability and security implications.

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

Int Things, 2024, 26: 101193
25 El-Hindi M, Ziegler T, Binnig C. Towards Merkle trees for high-performance data systems. In: Proceedings of the 1st Workshop on Verifiable Database Systems, 2023. 28–33
26 Kuznetsov O, Rusnak A, Yezhov A, et al. Evaluating the security of Merkle trees: an analysis of data falsification probabilities. Cryptography, 2024, 8: 33
27 Liu Y Z, Liu A D, Xia Y, et al. A blockchain-based cross-domain authentication management system for IoT devices. IEEE Trans Netw Sci Eng, 2023, 11: 115–127
28 Sarenche R, Aghili F, Yoshizawa T, et al. DASLog: decentralized auditable secure logging for UAV ecosystems. IEEE Internet Things, 2023, 10: 20264–20284
29 Xue L Y, Huang H P, Xiao F, et al. A privacy-enhanced traceable anonymous transaction scheme for blockchain. IEEE Trans Inf Foren Sec, 2025, 20: 1176–1191
30 Sudha K S, Jeyanthi N. Merkle hash tree improved strongly constrained discrete hashing function-based authentication scheme for enabling security for smart home IoT applications. Peer Peer Netw Appl, 2023, 16: 2367–2379
31 Kuznetsov O, Kanonik D, Rusnak A, et al. Adaptive Merkle trees for enhanced blockchain scalability. Int Things, 2024, 27: 101315
32 Gu K, Wang X Q, Li X. Dynamic outsourced data audit scheme for Merkle hash grid-based fog storage with privacy-preserving. IEEE Trans Sust Comput, 2024, 9: 695–711
33 Zhang X W, Feng J Q, Yin Z J, et al. Trusted query method for data provenance based on blockchain. J Appl Sci, 2021, 39: 42–54 [张学旺, 冯家琦, 殷梓杰, 等. 基于区块链的数据溯源可信查询方法. 应用科学学报, 2021, 39: 42–54]
34 Zhang L Z, Zhu Y Q, An Y Z, et al. The optimization problem of system load balancing and its solution for industrial Internet-of-Things data management. Sci Sin Inform, 2024, 54: 2343–2367 [张凌哲, 朱妤晴, 安彦哲, 等. 工业物联网数据管理中的系统负载均衡最优化问题及其求解. 中国科学: 信息科学, 2024, 54: 2343–2367]
35 Thuy T L, Nguyen M K, Bui T D, et al. Blockchain-enabled water quality monitoring: a comprehensive review of digital innovations and challenges. Water, 2025, 17: 2522
36 Palviainen M, Soininen J P, Arnold M. Requirements for water data ecosystems: results from a business ecosystem case study. Environ Syst Decis, 2024, 44: 994–1018
37 Liu J H, Ng Y C, Lim K H. A large-scale industrial and professional occupation dataset. 2020. ArXiv:2005.02780
38 Liu W, Zhang C, She W, et al. Data trusted traceability method based on Merkle mountain range. J Comput Appl, 2022, 42: 2765–2771 [刘炜, 张聪, 佘维, 等. 基于 Merkle 山脉的数据可信溯源方法. 计算机应用, 2022, 42: 2765–2771]

https://www.sciengine.com/doi/10.1360/SSI-2025-0411

Efficient trusted data traceability method based on a segmented Merkle mountain range

Xuewang ZHANG[1*], Junjie ZHANG[1] & Jinzhao LIN[2]

1. *School of Computer Science and Technology (National Exemplary Software School), Chongqing University of Posts and Telecommunications, Chongqing 400065, China*
2. *Chongqing Key Laboratory of Photo Electronic Information Sensing and Transmitting Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China*
* Corresponding author. E-mail: zhangxw@cqupt.edu.cn

Abstract Data provenance is a critical component of trustworthy governance in a trusted data space. To address the bottlenecks of high-concurrency data ingestion and real-time verification in blockchain provenance scenarios, this paper proposes an efficient trusted data traceability method based on a segmented Merkle mountain range. This method stores raw provenance records on the interplanetary file system (IPFS) and introduces a dual-layer indexing scheme-local balanced tree combined with global Merkle mountain range-that partitions data into segments according to a configurable threshold and constructs them segment-by-segment. This design optimizes both insertion and integrity-proof verification processes. The experiments measured time overheads for each stage under varying data scales and segmentation thresholds. Both theoretical analysis and experimental results show that the proposed method significantly improves the efficiency of batch data insertion compared to the traditional Merkle Mountain Range. Furthermore, as data distribution becomes more concentrated, the batch integrity verification latency demonstrates even more pronounced advantages over the conventional approach, highlighting its superior provenance performance in large-scale data scenarios.

Keywords trusted data provenance, segmented Merkle mountain range, Merkle tree, blockchain, integrity verification

https://www.sciengine.com/doi/10.1360/SSI-2025-0411