软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(11):5312−5329 [doi: 10.13328/j.cnki.jos.006748]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

SMT: 一种区块链上适用于流数据高效认证的数据结构*

孙钰山, 杨靖聪, 夏 琦, 高建彬

(电子科技大学 计算机科学与工程学院 (网络空间安全学院), 四川 成都 611731)
通信作者: 夏琦, E-mail: xiaqi@uestc.edu.cn; 高建彬, E-mail: gaojb@uestc.edu.cn

摘 要: 认证数据结构 (authenticated data structure, ADS) 解决了数据外包存储场景下服务器的不可信问题, 用户通过 ADS 可以验证不可信服务器返回查询结果的正确性与完整性, 但数据拥有者的安全性难以保证, 攻击者可以篡改数据拥有者存储的 ADS, 破坏对查询结果的完整性、正确性验证. 数据拥有者将 ADS 存储在区块链上, 借助区块链的不可篡改性, 可以解决上述问题. 但现有 ADS 实现方案在区块链上维护成本较高并且大部分只支持静态数据的可验证查询, 目前缺少一种针对区块链设计的高效 ADS. 通过分析智能合约的 gas 消耗机制与基于传统 MHT 的 ADS 的 gas 开销, 提出一种新型 ADS 认证结构 SMT, 实现对流数据的高效可验证查询, 并且在区块链上具备更低的 gas 消耗. 从理论及实验出发, 验证了 SMT 的高效性, 通过安全性分析, 证明了 SMT 的安全性.
关键词: 区块链技术; 认证数据结构; 可信存储; 流数据; gas 开销
中图法分类号: TP311

中文引用格式: 孙钰山, 杨靖聪, 夏琦, 高建彬. SMT: 一种区块链上适用于流数据高效认证的数据结构. 软件学报, 2023, 34(11): 5312–5329. http://www.jos.org.cn/1000-9825/6748.htm
英文引用格式: Sun YS, Yang JC, Xia Q, Gao JB. SMT: Efficient Authenticated Data Structure for Streaming Data on Blockchain. Ruan Jian Xue Bao/Journal of Software, 2023, 34(11): 5312–5329 (in Chinese). http://www.jos.org.cn/1000-9825/6748.htm

SMT: Efficient Authenticated Data Structure for Streaming Data on Blockchain

SUN Yu-Shan, YANG Jing-Cong, XIA Qi, GAO Jian-Bin

(School of Computer Science and Engineering (School of Cyber Security), University of Electronic Science and Technology of China, Chengdu 611731, China)

Abstract: The authenticated data structure (ADS) solves the problem of untrusted servers in outsourced data storage scenarios as users can verify the correctness and integrity of the query results returned by untrusted servers through the ADS. Nevertheless, the security of data owners is difficult to guarantee, and attackers can tamper with the ADS stored by data owners to impede the integrity and correctness verification of query results. Data owners can store the ADS on the blockchain to solve the above problem by leveraging the immutable nature of the blockchain. However, the existing ADS implementation schemes have high maintenance costs on the blockchain and most of them only support the verifiable query of static data. At present, an efficient ADS tailored to the blockchain is still to be designed. By analyzing the gas consumption mechanism of smart contracts and the gas consumption of the ADS based on the traditional Merkle hash tree (MHT), this study proposes SMT, a new ADS, which achieves efficient and verifiable query of streaming data and has a lower gas consumption on the blockchain. Finally, the study verifies the efficiency of SMT both theoretically and experimentally and proves the security of SMT through security analysis.
Key words: blockchain technology; authenticated data structure (ADS); trusted storage; streaming data; gas consumption

随着云计算和云存储的快速发展, 物联网设备将采集的数据上传到云服务器中进行存储, 这样解决了物联网设备数据存储能力有限的问题, 并且方便用户共享数据. 用户将自己的数据加密后上传到云服务器并设置访问控

* 基金项目: 四川省卫生健康基础资源共享区块链服务平台共建及应用试点项目 (2021ZXKY06001)
收稿时间: 2022-01-24; 修改时间: 2022-03-29, 2022-07-03; 采用时间: 2022-07-25; jos 在线出版时间: 2023-03-29
CNKI 网络首发时间: 2023-03-30

© 中国科学院软件研究所 http://www.jos.org.cn

制策略, 满足访问控制策略的用户可以通过云服务器获得数据, 再解密后得到原始数据. 基于云的数据存储与共享给用户带来了很多便利, 但数据的安全性依赖于云服务器. 云服务器存储的数据可能发生丢失篡改等情况. 上述情况下, 用户取回的数据是不完整、不正确的, 因此如何在该环境下保证数据的完整性与正确性成为研究的一个热点.

近年来区块链技术由于去中心化与不可篡改性, 广泛应用于金融、医疗保健、物联网和供应链等场景中. 区块链是由网络中不信任节点共同维护的一种追加数据结构, 随着具备智能合约功能的二代区块链的出现, 区块链技术也被采用为更加通用的数据可信存储方案. 由于区块链网络中各节点存储与维护了相同的数据副本, 导致在区块链中存储原始数据成本开销过高. 为解决上述问题, 之前的研究提了一种混合存储架构, 数据拥有者 (如物联网设备) 将数据摘要发送到区块链上安全存储, 原始数据脱离区块链存储在云服务器上, 区块链与云服务器动态维护了 ADS (authenticated data structure). ADS 是混合存储架构的核心, 用户使用 ADS 可以验证从链外检索数据的完整性与正确性. 目前实现认证数据结构方式主要有跳表、索引 hash 链表、MAC tree 和区块链上使用最广泛的 Merkle hash tree (MHT). 在以太坊中智能合约的运行需要花费矿工的资源, 用户通过 gas (燃料) 向矿工支付存储和计算费用, 并且智能合约中不同的操作消耗的 gas 差距非常大, 写数据操作远高于读数据与更新数据操作 (20000 vs. 200, 5 000). MHT 中追加写入数据时需要更新叶节点到根节点路径上所有节点信息, 并且需要存储完整的 MHT, 因此在区块链上基于 MHT 构建支持流数据的 ADS 成本过高.

在区块链上设计一种支持流数据高效追加更新的新型数据结构, 是实现基于区块链的流数据可验证查询的关键. 本文提出了一种新型 ADS 认证结构 SMT, 可以在区块链上能有效地维护同时支持高效的可验证查询. 首先, 本文分析智能合约的 gas 消耗机制与基于传统 MHT 的 ADS 的 gas 开销, 得到相关问题, 进一步提出了区块链上高效 ADS 设计原则. 基于该原则提出一种由子树集合描述的新型 ADS (SMT), SMT 在区块链上只存储各子树的根哈希值, 在流数据更新时只更新一个根哈希值, 降低了区块链上维护 ADS 的成本, 同时提高了验证查询结果的性能. 再通过安全性分析将 SMT 安全性规约到使用的哈希函数的抗碰撞性, 证明了 SMT 的安全性. 通过模型分析得到了 SMT 各操作在理论上的 gas 开销, 验证了 SMT 的高效性. 最后, 通过对比实验也验证了 SMT 本文提出的新型 ADS 执行流数据可验证查询的有效性.

本文第 1 节介绍可验证查询相关方法和研究现状. 第 2 节介绍本文所需的基础知识, 包括哈希函数、默克尔树、区块链和智能合约. 第 3 节介绍本文构建的新型 ADS (SMT) 模型. 第 4 节通过对比实验验证了所提模型的有效性. 第 5 节总结全文.

1 可验证查询相关工作

云存储的快速发展给广大用户带来了诸多便利同时, 也改变了传统意义数据存储的模式. 在云存储环境下, 数据拥有者的所有数据外包存储在第三方的存储设备上, 存储设备作为代理商响应用户的查询请求, 由于第三方服务器是不可信的, 存储在第三方存储设备上的数据可能被恶意的篡改或丢失, 而可验证查询 (authenticated query processing) 技术实现了用户对不可信服务器返回数据的完整性、正确性验证, 解决了外包存储环境下[1−3]第三方不可信问题, 近年来收到了广泛的研究.

目前对于可验证查询的相关研究大多在外包存储场景下开展, 可验证计算技术 (verifiable computation, VC) 与认证数据结构 (ADS)[4]是实现可验证查询的主要方法. 相较于基于 VC 的可验证查询方案, 基于 ADS 可验证查询方案只适用于特定类型的查询方式, 但具备更高的性能. 目前基于 ADS 的可验证查询方案的评价指标[5]主要有以几点: (1) 数据拥有者的计算开销. (2) 数据拥有者与第三方存储服务器的通信成本. (3) 第三方存储设备的存储开销. (4) 第三方服务器的计算成本. (5) 用户的验证成本. ADS 的主要实现方式为数字签名技术与 MHT[6]. 数字签名技术通过非对称加密技术, 解决了外包存储场景下对查询结构的完整性与正确性验证.

在基于 VC 的可验证查询领域, 通过 SNARKs 算法[7]实现的 VC 方案, 支持任意类型的查询方式 (联合查询、范围查询和等值查询等), 但 SNARKs 算法需要执行预处理过程, 该过程将数据与查询算法硬编码为电路并且根据电路生成 PK (proving key) 和 VK (verification key), 需要消耗大量的计算时间与内存空间. 在查询方案较复杂时,

© 中国科学院软件研究所 http://www.jos.org.cn

基于 VC 的可验证查询方案开销过大. 针对上述问题, Ben-Sasso 等人[8]改进了 SNARKs 算法, 将 SNARKs 预处理过程开销降低到依赖数据集合的上限大小与查询算法的复杂度. 2017 年 Zhang 等人[9]通过一种交互式协议实现了可验证的 SQL 查询方案 vSQL, 该方案解决了基于 VC 的可验证查询方案的开销问题, 但只适用于固定查询方式的关系数据库.

在基于签名改进的 ADS 可验证查询领域, Mykletun 等人[10]提出了一种基于聚合签名技术的数据完整性验证方案, 解决了多数据拥有者的外包存储架构下数据完整性验证问题, 但该方案不能将不同用户的签名进行聚合, 计算复杂程度较高. Narasimha 等人[11]提出了名 DSAC 的方案, 通过聚合与链式签名的方式保证了外包查询中数据的正确性与完整性. Pang 等人[12]在关系型数据库的连接查询场景下, 提出一种具备访问控制模式的可验证查询方案. 基于数字签名的可验证查询方案在小数据集情况下具备较高的性能, 但需要对每个数据记录进行签名, 不适用于大型数据集合[13].

在基于数据架构改进的可验证查询领域, MHT 被广泛应用于各种索引结构的构建中[4,14,15]. Papamanthou 等人[16]针对流数据提出了一种认证数据结构, 通过格密码相关技术改进了传统 MHT, 降低了流数据的可验证查询时间复杂度. Papadopoulos 等人[17]提出了一种验证流数据完整性的开源框架 VeriStream, 该框架传输带宽与时间复杂度方面具备良好的表现, 但需要数据拥有者在本地维护由所有数据构建的 MHT, 并且执行可验证查询时具有较高的延时, 不适用于在线可验证查询.

在基于区块链的可验证查询领域, Xu 等人[18]提出了 vChain 的通用框架, 通过加密累加器与 MHT 在区块链上构建了一种新型 ADS, 实现了区块链的等值查询于范围查询的完整性与正确性. 但 vChain 与传统区块链差异过大, 直接应用价值较低. Zhang 等人[19]在区块链上提出了 GEM^2-tree 的新型 ADS 认证结构, 该结构通过智能合约在区块链上进行动态维护, 主要思想是将消耗 gas 较高的更新操作替换为计算操作, 降低了 ADS 在区块链上的维护成本.

综合各类可验证查询研究现状, 本文提出适合区块链流数据特点的可验证查询数据结构 SMT, 基于 MHT 进行独立子树设计, 与上述 ADS 类方案项目, SMT 不仅特化了流数据处理能力, 还兼顾了区块链低数据存储量和高验证效率的要求, 并能降低区块链上 gas 开销.

2 基础知识

2.1 哈希函数

加密哈希函数 $hash$ 将任意长度的消息 m 映射到固定长度的消息摘要 $hash(m)$. 它具有两个重要的性能: 单向特性和抗碰撞特性. 单向性表明, 给定一个摘要 $hash(m)$, 一个 PPT 敌手可以找到原始消息 m 的概率可以忽略不计. 抗碰撞性意味着一个 PPT 敌手在计算上找到两个不同的消息 m_1,m_2 使 $hash(m_1) = hash(m_2)$ 是不可行的. 目前典型的加密哈希算法分为 SHA-1、SHA-2、SHA-3 这几种类型. 表 1 说明了常见哈希函数在区块链上的 gas 消耗. 本文选择以太坊中广泛使用的 Keccak-256 作为构建 SMT 的加密哈希函数.

表 1 常见哈希函数的 gas 消耗

哈希函数	Gas cost
SHA-256	60
MiMC e7r91	8.9k
Poseidon t6f8p57	56.4k
Keccak-256	1.4k

2.2 区块链和智能合约

近年来随着比特币与以太坊的出现, 区块链技术受到了各方的关注. 由于区块链技术具备可溯源、防篡改、分布式共识等特性, 目前被广泛应用于金融、物流、物联网及城市建设[20−24]等领域. 区块链是由多方共同维护的

© 中国科学院软件研究所 http://www.jos.org.cn

去中心化的分布式账本, 它通过 P2P 网络协议、共识算法、非对称加密、哈希函数等关键技术, 解决数据传递与交换过程中的信任问题. 区块链的链式结构是将共识算法广播的数块按时间戳进行存储与验证的数据结构, 通过密码学技术保证数据传输与访问的安全性. 目前区块链可分为公有链、私有链和联盟链. 在公有链中节点可以任意加入与退出区块链网络, 而在联盟链中通过认证授权的节点才能接入公有链网络.

图 1 展示了区块链链式数据结构, 它以区块为存储单元, 每个区块中包含区块头和区块体. 区块中包含状态树、交易树和收据树, 并统一使用 MPT (Merkle Patricia trie) 树存储, MPT 树相较于 Merkle 树具备支持多版本、去重以及快速更新的特点. 其中交易的执行会修改状态树中的状态, 并且形成交易的回执. 智能合约部署时, 会在状态树中创建一个合约账户, 并将智能合约代码存储在不变的 *Code* 中, 合约中定义的状态变量存储在 *Storage* 中. 当新区块被发布时, 矿工通过账户地址检索账户信息, 若交易为转账交易则修改账户中的 *Balance* 字段, 若为交易为合约调用则将智能合约代码载入 EVM 虚拟机中执行, 并根据合约运行情况修改 *Storage* 中的合约变量. 区块间会共享不变状态的分支, 并为变化状态创建新分支, 新区块中的状态树中包含创建新的分支与指向不变分支的索引指针.

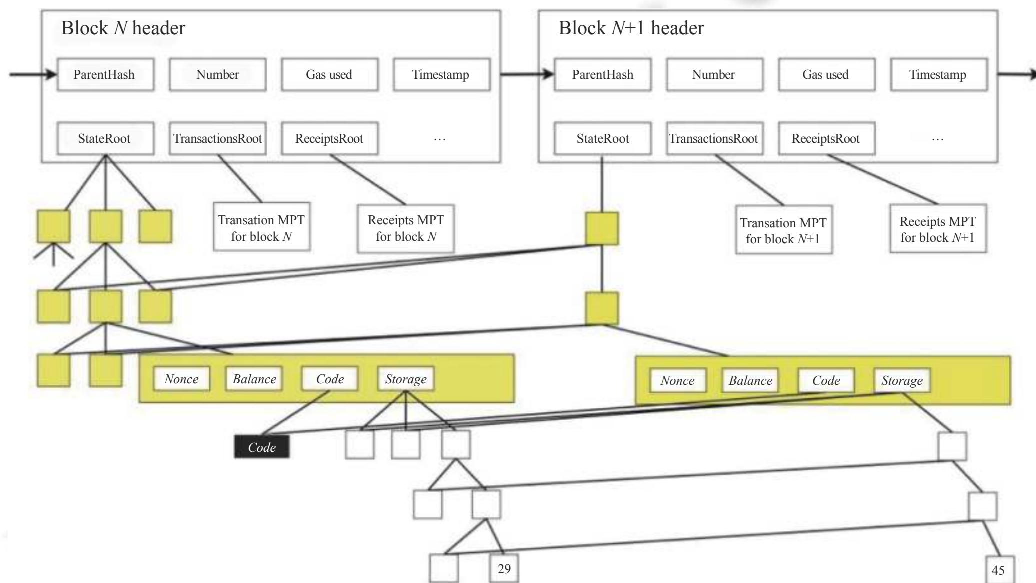


图 1 链式数据结构

表 2 显示了以太坊平台中智能合约常见操作的 gas. 可以看出, 存储数据到区块链上比更新区块链上数据消耗更多 gas, 而加载区块链上的数据和访问内存变量消耗少量 gas. 此外, 为了防止智能合约浪费太多的计算资源. 矿工们引入了一个燃料极限 (8 000 000). 如果总燃料消耗超过燃料极限, 合约的执行将被中止, 所以基于智能合约编程时应尽量减少 gas 消耗.

表 2 智能合约常见操作的 gas 消耗

Operation	Gas used	Explanation
C_{load}	200	从区块链上加载一个变量到内存中
C_{store}	20 000	向区块链上存储一个变量
C_{update}	5 000	更新区块链上的一个变量
C_{mem}	3	访问一次内存中的变量

2.3 MHT

MHT 是由 Merkle[25]提出的最早认证数据结构, 能在对数时间范围内验证数据集合的存在性, 目前被广泛应

© 中国科学院软件研究所 http://www.jos.org.cn

用于区块链、可验证查询、索引结构和订阅查询中. MHT 是一种二叉树结构, 其叶节点存储了各数据块的哈希值, 中间节点是由左右节点连接后的哈希值. 进行可验证查询时只需要对叶节点到根节点的一条或若干条路径进行处理, 就可以高效的证明数据集合的完整性和正确性. 将完整的数据集合进行分块后, 通过数据块迭代构建 MHT, 构建完成的 MHT 支持高效的数据更新操作, 但是不支持类似流数据的追加写入操作. 因此 MHT 常用于构建静态数据的 ADS, 对于动态数据的支持不友好.

图 2 介绍了一种基于 MHT 的认证数据结构 ADS 方案, 分析了此方案在区块链上的维护成本与相关问题, 为本文所提的新型 ADS 提供思路. 云服务提供商与区块链中创建并维护了两个相同的 MHT, 其中云服务提供商需要通过数组额外存储原始数据. 数据拥有者 (物联网设备) 采集到新数据块后, 将原始数据哈希值发送到区块链上触发智能合约更新 *data_node* 状态, 并将原始数据发送给云服务提供商. 云服务提供商接收数据后, 更新 *data_node* 与 *data_arr* 状态.

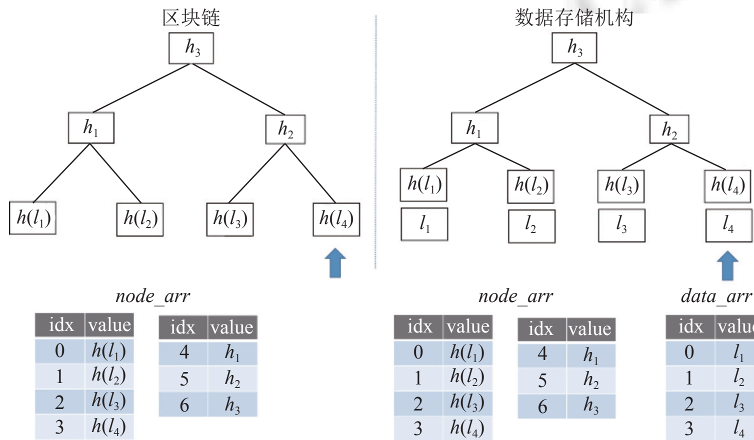


图 2　基于 MHT 的 ADS

云服务提供商收到查询请求 Q 时, 检索本地 *data_arr* 得到查询结果 R, 根据 *data_node* 维护的 MHT 生成验证对象 VO_{sp}, 返回 $\langle R, VO_{sp}\rangle$ 给用户. 用户调用智能合约获取区块链上维护的 MHT 根节点值 VO_{chain}, 通过本地重新构建 MHT 得到根节点值 $MHT^{\text{root}}_{\text{rebuild}}$, $MHT^{\text{root}}_{\text{rebuild}}$ 与 VO_{chain} 相等则验证通过. 例如图 2 所示云服务提供商收到 $Q=[2,3]$ 查询数据流中的第 2 块到第 3 块数据时, 各参数如下 $R=[l_2,l_3], VO_{sp}=[h(l_1),h(l_4)], VO_{\text{chain}}=[h_3], MHT^{\text{root}}_{\text{rebuild}}=h_3$. $R=[l_2,l_3], VO_{sp}=[h(l_1),h(l_4)], VO_{\text{chain}}=[h_3], MHT^{\text{root}}_{\text{rebuild}}=h_3$.

假定物联网设备采集了 n 块数据, 分析区块链上维护 MHT 的开销. 由于区块链上需要存储完整的 MHT 结构, MHT 的节点数量为数据块数的 2 倍, 存储开销为 $C^{\text{save}}_{\text{MHT}}=(2n-1)\times C_{\text{store}}$. 流式数据追加更新时, 需要更新 1 个叶节点与 $\log_2 n-1$ 个前驱节点, 每个前驱节点更新时需要加载另一个孩子节点进行哈希运算, 总共需要加载 $\log_2 n-1$ 个孩子节点与进行 $\log_2 n-1$ 次哈希运算, 总花费 $C^{\text{insert}}_{\text{MHT}}=(\log_2 n-1)\times\left(C_{\text{load}}+C_{\text{hash}}+C_{\text{update}}\right)+C_{\text{update}}$. 进行查询验证时, 需要通过智能合约获取区块链上 MHT 的根节点值, 所以区块链上验证查询开销为 $C^{\text{verify}}_{\text{MHT}}=C_{\text{load}}$.

3　基于区块链的新型 ADS 认证结构

如图 3 所示本系统由 4 个部组成: 数据拥有者、具备智能合约功能的区块链系统、云服务提供商和用户. 区块链系统与云服务提供商共同构建基于区块链的可信存储系统. 流数据由二元组 $\langle i, v_i\rangle$ 表示, 其中 i 为数据块的编号, v_i 是第 i 块流数据的内容. 流数据更新时, 数据拥有者发送 $o_i=\langle i, v_i\rangle$ 到云服务提供商, 由于区块链的公开性发送 $o_i=\langle i, h(v_i)\rangle$ 到区块链上.

区块链与云服务提供商维护认证数据结构 ADS 构建了可信存储系统, 流数据的更新会触发智能合约更新区块链中的 ADS, 同时云服务提供商会相应的更新本地数据. 用户向云服务提供商请求数据, 云服务提供商检索本

地 ADS 得到查询结果 R 并生成验证对象 VO_{sp}, 返回 $\langle R, VO_{sp}\rangle$ 给用户. 用户调用智能合约得到区块链上存储的 VO_{chain}, 通过 VO_{sp} 与 VO_{chain} 验证查询结果 R 的完整性与正确性.

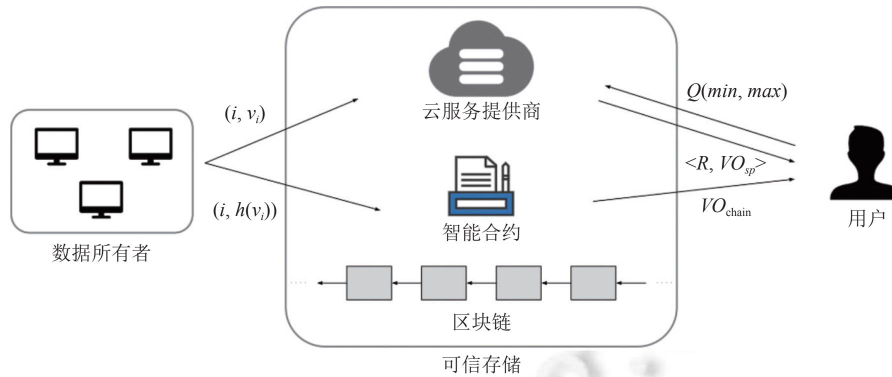


图 3 系统组成

3.1 区块链上 ADS 设计原则

基于 MHT 的可验证流数据查询方案在数据插入时, 需要执行 $\log_2 n$ 次高 gas 消耗操作 $(C_{\text{load}}, C_{\text{update}})$, 执行查询验证时只需获取 MHT 的根哈希值 VO_{chain}, 而区块链上需维护完整的 MHT. 基于上述分析本文得到如下原则设计新型 ADS 认证结构, 新型 ADS 结构支持可验证查询的同时在区块链上具备高效性.

(1) 区块链上存储最少量的数据. 基于 MHT 的方案中, 区块链上需要完整的树型结构 (叶子节点、非叶子节点), 而进行查询有效性验证时只需 MHT 的根哈希值, 存储其余节点的是为了在流数据更新时, 快速计算 MHT 的根哈希值. 而区块链上存储数据 C_{store} 消耗大量 gas, 当流数据规模较大时基于 MHT 的方案在区块链上需消耗大量 gas.

(2) 减少流数据更新时区块链上更新节点的数量. MHT 是基于二叉树构建的数据结构, 树的高度随着存储的数据规模成对数关系. 流数据更新时, 区块链需要更新数据存储节点到根节点路径上所有节点数据, 在流数据规模较大时消耗 gas 过高.

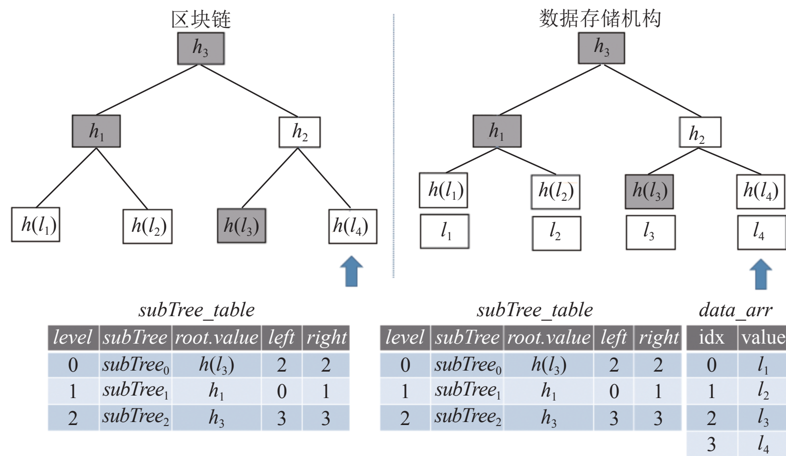
(3) 适用于任意大小数据. 物联网设备采集的数据是无边界的流式数据, 基于 MHT 的 ADS 认证方案只在数据规模较小情况下性能与 gas 开销表现较好, 而一种理想的 ADS 认证方案需要适用于各种规模数据.

3.2 SMT 的具体设计

按照上述 ADS 设计原则我们提出了一种新型 ADS 认证结构 SMT. SMT 在区块链上具备低 gas 开销特点, 并且支持高效的可验证查询. SMT 的主要设计思想是通过一个子树集合表示完整的 SMT, 每个子树负责各自包含数据集合的查询与验证. 由于流数据的更新只有追加更新, 不存在对已写入流数据的更改, 所以对 SMT 在设计时只考虑了追加更新. 图 4 展示了一个 SMT 的实例, SMT 分为采用混合架构进行存储, 区块链上维护 SMT_{onchain} 和数据存储机构维护链下 SMT_{offchain}. 数据存储机构响应用户的查询请求, 检索 SMT_{offchain} 得到查询结果 R 并生成验证对象 VO_{sp}, 用户通过 VO_{sp} 与 SMT_{onchain} 中维护的 VO_{chain}, 验证 R 的正确性与完备性.

SMT 维护了存储流数据的数组 $data_arr$, $data_arr$ 只存在于 SMT_{offchain} 中, 其数组下标标识流数据块索引, 值存储了原始数据. SMT 使用一个子树集合表示完整的树结构, 该结构保障了 SMT_{offchain} 中流数据的完整性, 并且基于该结构可构建流数据的可验证查询方案. SMT 中的子树集合使用子树表格 $subTree_table$ 表示, SMT_{onchain} 和 SMT_{offchain} 维护了相同的 $subTree_table$, $subTree_table$ 随着流数据的追加更新而变化, 其中每行数据描述了子树的元信息 ($level$ 代表子树根节点所在二叉树层数, $root.value$ 代表子数根节点值, $\langle left, right\rangle$ 代表子树包含数据集合的下标), 并且每层都存在唯一的子树根节点. 相较于传统 MHT 存储 $2n$ 个节点, SMT_{onchain} 在区块链上存储了 $\log_2 n$ 个节点, 具备更低的 gas 消耗.

© 中国科学院软件研究所 http://www.jos.org.cn



level	subTree	root.value	left	right
0	$subTree_0$	$h(l_3)$	2	2
1	$subTree_1$	h_1	0	1
2	$subTree_2$	h_3	3	3

level	subTree	root.value	left	right
0	$subTree_0$	$h(l_3)$	2	2
1	$subTree_1$	h_1	0	1
2	$subTree_2$	h_3	3	3

idx	value
0	l_1
1	l_2
2	l_3
3	l_4

图 4　基于 SMT 的 ADS 认证结构

SMT 对流数据追加更新的设计如下, 物联网设备采集到新数据后, 发送原始数据到云服务提供商, 发送数据哈希值到区块链上. 原始数据直接追加写入 $data_arr$ 中, 对于 $subTree_table$ 云服务提供商与智能合约执行以下相同的操作, 如果查询节点在树中属于右孩子节点, 通过当前层子树根哈希值迭代计算哈希, 上述操作结束后更新节点所在层对应子树的根哈希值. SMT 在流数据更新时只需执行一次追加写入操作与更新一个节点数据, 满足设计原则 2 和设计原则 3 具备高效性.

SMT 对数据检索设计如下, 云服务提供商收到查询请求后, 分别在 SMT_{offchain} 各子树中检索查询结果 R 生成并生成验证对象 VO_{sp} . 相比于存储完整节点信息的 MHT, SMT_{offchain} 每个子树除根节点外只存储叶子节点数据, 云服务提供商在 SMT 子树中生成验证对象时, 需要额外的哈希运算得到非叶子节点信息, SMT_{offchain} 在数据检索设计上虽然提高了云服务提供商的计算成本, 但是降低了云服务商与区块链上的存储成本, 特别是区块链上存储操作本身花销较大, 此设计满足设计原则 1 是有效的.

SMT 对于查询结果验证设计如下, 用户通过云服务提供商返回的 $\langle R, VO_{sp}\rangle$ 重新构建各子树, 通过对比计算得到的子树根哈希值与区块链上 SMT_{onchain} 维护的子树根哈希值 VO_{chain} , 验证查询结果的正确性. SMT 与 MHT 在验证查询结果时都通过哈希运算得到根哈希值, 哈希运算的次数与查询验证开销成线性关系, 由于 SMT 是多子树结构, 云服务提供商返回 VO_{sp} 中元素更少, 表明了 SMT 在查询结果验证时的高效性.

3.3　SMT 支持的数据操作

SMT 支持如下操作: 初始化 SMT、流数据追加更新、云服务提供商执行可验证查询检索、用户对查询结果的验证. 云服务提供商使用 $subTree_table$ 和 $data_arr$ 在本地维护 SMT_{offchain} . 本文将 SMT_{onchain} 通过合约账户维护在区块链状态树中, 合约账户包含 $\langle Nonce, Balance, Code, Storage\rangle$ 属性, $Storage$ 中存储 $subTree_table$ 的各层子树根哈希值 $root.value$, SMT_{onchain} 支持的数据操作通过合约实现并存储在 $Code$ 中.

● 初始化 SMT. 算法 1 描述了初始化 SMT 操作, 数据拥有者向区块链和云服务提供商发送初始化 SMT 请求, 指定创建二叉树的层树 $level$, 与默认零值 zero, 其中 zero 用于计算存储数据为空时各层子树的根哈希值. 区块链收到请求后初始化 SMT_{onchain} , 首先在状态树中创建一个智能合约账户, 并将 SMT_{onchain} 相关操作的合约代码存储在 $Code$ 字段中, 后续约执行以下操作: 设置第 0 层子树根哈希值为 zero, 再迭代计算得到各层根哈希值, 在新区块的合约账户中写入各层子树根哈希值. 云服务商收到请求后初始化 SMT_{offchain} , 执行与初始化 SMT_{onchain} 相同的迭代运算得到各层子树根哈希值, 将结果写入本地 $subTree_table$ 中.

算法 1. 初始化 SMT (initSMT).

Input: initial level $level$, default value zero .

1. $arr \leftarrow []$
2. $i \leftarrow 1$; $arr.append(\text{zero})$
3. **while** $i < level$ **do**
4. $\quad arr.append(h(arr[i-1] | arr[i-1]))$
5. $\quad i \leftarrow i+1$ /**初始化 SMT_{onchain} **/
6. $newBlock \leftarrow creatBlock()$
7. $account \leftarrow newBlock.createContractAccount()$
8. $account.add(0, \text{zero})$
9. **for** $b \in arr$ **do**
10. $\quad account.Storage.add(b)$
11. $blockChain.append(newBlock)$ /**初始化 SMT_{offchain} **/
12. $subTree_table \leftarrow []$
13. $data_arr \leftarrow []$
14. **for** $b \in arr$ **do**
15. $\quad subTree_table.append(b)$

图 5 和图 6 展示了 $level=3$ 的 SMT 初始化后的结构, 当前 SMT 包含 3 棵子树并且最大容量为 4. 为满足设计原则 1, SMT_{offchain} 中 $subTree_table$ 中只存储了各子树的根哈希值, $\langle level, subTree \rangle$ 通过数组下标进行标识, $\langle left, right \rangle$ 通过 SMT 的层数与当前插入元素的个数计算得到, 详细计算过程见算法 2. SMT_{onchain} 在创建时只在区块 N–1 中增加了各层子树根哈希值 $root.value$, 其中各层根哈希值为 $\langle h_1 = \text{zero}, h_2 = hash(h_1|h_1), h_2 = hash(h_2|h_2) \rangle$, $\langle level, subTree \rangle$ 字段通过 $Storage$ 中索引的顺序确定, 执行可验证查询时 SMT_{onchain} 使用各子树根哈希值构建 VO_{chain} 验证 SMT_{offchain} 返回查询结果, 因此 $subTree_table$ 中 $\langle left, right \rangle$ 字段不需在 SMT_{onchain} 中维护.

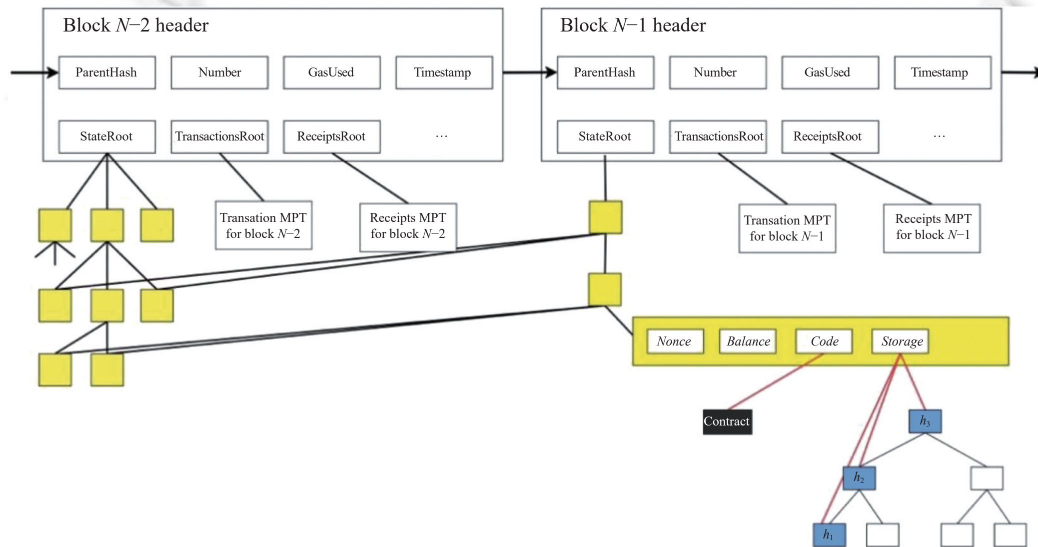
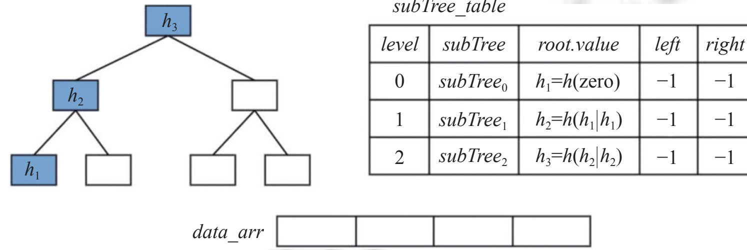


图 5 SMT_{onchain} 初始化示意图

● 流数据追加更新. 算法 2 描述了初始化 SMT 操作, 流式数据更新时只在原始数据后进行追加更新, 不需要删除、修改原始数据. 数据拥有者将追加的原数据 l_i 发送给云服务提供商, 数据哈希值 $h(l_i)$ 发送到区块链上. 云服务提供商和区块链修改 SMT_{offchain} 和 SMT_{onchain} 的数据结构. 若 SMT 中存储的数据达到最大容量则执行扩容, 扩容

© 中国科学院软件研究所 http://www.jos.org.cn

的基本思想是创建一个比原始 SMT 层数多 1 层的新 SMT, 将原始 SMT 作为新 SMT 的左子树. 在层数为 $level+1$ 的 SMT 左子树中插入数据与在层数为 $level$ 的 SMT 插入数据时更新操作完全相同, 所以云服务提供商只需在 SMT_{offchain} 中新增一个表示 $level+1$ 层子树的元数据信息, 而区块链需在新区块的合约账户新增一个表示表示 $level+1$ 层子树根哈希值维护 SMT_{onchain} (步骤 3–6). SMT 扩容后云服务提供商将 l_i 写入本地 $data_arr$ 中, 同时云服务提供商与区块链将 $h(l_i)$ 追加写入 SMT_{offchain} 和 SMT_{onchain} 叶节点 N_i 中, 若 N_i 为右孩子节点, 使用 N_i 所在层子树根哈希值计算 $h=hash\left(root.value_j|h\right)$, 并设置 N_i 为其父节点迭代计算 h, 当 N_i 为左孩子节点或根节点时结束. 设上述迭代计算在第 k 层结束, 云服务提供商维护动态维护 SMT_{offchain}, 更新本地 $subTree_table$ 中第 k 层子树根哈希值为 h (步骤 7–14), 区块链动态维护 SMT_{onchain}, 在当新区块状态树中创建第 k 层子树根节点哈希值 h, 并构建指向不变子树根哈希值的索引指针 (步骤 15–22).



level	subTree	root.value	left	right
0	$subTree_0$	$h_1=h(\text{zero})$	−1	−1
1	$subTree_1$	$h_2=h(h_1\|h_1)$	−1	−1
2	$subTree_2$	$h_3=h(h_2\|h_2)$	−1	−1

图 6 SMT_{offchain} 初始化示意图

算法 2. 插入元素 ($insert$).

Input: Data value l_i, Data hash value $h(l_i)$.

1. $account \leftarrow block_{\text{last}}.getAccount()$
2. $newSubTreeRoot \leftarrow h(arr[len(arr)-1]|arr[len(arr)-1])$ /**若 SMT 达到最大容量, 执行扩容操作**/
3. **if** $2^{level-1} \leqslant insertIndex$ **then**
4. $\quad SMT.level \leftarrow SMT.level+1$
5. $\quad SMT_{\text{offchain}}.append(newSubTreeRoot)$
6. $\quad account.Storage.add(newSubTreeRoot)$ /** SMT_{offchain} 追加写入**/
7. $N_i = SMT_{\text{offchain}}.getNode(insertIndex)$
8. $current_hash \leftarrow h(l_i)$
9. **while** N_i $isarightnode$ **do**
10. $\quad left \leftarrow subTree_table[getlevel(N_i)]$
11. $\quad right \leftarrow current_hash$
12. $\quad current_hash \leftarrow h(left, right)$
13. $\quad N_i \leftarrow getParentNode(N_i)$
14. $subTree_table[getlevel(N_i)] \leftarrow current_hash$ /** SMT_{onchain} 追加写入**/
15. $newBlock \leftarrow creatBlock()$
16. $newAccount \leftarrow newBlock.createContractAccount()$;
17. 迭代计算修改的子树根哈希值 $current_hash$ 与子树所在层数 $level$
18. **for** $b \in account.SMT_{\text{onchain}}$ **do**
19. $\quad$ if b is not the $subTree$ of $level$
20. $\quad\quad newAccount.createIndex(b)$

© 中国科学院软件研究所 http://www.jos.org.cn

21. *newAccount.add*(*current_hash*)
22. *blockChain.append*(*newBlock*)

图 7 和图 8 展示了 *level*=3 的 SMT 插入 $\langle l_1,l_2,l_3,l_4,l_5\rangle$ 后的结构, 首先数据拥有者向 SMT 追加写入 $\langle l_1,l_2,l_3,l_4\rangle$, 上述操作导致 $SMT_{onchain}$ 数据和 $SMT_{offchain}$ 的结构变化. 由于 $\langle l_1,l_2,l_3,l_4\rangle$ 追加写入时会更新 $SMT_{onchain}$ 所有子树根哈希值, 因此区块 *N* 的状态树创建了所有子树的根节点哈希值. $SMT_{offchain}$ 的结构以 h_3 为根节点的树形结构, 其中第 0 层子树根哈希为 $h(l_3)$. 后续数据拥有者向 SMT 树中写入 l_5, 当前 SMT 最大容量为 4 执行扩容操作, $SMT_{onchain}$ 在区块 *N*+1 中创建第 3 层子树根节点哈希值 h_4, $SMT_{offchain}$ 将 $subTree_3$ 的元数据追加写入 *subTree_table*. 由于插入节点位于 SMT 中第 5 个叶节点, 因此云服务提供商更新 $SMT_{offchain}$ 中 *subTree_table* 第 0 层子树根节点 $h(l_3)$ 为 $h(l_5)$, 区块链在区块 *N*+1 创建第 0 层子树根节点 $h_1 = h(l_5)$, 并构建索引指针指向区块 *N* 中不变的第 2 层和第 3 层子树根节点 $\langle h_2,h_3\rangle$.

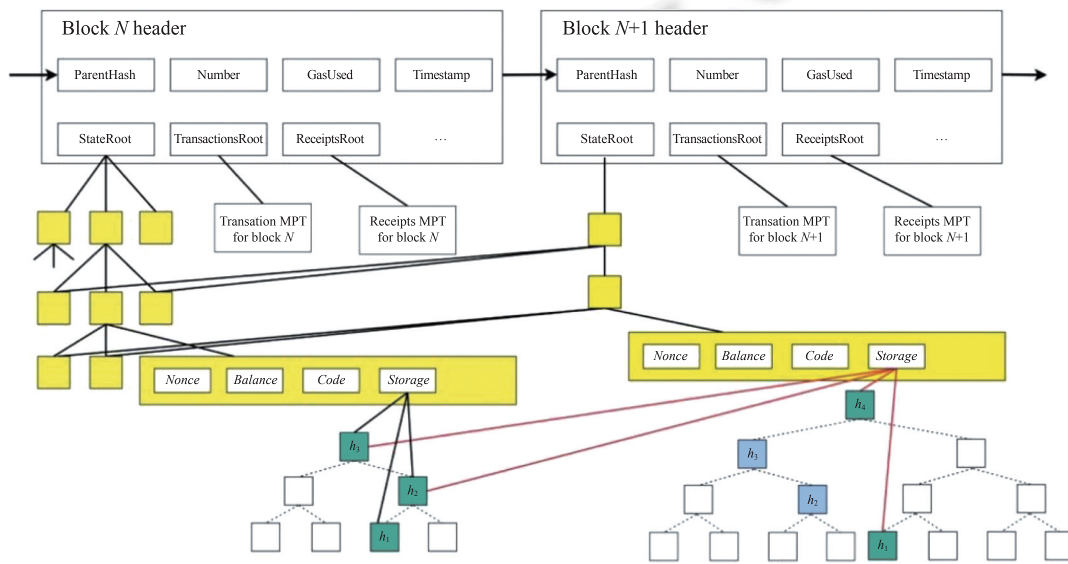
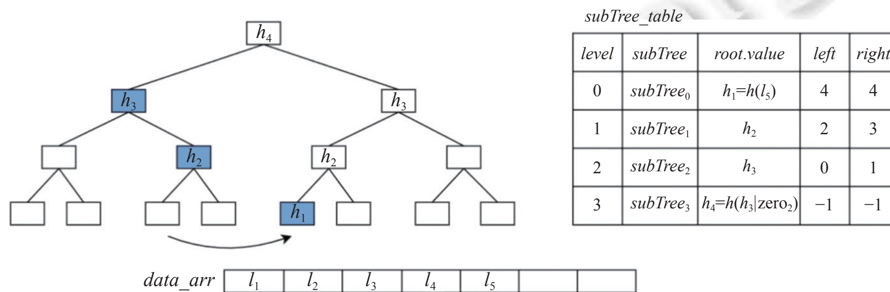


图 7 流数据追加写入 $SMT_{onchain}$ 示意图



level	subTree	root.value	left	right
0	$subTree_0$	$h_1=h(l_5)$	4	4
1	$subTree_1$	h_2	2	3
2	$subTree_2$	h_3	0	1
3	$subTree_3$	$h_4=h(h_3\|zero_2)$	−1	−1

图 8 流数据追加写入 $SMT_{offchain}$ 示意图

● 云服务提供商执行可验证查询. 算法 3–6 描述了云服务提供商执行可验证查询的过程, 云服务提供商收到用户的查询请求后 $Q = [min,max]$, 检索本地 $SMT_{offchain}$ 得到查询结果与验证对象 $\langle R,VO_{sp}\rangle$. $SMT_{offchain}$ 包含多个子树, 每个子树中都可能包含查询结果, 云服务提供商调用 *subTreeRangeQuery* 在各子树中查询结果 r_i 并生成验证对象 vo_i, 将 $\langle r_i,vo_i\rangle$ 汇总 $\langle R,VO_{sp}\rangle$ 返回给用户.

算法 3. 计算各子树边界 ($getSubTreesBoundary$).

Input: Layer of the subtree $level$, the number of data $index$, the boundaries of index l, r;
Output: subtrees boundaries B.

1. $cap = 2^{level-1}$
2. **if** $level == 0$ **return**
2. **if** $index < cap$ **then**
3. $B[level] \leftarrow [-1,-1]$
4. $getSubTreesBoundary(level-1, index, l, r)$
5. **else if** $index == cap$ **then**
6. $B[level] \leftarrow [r, r]$
7. $getSubTreesBoundary(level-1, index-1, l, r-1)$
8. **else if** $cap < index < 1.5cap$ **then**
9. **if** $B[level-1] = \phi$ **then** $getSubTreesBoundary(level-1, index-cap/2, l+cap/2, r)$
10. **for** b_i in B and $level$ less than current $level$ **do** $[l, r] = [l, r] - b_i$
11. $B[level] \leftarrow [l, r]$
12. **else if** $index \geqslant 1.5cap$ **then**
13 $B[level] \leftarrow [l, r]$
14. $getSubTreesBoundary(level-1, index-cap, l+cap, r)$
15. **return** the static variable B

算法 4. 云服务器通过 SMT 实现可验证查询检索 ($processQueryBySp$).

Input: Query rang $Q = [min, max]$, the subtree table in SMT_{offchain} $subTree_table$;
Output: Query result R, Verify object VO_{sp}.

1. **for** $subTree_i \in subTree_table$ **do**
2. $\langle r_i, vo_i \rangle \leftarrow subTreeRangeQuery(Q, subTree_i)$
3. Append vo_i to VO_{sp}
4. Append r_i to R
5. **return** $\langle R, VO_{sp} \rangle$

算法 5. 子树中查询结果与生成验证对象 ($subTreeRangeQuery$).

Input: Query rang $Q = [min, max]$, the subtree of MHT $subTree_i$;
Output: Query result r_i, Verify object vo_i.

1. **if** $\phi = [min, max] \cap [subTree_i.left, subTree_i.right]$ **then**
2. **return** $\langle \phi, subTree.root.value \rangle$
3. **return** $getVoByRecursion(Q, subTree_i.left, subTree_i.right)$

算法 6. 通过递归查询结果与生成验证对象 ($getVoByRecursion$).

Input: Query rang $Q = [min, max]$, the boundaries of Query $left, right$;
Output: Query result r_i, Verify object vo_i.

© 中国科学院软件研究所 http://www.jos.org.cn

1. $mid = (left + right)/2$
2. **if** $\phi = [min, max] \cap [left, right]$ **then**
3. $vo_i = processHashFunc(left, right)$
4. **else if** $[min, max] \subseteq [left, right]$ **then**
5. $r_l, vo_l = getVoByRecursion(Q, left, mid)$
6. $r_r, vo_r = getVoByRecursion(Q, mid+1, right)$
7. Append r_l, r_r to r_i
8. Append vo_l, vo_r to vo_i
9. **else if** $\phi \neq [min, max] \cap [left, mid]$ **then**
10. $r_l, vo_l = getVoByRecursion(Q, left, mid)$
11. $vo_r = processHashFunc(mid+1, right)$
12. Append r_l to r_i
13. Append vo_l, vo_r to vo_i
14. **else if** $\phi \neq [min, max] \cap [mid+1, right]$ **then**
15. $vo_l = processHashFunc(left, mid)$
16. $r_r, vo_r = getVoByRecursion(Q, mid+1, right)$
17. Append r_r to r_i
18. Append vo_l, vo_r to vo_i
19. **return** $\langle r_i, vo_i \rangle$

子树集合表示了 SMT_{offchain}, 由于二叉树的每层都存在唯一子树根节点, 层数较高的子树可能包含层数较低的子树, SMT_{offchain} 通过 $\langle left, right \rangle$ 标识了各子树查询边界. 算法 3 描述了 SMT_{offchain} 各子树边界的计算过程, 算法输入参数为层数 $level$、元素个数 $number$, 递归求各子树边界的整体流程如下: 若 $number$ 小于层数为 $level$ 的二叉树容量, 说明当前子树不包含任何元素, 边界为空 $[-1, -1]$ (步骤 2–4), 若 $number$ 等于二叉树容量, 说明当前子树边界只包含最后一个元素 (步骤 5–7), 若 $level$ 大于二叉树容量但小于 1.5 倍二叉树容量, 说明当前子树边界存在并可通过更低层的子树边界计算得到 (步骤 8–11), 若 $number$ 大于等于二叉树容量的 1.5 倍则说明当前子树包含当前所有元素 (步骤 12–14). 通过上述递归过程可求 SMT_{offchain} 各层子树边界.

表 3 展示了图 8 中 SMT_{offchain} 各子树边界的计算实例, 首先计算的第 3 层子树的子树边界, 由于 cap 小于 number, 第 3 层边界为 $\langle -1, -1 \rangle$, 递归计算第 2 层子树边界, 由于 $number$ 大于 cap 并且小于 1.5cap, 递归计算第 1 层子树边界, 由于 $number$ 等于 1.5 倍 cap, 设置第 1 层二叉树边界为 $\langle 2, 3 \rangle$, 递归计算第 0 层子树边界, 由于 cap 等于 $number$, 设置第 0 层子树边界为 $\langle 4, 4 \rangle$, 最后根据第 0 层和第 1 层子树边界计算得到第 2 层子树边界 $\langle 0, 1 \rangle$.

表 3 子树边界计算实例

$level$	cap	$number$	$[l, r]$	$\langle left, right \rangle$
3	8	5	[0, 4]	$\langle -1, -1 \rangle$
2	4	5	[0, 4]	$\langle 0, 1 \rangle$
1	2	3	[2, 4]	$\langle 2, 3 \rangle$
0	1	1	[4, 4]	$\langle 4, 4 \rangle$

subTreeRangeQuery 执行时判断当前查询范围与子树边界是否存在交集, 若不存在直接返回子树根节点值, 存在则调用 *getVoByRecursion* 算法执行递归查询. *getVoByRecursion* 通过 $\langle left, right \rangle$ 描述了每次递归过程中的查询边界, 若查询边界与查询范围不相交则通过查询边界构建验证元素 (步骤 2–3). 若查询边界包含查询范围则将查

© 中国科学院软件研究所 http://www.jos.org.cn

询边界平均分为左右边界, 递归调用 *getVoByRecursion* 返回结果 (步骤 4–8). 若左边界与查询范围相交则通过左边界递归得到左半区的查询结果, 并通过右边界生成右半区的验证元素 (步骤 9–13). 若右边界与查询范围相交则通过右边界递归得到右半区的查询结果, 并通过左边界生成左半区的验证元素 (步骤 14–18). 由于 SMT 只存储各子树根哈希值, 调用 *processHashFunc* 生成验证元素时, 将查询边界中的元素构建为一棵完整的 MHT 并返回根节点值.

以图 8 中 SMT_{offchain} 进行可验证查询事例说明, 用户向云服务提供商发起查询 $Q=[1,4]$, 云服务提供商在各层子树中查询数据并生成验证对象. 由于第 3 层子树边界 $\phi=[-1,-1]\cap Q$, 第 3 层查询结果与验证对象为 $\langle\phi,h_4\rangle$, 第 2 层子树返回 $\langle l_2,h(l_1)\rangle$, 第 1 层子树返回 $\langle(l_3,l_4),\phi\rangle$, 第 0 层子树返回 $\langle l_5,\phi\rangle$. 合并所有查询结果与验证对象, 返回 $\{\langle\phi,l_2,(l_3,l_4),l_5\rangle,\langle h_4,h(l_1),\phi,\phi\rangle\}$ 给用户.

● 用户验证查询结果. 算法 7 描述了算法的整体流程, 用户首先调用智能合约获取区块链上 SMT_{onchain} 各子树的根哈希值 VO_{chain}, 再分别验证各子树返回结果的正确性与完整性. *subTreeVerify* 验证的过程与传统 MHT 相同, 用户通过查询结果 r_i 与 vo_i 中的兄弟节点哈希值或相邻的非叶子节点哈希值, 重新构建二叉树的根哈希值. 如果重新构建的根哈希值与 VO_{chain} 中的哈希值相同则通过验证.

算法 7. 用户验证查询结果 (*verifyByclient*).

Input: VO_{sp} from the *SP*, VO_{chain} from the blockchain, Query result *R*;
Output: Whether the verification is passed.

1. **for** $\langle vo_i,r_i\rangle$ in VO_{sp}, *R* **do**
2. $T_i\leftarrow$subtree root value from VO_{chain}
3. $state\leftarrow subTreeVerify(vo_i,r_i,T_i)$
4. **if** $state=false$ **return** false
5. **return** true

以图 8 中 SMT_{offchain} 对 $Q=[1,4]$ 查询结果的验证进行说明, 用户获取区块链上最新区块中存储的各层子树根哈希值得到 $VO_{\text{chain}}=\{h_1',h_2',h_3',h_4'\}$, 对云服务提供商返回的查询结果进行验证. 第 0 层子树验证 $h_1'=h(l_5)$, 第 1 层子树验证 $h_2'=h(l_3|l_4)$, 第 2 层子树验证 $h_3'=h\left(h(h(l_1)|l_2)|h_2'\right)$, 第 3 层子树不存在查询结果, 因此不需验证.

3.4 SMT 的安全性

本节分析了 SMT 方案的在查询时的安全性, SMT 的安全性可规约为构建 SMT 时使用的哈希函数的安全性. 在证明 SMT 安全性时, 我们假设攻击者能够能攻破 SMT 的安全性, 及攻击者伪造或篡改的查询结果可以通过用户验证. 如果 SMT 满足以下性质则 SMT 是安全的, 其中 *D* 为查询范围, VO_{chain} 为根据查询范围在区块链上得到的验证对象, $\langle R,VO_{sp}\rangle$ 为攻击者返回的查询结果与验证对象, $Q(D)$ 为正确的查询结果, VO_{right} 为云服务提供商返回的正确验证对象.

● 攻击模型. 下面通过定义一个挑战者 *C* 和攻击者 *A* 之间的交互式游戏 $G_A^{\text{SMT}}(k)$, 这里 *k* 为安全参数. 挑战者 *C* 随机生成一个查询范围 *D* 将 *D* 转发给攻击者 *A*, 并通过智能合约获取区块链上与 *D* 相对应的验证对象 VO_{chain}. 攻击者 *A* 检索云服务提供商中存储的数据得到 $\langle Q(D),VO_{\text{right}}\rangle$, 篡改正确的结果得到 $\langle R,VO_{\text{chain}}\rangle$ 并返回挑战者. 挑战者根据 $\langle R,VO_{sp},VO_{\text{chain}}\rangle$ 验证结果, 对于 SMT 方案查询的安全性假设形式化的定义如下.

1) *A* 返回的 *R* 中存在不属于 $Q(D)$ 的对象时, *A* 使用 $\langle R,VO_{sp},VO_{\text{chain}}\rangle$ 验证时无法通过.

2) *A* 返回的 *R* 中缺少有效查询结果时, *A* 使用 $\langle R,VO_{sp},VO_{\text{chain}}\rangle$ 验证时无法通过.

● 建立阶段. 运行 *initSMT* 算法初始化 SMT 并调用 *insert* 算法将测试数据集合插入到 SMT 中, 挑战者 *C* 可以获取维护在区块链上的完整数据. 攻击者 *A* 可以获取云服务提供商维护的完整数据.

● 查询问询. 挑战者生成查询范围 D_i $(1\leqslant i\leqslant z)$ 并发送给攻击者 *A*, 并获取区块链上与 D_i 对应的验证对象

© 中国科学院软件研究所 http://www.jos.org.cn

VO^i_{chain}. 攻击者 A 调用 $processQueryBySp(D_i)$ 得到查询结果 $\langle Q(D_i), VO^i_{\text{right}}\rangle$, 篡改得到 $\langle R_i, VO^i_{sp}\rangle$ 并发送给挑战者 C. 这里用 $P := \{(Q(D_i), R_i, VO^i_{sp}, VO^i_{\text{chain}}), (Q(D_{z(k)}), R_{z(k)}, VO^{z(k)}_{sp}, VO^{z(k)}_{\text{chain}})\}$ 表示一串问询应答的结果.

● 输出阶段. 如果攻击者 A 输出的查询结果能通过挑战者 C 的验证, 则认为攻击者 A 获胜, 否则挑战者 C 获胜, A 输出的查询结果可分为以下两种情况.

(1) 情况 1: $\{r_j | r_j \notin Q(D_i) \wedge r_j \in R_i\} \neq \phi, R_i$ 中存在不属于真正结果的元素.

(2) 情况 2: $\{r_j | r_j \in Q(D_i) \wedge r_j \notin R_i\} \neq \phi, R_i$ 中缺少一个有效的答案.

攻击者 A 赢得上述游戏的概论表示为 Adv_A^{SMT}, 若对于任意 PPT 攻击者 A 赢得上述游戏的优势 Adv_A^{SMT} 可以忽略不计, 则 $SMT(processQueryBySp, verifyByClinet)$ 可验证查询是安全的.

证明: 当 A 输出的查询结果属于情况 1 时, 表明 R_i 中存在不属于真正结果的元素, 及攻击者 A 对返回的结果进行了篡改. 由于挑战者需要通过 $\langle R_i, VO^i_{sp}\rangle$ 重新构建 r_j 对应子树的根哈希值, 并将结果与 VO^i_{chain} 中根哈希值进行比较, 这样的篡改结果意味着存储在两个 SMT 子树具有不同的元素但有相同的根哈希值, 这意味着构建 SMT 使用的哈希函数碰撞成功, 这与我们的假设矛盾, 在使用抗碰撞哈希构建 SMT 时, Adv_A^{SMT} 可忽略.

当 A 输出的查询结果属于情况 2 时, 表明 R_i 中缺少一个有效的答案, 由于挑战者会通过 SMT 整个子树或子树中除查询范围外的边界节点来验证查询结果的完整性 (重新构建 SMT), 缺失有效的答案必将导致与 SMT 子树的哈希碰撞, 这与我们假设相反, 此时 Adv_A^{SMT} 可忽略. 综上, SMT 方案查询的安全性可归约为构建 SMT 时使用的哈希函数的抗碰撞性, 当所采用的哈希函数抗碰撞时, 敌手的优势可忽略, SMT 方案查询具有安全性.

3.5 SMT 的成本分析

本节分析了流数据为 n 块时, SMT 在区块链上维护成本与执行可验证查询的 gas 消耗, 其中维护成本包括流数据更新时 SMT 的更新成本与存储成本.

(1) 流数据更新成本

流数据更新时, 由于 SMT 的特殊性, 插入元素的开销和插入元素的具体位置相关, 本文分析了插入元素时的最低开销、最高开销、平均开销.

最低开销分析如下, 如图 8 插入 $h(l_3)$ 元素, $h(l_3)$ 元素本身为左孩子节点插入算法得到, 只需要更新 $subTree_0.root.value = h(l_3)$, 开销成本如下:

$$C_{\text{SMT}}^{\text{insert_min}} = C_{\text{update}}.$$

最高开销分析如下, 如图 8 插入 $h(l_4)$ 元素, $h(l_4)$ 元素为右孩子节点, 需要加载当前层子树根哈希值 $h(l_3)$ 值并计算得到 h_2, 由于 h_2 为右孩子节点通过相同的计算得到 h_3, h_3 为右孩子节点迭代操作结束, 更新 $subTree_2.root.value = h_3$, 开销成本如下:

$$C_{\text{SMT}}^{\text{insert_max}} = (\log_2 n + 1) \times (C_{\text{load}} + C_{\text{hash}}) + C_{\text{update}}.$$

平均开销分析如下, 由于插入元素的成本由 SMT 中插入元素的位置确定, 难以直接得到插入元素的平均开销, 本文首先分析从插入第 1 个数据到插入第 n 的数据的整体开销再计算平均开销. 插入元素时只需更新一次 $subTree_i.root.value$ 的值, 不同点在于从区块链中加载元素个数与哈希运算的次数, 并且加载元素与哈希运算的次数相同. 设 a_i 为层数为 i 的 SMT 插入所有元素 (元素个数为 2^{i-1}) 的哈希运算次数, 其中 $a_1 = 0$, $a_2 = 2$, $a_3 = 5$ 以此类推, a_i 与 a_{i-1} 构成如下递推公式:

$$\begin{cases} a_i = 2a_{i-1} + 1 \\ a_1 = 0 \end{cases}.$$

由递推公式推导, 并代入 $a_1 = 0$ 后 a_i 表达式:

$$a_i = \begin{cases} \sum_{k=2}^{i-1} 2^k + 1, & i > 1 \\ 0, & i = 1 \end{cases}.$$

推导在元素个数为 n 及 SMT 层数为 $\log_2 n + 1$ 的 SMT 中插入所有元素的 gas 开销为:

© 中国科学院软件研究所 http://www.jos.org.cn

$$C_{\text{SMT}}^{\text{insert_sum}}=\begin{cases}n\times C_{\text{update}}+(C_{\text{load}}+C_{\text{hash}})\times a_{\log_2 n+1}, & n>1\\ n\times C_{\text{update}}, & n=1\end{cases}.$$

将 a_i 代入后结果如下:

$$C_{\text{SMT}}^{\text{insert_sum}}=\begin{cases}\left(C_{\text{load}}+C_{\text{hash}}+C_{\text{update}}\right)\times n, & n>1\\ n\times C_{\text{update}}, & n=1\end{cases}.$$

通过上述分析得到查询元素的平局开销如下:

$$C_{\text{SMT}}^{\text{insert_avg}}=\begin{cases}C_{\text{load}}+C_{\text{hash}}+C_{\text{update}}, & n>1\\ C_{\text{update}}, & n=1\end{cases}.$$

(2) 存储开销

如图 8 所示, 区块链上 SMT_{onchain} 存储了各子树的根哈希值, 而子树的个数等于 SMT_{onchain} 的高度, 所以存储开销如下:

$$C_{\text{SMT}}^{\text{store}}=\lceil\log_2 n\rceil\times C_{\text{store}}.$$

(3) 验证开销

用户验证从云服务提供返回的查询结果时, 需要通过智能合约加载区块链中存储的各子树根哈希值 VO_{chain}, 由于子树的个数为 SMT 的高度所以验证开销为:

$$C_{\text{SMT}}^{\text{store}}=\lceil\log_2 n\rceil\times C_{\text{load}}.$$

4 实验分析

本节通过实验评估了本文提出的新型 ADS 认证数据结构 SMT 的各项指标性能, 并实现了基于 MHT 的 ADS 方案, 通过对比两种算法在区块链上的维护成本、云服务器实现可验证查询检索的时间、生成验证对象 $VO=VO_{sp}+VO_{\text{chain}}$ 的大小与用户验证查询结果的时间, 最终验证了 SMT 算法的高效性.

本文通过软件模拟了数据拥有者 (物联网设备) 采集流式数据的过程, 总共模拟了 2^{19} 个数据块, 每个数据块的大小为 1 MB. 选择以太坊作为区块链平台, 通过 Solidity 语言实现了 SMT 在区块链上相关功能, 选择华为云服务器作为云服务提供商, 使用 Inter Core i5-10400 4.3 GHz CPU 与 16 GB RAM 搭载 Ubuntu 18.04.1 LTS 操作系统的台式机器作为用户, 选择 SHA-3 作为加密哈希函数.

图 9 描述了物联网设备将不同大小的数据集合追加写入 SMT 与 MHT 维护 ADS 认证数据结构的总 gas 消耗. 随着物联网设备采集数据的增加, SMT 与 MHT 总 gas 消耗随之增加, 但 SMT 所消耗的 gas 远低于 MHT, 实验结果表明 SMT 在区块链上具备低 gas 消耗的特性. 导致上述情况的原因如下, 首先 SMT 在区块链上只需存储各子树的根哈希值而 MHT 需要存储完整的树结构, 同时区块链上存储操作 C_{store} 需要消耗最高的 gas. 其次虽然 MHT 与 SMT 的树高度随流数据规模的增加, 成同等速率的增加, 但 SMT 在更新时只需更新 1 个子树的根哈希值并平均执行 1 次哈希运算, 而 MHT 需要更新插入节点到根节点路径上 $\log_2 n+1$ 个节点信息, 同时执行 $\log_2 n$ 次哈希运算.

图 10 描述了在初始树高度为 8 层时, MHT 与 SMT 追加写入前 128 块数据时, 每次操作的 gas 消耗. SMT 每次追加更新消耗的 gas 远低于 MHT, 这表明了流数据更新时, SMT 在区块链上动态维护的低 gas 消耗特性. 导致上述情况的原因如下, MHT 由于每次插入操作所需修改的节点数量与树的层高成线性关系, 在层高为 8 层时每次消耗 gas 基本不变. 由于第 128 块数据所在叶节点与根节点组成路径上所有节点都为右孩子节点, 根据 SMT 插入算法插入第 128 块消耗的成本最高, 但 SMT 相比与 MHT 所需更新节点个数与哈希运算次数更少.

图 11 与图 12 描述了在不同大小的数据集合下, 模拟了 1 000 次查询并且每次查询占总数据量 0.1% 的情况下, 返回的验证对象 VO 的平均大小与验证查询结果的平均时间. 数据集合大小成指数变化, SMT 与 MHT 返回的 VO 大小与用户验证查询结果的时间成线性递增, 但 SMT 返回的 VO 大小与验证查询结果的平均时间总小于 MHT, 结合图 9 与图 10 的实验结果表明了 SMT 能大幅度降低区块链上 ADS 的维护成本的同时还能提高验证查询的效率.

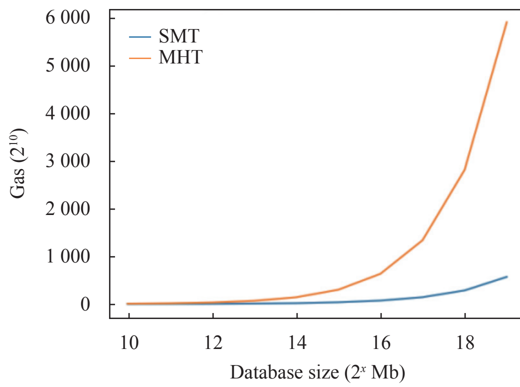


图 9　Gas 消耗与数据库大小关系

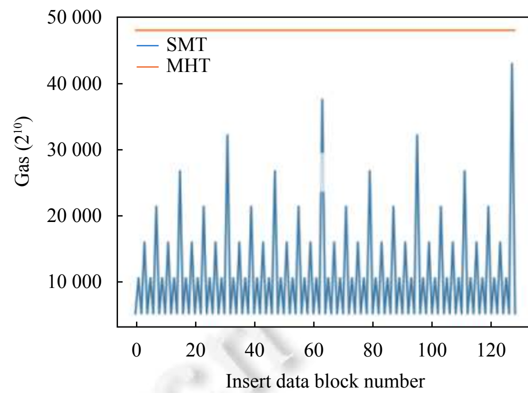


图 10　Gas 消耗与数据块数量关系

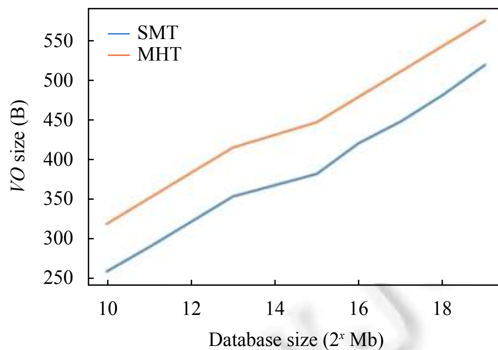


图 11　*VO* 大小与数据库大小关系

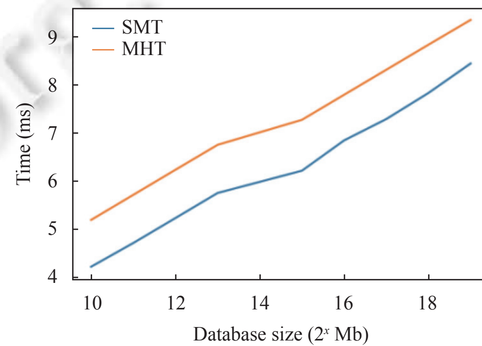


图 12　验证时间与数据库大小关系

5 总　结

本文在基于区块链的可信存储问题上研究了流数据的可验证查询相关问题, 由于流数据是没有边界大小的追加数据, 而传统认证数据结构如 MHT 是针对静态数据的动态操作, 不能够用于对数据流的动态操作与实时验证, 并且传统认证数据结构在区块链上维护成本过高,

解决上述问题的关键在于如何设计一种由区块链高效维护的新型 ADS. 本文通过分析以太坊智能合约的 gas 消耗机制与基于传统 MHT 的 ADS 的 gas 开销, 提出了 SMT 这一种新型 ADS, SMT 通过设计基于子树的独立数据验证查询机制, 能显著降低智能合约的存储与计算成本, 同时支持流数据查询结果的高效验证. 最后通过安全性分析证明了 SMT 的安全性, 基于以太坊智能合约对 SMT 模型的进行 gas 开销、验证对象大小及验证耗时这 3 个维度进行实验数据采集, 通过与理论值对比分析, 验证了 SMT 在区块链应用场景下相较 MHT 的高效性.

References:

[1] Hacigumus H, Iyer B, Mehrotra M. Providing database as a service. In: Proc. of the 18th Int'l Conf. on Data Engineering. San Jose: IEEE, 2002. 29–38. [doi: 10.1109/ICDE.2002.994695]

[2] Hu QC. Research on integrity checking scheme in outsourced storage environment [MS. Thesis]. Chengdu: University of Electronic Science and Technology of China, 2010 (in Chinese with English abstract).

[3] Zhang J, Tu XD. Research on integrity checking of data based on outsourced storage. Modern Science & Technology of Telecommunications, 2009, 39(8): 41–45 (in Chinese with English abstract). [doi: 10.3969/j.issn.1002-5316.2009.08.011]

[4] Li FF, Hadjieleftheriou M, Kollios G, Reyzin L. Dynamic authenticated index structures for outsourced databases. In: Proc. of the 2006 ACM SIGMOD Int'l Conf. on Management of Data. Chicago: Association for Computing Machinery, 2006. 121–132. [doi: 10.1145/1142473.1142488]

[5] Jiang T. Research on the key Technologies of data security in cloud storage [Ph.D. Thesis]. Xi'an: Xidian University, 2016 (in Chinese

with English abstract).
[6] Merkle RC. A certified digital signature. In: Brassard G, ed. Proc. of the 1990 Advances in Cryptology. New York: Springer, 1990. 218–238. [doi: 10.1007/0-387-34805-0_21]
[7] Parno B, Howell J, Gentry C, Raykova M. Pinocchio: Nearly practical verifiable computation. Communications of the ACM, 2016, 59(2): 103–112. [doi: 10.1145/2856449]
[8] Ben-Sasson E, Chiesa A, Tromer E, Virza M. Succinct non-interactive zero knowledge for a von Neumann architecture. In: Proc. of the 23rd USENIX Conf. on Security Symp. San Diego: USENIX Association, 2014. 781–796.
[9] Zhang YP, Genkin D, Katz J, Papadopoulos D, Papamanthou C. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In: Proc. of the 2017 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2017. 863–880. [doi: 10.1109/SP.2017.43]
[10] Mykletun E, Narasimha M, Tsudik G. Authentication and integrity in outsourced databases. ACM Trans. on Storage, 2006, 2(2): 107–138. [doi: 10.1145/1149976.1149977]
[11] Narasimha M, Tsudik G. DSAC: Integrity for outsourced databases with signature aggregation and chaining. In: Proc. of the 14th ACM Int’l Conf. on Information and Knowledge Management. Bremen: Association for Computing Machinery, 2005. 235–236. [doi: 10.1145/1099554.1099604]
[12] Pang H, Jain A, Ramamritham K, Tan KL. Verifying completeness of relational query results in data publishing. In: Proc. of the 2005 ACM SIGMOD Int’l Conf. on Management of Data. Baltimore: Association for Computing Machinery, 2005. 407–418. [doi: 10.1145/1066157.1066204]
[13] Pang HH, Tan KL. Authenticating query results in edge computing. In: Proc. of the 20th Int’l Conf. on Data Engineering. Boston: IEEE, 2004. 560–571. [doi: 10.1109/ICDE.2004.1320027]
[14] Yang Y, Papadopoulos S, Papadias D, Kollios G. Authenticated indexing for outsourced spatial databases. The VLDB Journal, 2009, 18(3): 631–648. [doi: 10.1007/s00778-008-0113-2]
[15] Chen Q, Hu HB, Xu JL. Authenticated online data integration services. In: Proc. of the 2015 ACM SIGMOD Int’l Conf. on Management of Data. Melbourne: Association for Computing Machinery, 2015. 167–181. [doi: 10.1145/2723372.2747649]
[16] Papamanthou C, Shi E, Tamassia R, Yi K. Streaming authenticated data structures. In: Proc. of the 32nd Annual Int’l Conf. on the Theory and Applications of Cryptographic Techniques. Athens: Springer, 2013. 353–370. [doi: 10.1007/978-3-642-38348-9_22]
[17] Papadopoulos S, Yang Y, Papadias D. Continuous authentication on relational streams. The VLDB Journal, 2010, 19(2): 161–180. [doi: 10.1007/s00778-009-0145-2]
[18] Xu C, Zhang C, Xu JL. vChain: Enabling verifiable boolean range queries over blockchain databases. In: Proc. of the 2019 Int’l Conf. on Management of Data. Amsterdam: Association for Computing Machinery, 2019. 141–158. [doi: 10.1145/3299869.3300083]
[19] Zhang C, Xu C, Xu JL, Tang YZ, Choi B. GEM^2-tree: A gas-efficient structure for authenticated range queries in blockchain. In: Proc. of the 35th Int’l Conf. on Data Engineering (ICDE). Macao: IEEE, 2019. 842–853. [doi: 10.1109/ICDE.2019.00080]
[20] Xia Q, Sifah EB, Agyekum KOBO, Xia H, Acheampong KN, Smahi A, Gao JB, Du XJ, Guizani M. Secured fine-grained selective access to outsourced cloud data in IoT environments. IEEE Internet of Things Journal, 2019, 6(6): 10749–10762. [doi: 10.1109/JIOT.2019.2941638]
[21] Asamoah KO, Xia H, Amofa S, Amankona OI, Luo KC, Xia Q, Gao JB, Du XJ, Guizani M. Zero-chain: A blockchain-based identity for digital city operating system. IEEE Internet of Things Journal, 2020, 7(10): 10336–10346. [doi: 10.1109/JIOT.2020.2986367]
[22] Xia Q, Gao JB, Xia H, Zhou T, Zhang XS. Blockchain data sovereignty technology and its applications. Journal of University of Electronic Science and Technology of China (Social Science Edition), 2020, 22(1): 5–11 (in Chinese with English abstract). [doi: 10.14071/j.1008-8105(2020)-1002]
[23] Cai T, Lin H, Chen WH, Zheng ZB, Yu Y. Efficient blockchain-empowered data sharing incentive scheme for Internet of Things. Ruan Jian Xue Bao/Journal of Software, 2021, 32(4): 953–972 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6229.htm [doi: 10.13328/j.cnki.jos.006229]
[24] Wei X, Wang XY, Yu Z, Guo SY, Qiu XS. Cross domain authentication for IoT based on consortium blockchain. Ruan Jian Xue Bao/Journal of Software, 2021, 32(8): 2613–2628 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6033.htm [doi: 10.13328/j.cnki.jos.006033]
[25] Merkle RC. A certified digital signature. In: Brassard G, ed. Proc. of the 2001 Advances in Cryptology. New York: Springer, 2001. 218–238.

附中文参考文献:

[2] 胡钦超. 外包存储环境下完整性检测方案的研究 [硕士学位论文]. 成都: 电子科技大学, 2010.

© 中国科学院软件研究所 http://www.jos.org.cn

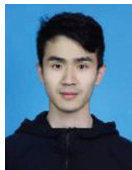
[3] 张佳, 涂晓东. 基于外包存储的数据完整性检验的研究. 现代电信科技, 2009, 39(8): 41–45. [doi: 10.3969/j.issn.1002-5316.2009.08.011]

[5] 姜涛. 云存储中数据安全关键技术研究 [博士学位论文]. 西安: 西安电子科技大学, 2016.

[22] 夏琦, 高建彬, 夏虎, 周涛, 张小松. 区块链数据主权技术与应用. 电子科技大学学报(社科版), 2020, 22(1): 5–11. [doi: 10.14071/j.1008-8105(2020)-1002]

[23] 蔡婷, 林晖, 陈武辉, 郑子彬, 余阳. 区块链赋能的高效物联网数据激励共享方案. 软件学报, 2021, 32(4): 953–972. http://www.jos.org.cn/1000-9825/6229.htm [doi: 10.13328/j.cnki.jos.006229]

[24] 魏欣, 王心妍, 于卓, 郭少勇, 邱雪松. 基于联盟链的物联网跨域认证. 软件学报, 2021, 32(8): 2613–2628. http://www.jos.org.cn/1000-9825/6033.htm [doi: 10.13328/j.cnki.jos.006033]



孙钰山(1996－), 男, 硕士生, 主要研究领域为区块链, 网络空间安全.

夏琦(1979－), 女, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为网络数据安全, 区块链理论及应用.

杨靖聪(1997－), 男, 硕士生, 主要研究领域为区块链, 网络空间安全.

高建彬(1976－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为区块链技术, 大数据安全.

www.jos.org.cn
www.jos.org.cn