Wang等人. 为云计算中的存储安全提供公开验证和数据动态

Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing

Qian Wang[1], Cong Wang[1], Jin Li[1], Kui Ren[1], and Wenjing Lou[2]

[1] Illinois Institute of Technology, Chicago IL 60616, USA
{qwang,cwang,jin.li,kren}@ece.iit.edu
[2] Worcester Polytechnic Institute, Worcester MA 01609, USA
wjlou@ece.wpi.edu

摘要

- 引入了第三方审计者（TPA）的概念，TPA代表云存储用户来验证存储在云中的动态数据的完整性。消除审计过程中用户的参与，实现云计算的规模经济。
- 支持动态数据操作，如文件块的修改、插入和删除，对数据动态的支持，也是实现实用性的重要一步，因为云计算中的服务并不局限于归档或备份数据。

  之前的工作只能达成两者其中之一，本文同时实现了这两者。

  我们首先确定了从之前的工作中直接扩展完全动态的数据更新的困难下和潜在的安全问题，然后展示了如何在我们的方案设计中构建一个无缝集成这两个显著特征的优雅验证方案。特别地，为了实现高效的数据动态操作，我们通过操作经典的Merkel哈希树构造，改进了可检索证明模型（PoR）。广泛的安全性和性能分析表明，该方案具有高效的和可证明的安全性。

1 Introduction

两个需要关注的问题：

- 问题1：TPA应该使用无块验证
- 问题2：安全的数据动态操作。对现有的PDP和PoR方案的直接扩展，以支持数据动态操作，可能会导致安全漏洞。

我们的贡献可以总结如下：

1. 我们提出了一个具有公共可验证性的POR模型，其中无块和无状态验证同时实现。
2. 我们为所提出的PoR构造配备了支持全动态数据操作的功能，特别是支持块插入，这是大多现有方案中缺失的。
3. 我们通过具体的实施和与最先进的比较，证明了我们提出的建设的安全性，并证明了我们的方案的性能。

1.1 Related Works

1. Ateniese 等人提出的PDP模型，使用基于RSA的同态标签来审计外包数据，从而可以提供公开验证。但他们没有考虑动态数据存储的情况，而将其方案从静态数据存储直接扩展到动态情况，存在许多设计和安全问题。
2. Ateniese后续提出了一个PDP方案的动态版本，但该系统对查询的数量施加了一个先验约束，且不支持完全动态的数据操作。它只允许功能有限的非常基本的块操作，并且不支持块插入。
3. Wang等人考虑了分布式场景中的动态数据存储，所提出的挑战-响应协议既可以确定数据的正确性，也可以定位可能的错误。只考虑了对动态数据操作的部分支持。
4. Juels描述了一个PoR模型，并对他们的方案给出了一个更严格的证明。在这个模型中，抽查和纠错码用于确保档案服务系统上的数据文件的“占有性”和“可检索性”。具体来说，一些被称为“哨兵”的特殊块被随机嵌入到数据文件F中用于检测，且F会进一步被加密来保护这些特殊块的位置。然而，就像Ateniese后续提出的PDP方案的动态版本一样，客户端可以执行的查询的数量也是一个固定的先验的，并且预先计算的“哨兵”的引入阻止了实现动态数据更新的发展。此外，他们的方案也不支持公共可验证性。
5. Shacham等人设计了一种具有充分安全性证明的改进的PoR方案。他们使用由BLS签名构建的公开可验证的同态身份验证器，并在随机喻言模型下是可证明安全的。基于BLS的构造，实现了公共可检索性，证明可以聚合成一个小的身份验证器值。但作者仍然只考虑静态数据文件。
6. Erway等人是第一个探索动态可证明数据占有（PDP）的结构。他们扩展了PDP模型，以支持对存储文件的可证明更新，通过使用基于排名的认证跳表。这个方案本质上是PDP解决方案的一个完全动态的版本。特别是，为了支持更新，特别是对于块插入，他们试图消除Ateniese的PDP模型中的“标签”计算中的索引信息。为了实现这一点，在验证过程之前，他们首先使用认证跳表的数据结构对被挑战或更新块的标签信息进行验证。然而，他们的方案的效率仍存在疑问。

可以看出，虽然现有的方案是在不同的数据存储系统下提供完整性验证，但支持公共验证和数据动态的问题尚未得到充分解决。如何实现安全高效的设计，以无缝集成数据存储服务的这两个重要组件，仍然是云计算中一项具有挑战性的开放任务。

2 Problem Statement

2.1 System Model

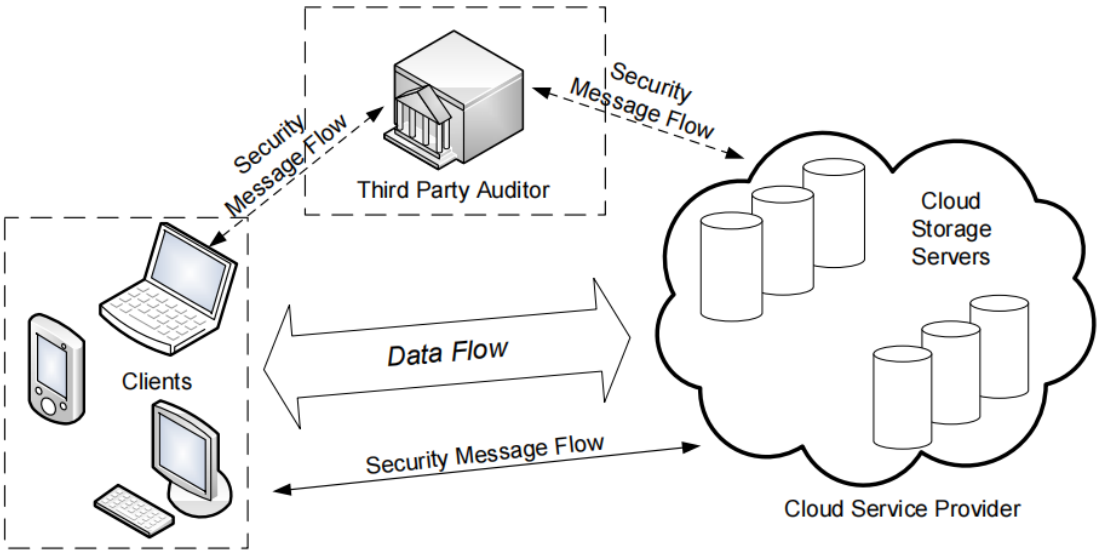


Fig. 1. Cloud data storage architecture

云数据存储的代表性网络架构如图1所示。可以识别出三个不同的网络实体：客户端：在云中存储大量数据文件并依赖云进行数据维护和计算的实体，可以是个人消费者或组织；云存储服务器（CSS）：由云服务提供商（CSP）管理的实体，拥有大量的存储空间和计算资源来维护客户端数据；第三方审计员（TPA）：TPA具有客户不具备的专业知识和能力，可以根据要求代表客户评估和揭露云存储服务的风险。

在云范例中，通过将大型数据文件放在远程服务器上，客户端可以减轻存储和计算的负担。由于客户端不再在本地拥有他们的数据，因此确保他们的数据被正确地存储和维护是至关重要的。也就是说，客户端应该配备一定的安全手段，以便即使不存在本地副本，它们也可以定期验证远程数据的正确性。如果客户不一定有时间、可行性或资源来监控他们的数据，他们可以将监视任务委托给受信任的TPA。为了保护客户端数据的隐私，执行审计时不向TPA显示原始数据文件。在本文中，我们只考虑具有公共可验证性的验证方案：任何拥有公钥的TPA都可以作为验证器。我们假设TPA是无偏的，而服务器是不可信的。请注意，我们在本文中没有讨论数据隐私问题，因为云计算中的数据隐私主题与我们在这里研究的问题是正交的。为了实现应用程序的目的，客户端可以通过CSP与云服务器进行交互，以访问或检索其预存储的数据。更重要的是，在实际场景中，客户机可能经常对数据文件执行块级操作。在本文中，我们所考虑的这些操作的最一般的形式是修改、插入和删除。

2.2 Security Model

Shacham和Waters在《Compact Proofs of Retrievability》一文中提出了PoR系统。一般来说，检查方案是安全的，如果

（1）不存在一种多项式时间算法能够以不可忽略的概率七篇验证者；

（2）存在一种多项式时间提取器，它可以通过执行多次挑战-响应协议来恢复原始数据文件。根据该PoR系统的定义，客户端可以定期挑战存储服务器，以确保云数据的正确性，通过与服务器的交互可以恢复原始文件。

他们还定义了PoR方案的正确性和稳健性：

如果验证算法在与有效的证明者（例如，服务器返回有效响应）交互时接受了，说明方案是正确的。

如果任何一个欺骗客户端它存储了文件的服务器确实存储了拥有该文件，则该方案是健壮的。

请注意，在敌手和客户端之间的“游戏”中，敌手可以完全访问存储在服务器中的信息，即，对手可以扮演证明者（服务器）的角色。在验证过程中，对手的目标是成功地欺骗客户端，即尝试生成有效的响应，并在不被发现的情况下通过数据验证。

在验证过程中，我们的安全模型与原始的PoR有细微但关键的区别。

原始的PoR方案不考虑动态数据操作，并且根本不能支持块插入。这是因为签名的构造与文件索引信息i有关。因此，一旦插入了文件块，计算开销是不可接受的，因为所有相关文件块的签名都要用新的索引重新计算。

为了处理这个限制，我们在生成签名删除索引信息i，使用$H(m_i)$作为文件块m_i的标签，而不是$H(name||i)$或者$h(v||i)$，因此，在任何文件块上的个别数据操作都不会影响其他文件块。

回想一下，$H(name||i)$或者$h(v||i)$由客户端在验证过程中生成。然而，在我们的新构造中，没有数据信息的客户端没有能力计算$H(m_i)$。为了在实现无块操作的同时成功地执行验证，服务器应该接管计算$H(m_i)$的工作，然后将其返回给证明者。

这种变化的结果将导致一个严重的问题：它将给对手更多的机会，通过操纵$H(m_i)$或m_i来欺骗证明者。由于这种构造，我们的安全模型在验证和数据更新过程中都与原始的PoR模型有所不同。具体来说，在我们的方案中，标签应该在每个协议执行中进行身份验证，而不是由验证者计算或预先存储。（详情将见第3节）。

2.3 Design Goals

我们的设计目标可以被总结如下：

1. 对存储完整性的公开验证：允许任何人，而不仅仅是最初在云服务器上存储文件的客户机，都有能力按需验证存储数据的正确性；
2. 动态数据操作支持：允许客户机对数据文件执行块级操作，同时保持相同级别的数据完整性保证。设计应尽可能高效，以确保公共可验证性和动态数据操作支持的无缝集成；
3. 无块验证：为了效率和安全性的考虑，在验证过程中，验证者（例如TPA）不应检索任何被挑战的文件块。
4. 无状态验证：在整个数据存储的长期审计之间，消除在验证者侧对状态信息进行维护的需要。

3 The Proposed Scheme

3.1 Notation and Preliminaries

Bilinear Map

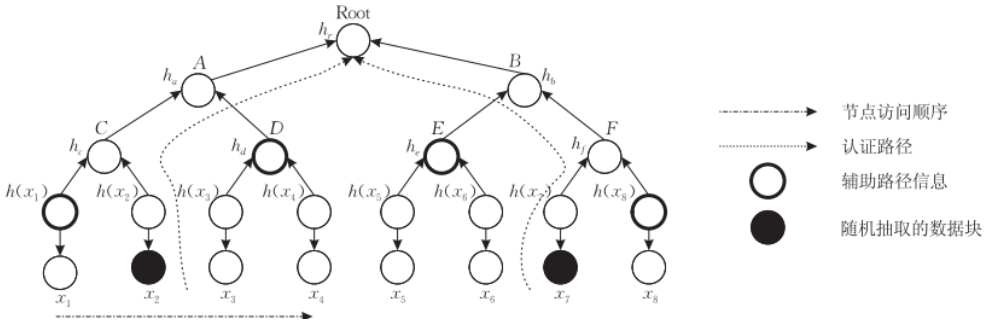
Bilinear Map. A bilinear map is a map $e : G \times G \rightarrow G_T$, where G is a Gap Diffie-Hellman (GDH) group and G_T is another multiplicative cyclic group of prime order p with the following properties [16]: (i) Computable: there exists an efficiently computable algorithm for computing e; (ii) Bilinear: for all $h_1, h_2 \in G$ and $a, b \in \mathbb{Z}_p$, $e(h_1^a, h_2^b) = e(h_1, h_2)^{ab}$; (iii) Non-degenerate: $e(g, g) \neq 1$, where g is a generator of G.

Merkle Hash Tree

默克尔哈希树是一种充分研究的身份认证结构，其目的是有效和安全地证明一组元素是没有损坏和改变的。它被构造为一个二叉树，其中MHT中的叶子是真实数据值的哈希值。虽然MHT通常用于验证数据块的值。

·下图描述了一个认证的例子。具有真实的h_r的验证者请求验证${x_2,x_7}$的完整性。

Me 叶节点值为数据块的哈希值，对叶节点的访问采用深度优先的方式进行，如图中黑虚线所示. 云服务器为了证明用户的数据是完整的，首先需要构造一条认证路径(如图中虚线所示)，及其辅助认证信息(Auxiliary Authentication Information，AAI)组成证据，返回给 TPA. TPA 根据认证路径和辅助认证信息重新计算根节点的哈希值，比对本地存储的根节点哈希值来判断数据在位置上是否是完整的. 例如，倘若 TPA 发出的挑战请求中选取块索引为{2,7}，云服务器需在返回证据{μ,σ}的同时，返回x_2认证路径{$h(x_2)$,h_c,h_a}、辅助认证路径 $\Omega=\{h(x_1),h_d\}$和 x_7的认证路径{x_7,h_f,h_b}、辅助认证路径$\Omega=\{h(x_8),h_e\}$给 TPA，TPA 重新计算 Merkle 哈希树的根节点哈希值，即根据 $h_c=h(h(x_1)\|h(x_2))$，$h_a=h(h_c\|h_d)$，$h_f=h(h(x_7)\|h(x_8))$，$h_b=h(h_e\|h_f)$，最后计算出根节点哈希值 $h_R=h(h_a\|h_b)$，对比存储在本地的根哈希值来判断数据文件是否是完整的.



但是，在本文中，我们进一步使用MHT来同时验证数据块的值和**位置**。我们把叶节点当作从左到右的序列，从而任何叶节点都可以通过遵循这个序列和在MHT中计算根节点的方法来唯一确定。

> 使用MHT确保数据节点在位置上的完整性，利用基于BLS签名的PDP机制来确保数据块内容的完整性。
> 动态更新操作时，在更新节点的同时，需返回辅助认证信息给用户，用户重新生成根节点的哈希值. 之后，更新存储在TPA中的根哈希值。

3.2 Definition

$(pk, sk) \leftarrow KeyGen(1^k)$. *This probabilistic algorithm is run by the client. It takes as input security parameter* 1^k, *and returns public key pk and private key sk.*

$(\Phi, sig_{sk}(H(R))) \leftarrow SigGen(sk, F)$. *This algorithm is run by the client. It takes as input private key sk and a file F which is an ordered collection of blocks* $\{m_i\}$, *and outputs the signature set* Φ, *which is an ordered collection of signatures* $\{\sigma_i\}$ *on* $\{m_i\}$. *It also outputs metadata-the signature* $sig_{sk}(H(R))$ *of the root R of a Merkle hash tree. In our construction, the leaf nodes of the Merkle hash tree are hashes of* $H(m_i)$.

$(P) \leftarrow GenProof(F, \Phi, chal)$. *This algorithm is run by the server. It takes as input a file* F, *its signatures* Φ, *and a challenge* $chal$. *It outputs a data integrity proof* P *for the blocks specified by* $chal$.

$\{TRUE, FALSE\} \leftarrow VerifyProof(pk, chal, P)$. *This algorithm can be run by either the client or the third party auditor upon receipt of the proof* P. *It takes as input the public key* pk, *the challenge* $chal$, *and the proof* P *returned from the server, and outputs* $TRUE$ *if the integrity of the file is verified as correct, or* $FALSE$ *otherwise.*

$(F', \Phi', P_{update}) \leftarrow ExecUpdate(F, \Phi, update)$. *This algorithm is run by the server. It takes as input a file* F, *its signatures* Φ, *and a data operation request "update" from client. It outputs an updated file* F', *updated signatures* Φ' *and a proof* P_{update} *for the operation.*

$\{(TRUE, sig_{sk}(H(R'))), FALSE\} \leftarrow VerifyUpdate(pk, update, P_{update})$. *This algorithm is run by the client. It takes as input public key* pk, *the signature* $sig_{sk}(H(R))$, *an operation request "update", and the proof* P_{update} *from server. If the verification successes, it outputs a signature* $sig_{sk}(H(R'))$ *for the new root* R', *or* $FALSE$ *otherwise.*

3.3 Our Construction

以BLS短签名为基础，构造了一个支持数据动态操作的系统。该方案在RSA构造中也可以实施。在第3.4节的讨论中，我们将展示对以前工作[2,1]的直接扩展存在安全问题，我们相信支持动态数据操作的协议设计是云存储系统的一项具有重大挑战性的主要任务。

现在我们开始展示方案的主要思想。

Now we start to present the main idea behind our scheme. As in the previous PoR systems [3,1], we assume the client encodes the raw data file $\widetilde{F}$ into F using Reed-Solomon codes and divides the encoded file F into n blocks $m_1, \ldots, m_n$[1],

where $m_i \in \mathbb{Z}_p$ and p is a large prime. Let $e : G \times G \rightarrow G_T$ be a bilinear map, with a hash function $H : \{0,1\}^* \rightarrow G$, viewed as a random oracle [1]. Let g be the generator of G. h is a cryptographic hash function. The procedure of our protocol execution is as follows:

Setup

■ **Setup**: The client's public key and private key are generated by invoking $KeyGen(\cdot)$. By running $SigGen(\cdot)$, the raw data file F is pre-processed and the homomorphic authenticators together with metadata are produced.

$KeyGen(1^k)$. The client chooses a random $\alpha \leftarrow \mathbb{Z}_p$ and computes $v \leftarrow g^\alpha$. The secret key is $sk = (\alpha)$ and the public key is $pk = (v)$.

$SigGen(sk, F)$. Given $F = (m_1, \ldots, m_n)$, the client chooses a random element $u \leftarrow G$ and computes signature σ_i for each block m_i $(i = 1, \ldots, n)$ as $\sigma_i \leftarrow (H(m_i) \cdot u^{m_i})^\alpha$. Denote the set of signatures by $\Phi = \{\sigma_i\}$, $1 \leq i \leq n$. The client then generates a root R based on the construction of Merkle Hash Tree (MHT), where the leave nodes of the tree are an ordered set of BLS hashes of "file tags" $H(m_i)$ $(i = 1, \ldots, n)$. Next, the client signs the root R under the private key α: $sig_{sk}(H(R)) \leftarrow (H(R))^\alpha$. The client sends $\{F, \Phi, sig_{sk}(H(R))\}$ to the server and deletes them from its local storage.

Default Integrity Verifification

■ **Default Integrity Verification**: The client or the third party, e.g., TPA, can verify the integrity of the outsourced data by challenging the server. To generate the message "*chal*", the TPA (verifier) picks a random c-element subset $I = \{s_1, \ldots, s_c\}$ of set $[1, n]$, where we assume $s_1 \leq \cdots \leq s_c$. For each $i \in I$ the TPA chooses a random element $\nu_i \leftarrow \mathbb{Z}_p$. The message "*chal*" specifies the positions of the blocks to be checked in this challenge phase. The verifier sends the *chal* $\{(i, \nu_i)\}_{s_1 \leq i \leq s_c}$ to the prover (server).

$GenProof(F, \Phi, chal)$. Upon receiving the challenge $chal = \{(i, \nu_i)\}_{s_1 \leq i \leq s_c}$, the server computes

$$\mu = \sum_{i=s_1}^{s_c} \nu_i m_i \in \mathbb{Z}_p \quad \text{and} \quad \sigma = \prod_{i=s_1}^{s_c} \sigma_i^{\nu_i} \in G.$$

In addition, the prover will also provide the verifier with a small amount of auxiliary information $\{\Omega_i\}_{s_1 \leq i \leq s_c}$, which are the node siblings on the path from the leaves $\{h(H(m_i))\}_{s_1 \leq i \leq s_c}$ to the root R of the MHT. The prover responds the verifier with proof $P = \{\mu, \sigma, \{H(m_i), \Omega_i\}_{s_1 \leq i \leq s_c}, sig_{sk}(H(R))\}$.

$VerifyProof(pk, chal, P)$. Upon receiving the responses from the prover, the verifier generates root R using $\{H(m_i), \Omega_i\}_{s_1 \leq i \leq s_c}$ and authenticates it by checking $e(sig_{sk}(H(R)), g) \stackrel{?}{=} e(H(R), g^\alpha)$. If the authentication fails, the verifier rejects by emitting *FALSE*. Otherwise, the verifier checks

$$e(\sigma, g) \stackrel{?}{=} e(\prod_{i=s_1}^{s_c} H(m_i)^{\nu_i} \cdot u^\mu, v).$$

If so, output $TRUE$; otherwise $FALSE$. The protocol is illustrated in Fig. 2.

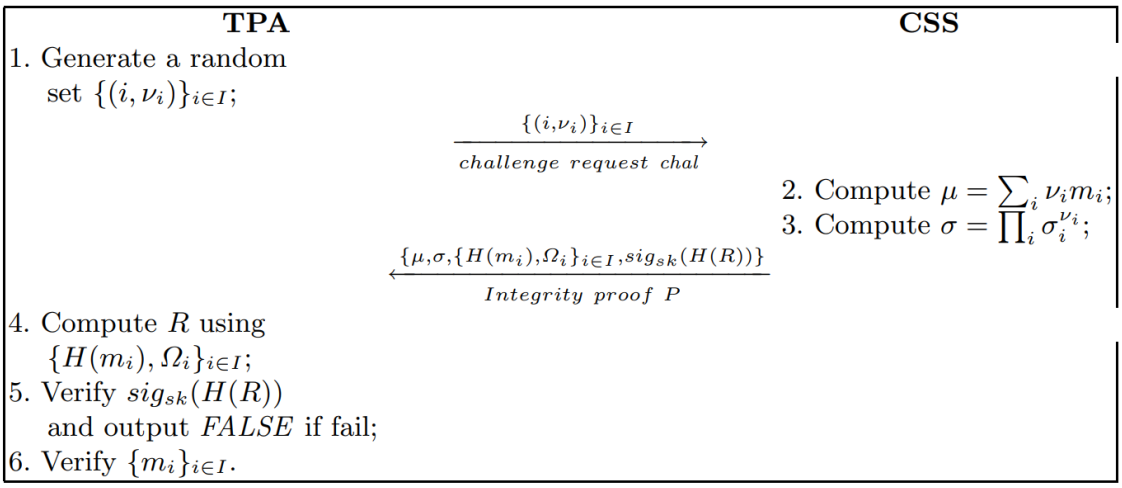


Fig. 2. Protocols for Default Integrity Verification

敌手拥有计算H（mi）的优势，可以通过操作H（mi）和mi来欺骗验证者。例如，假设验证程序想要一次检查m1和m2的完整性。在收到挑战后，证明者可以只使用文件中两个块的任意组合来计算这对块（σ，μ）。现在，以这种方式制定的响应可以成功地通过完整性检查。因此，为了防止这种攻击，我们应该在验证之前首先对标签信息进行身份验证，即确保这些标签与要检查的块相对应。

Dynamic Data Operation with Integrity Assurance

现在，我们将展示了我们的方案如何显式和有效地处理完全动态的数据操作，包括用于云数据存储的数据修改(M)、数据插入(I)和数据删除(D)。

请注意，在以下关于动态操作的协议设计的描述中，我们假设文件F和签名Φ已经生成并正确地存储在服务器上。根的元数据R已由客户端签名并存储在云服务器上，因此任何拥有客户机公钥的人都可以质疑数据存储的正确性。

- **Data Modification**

我们从数据修改开始，这是云数据存储中最常用的操作之一。一个基本的数据修改操作是指用新的块替换指定的块。

Suppose the client wants to modify the i-th block m_i to m_i'. The protocol procedures are described in Fig. 3. At start, based on the new block m_i', the client generates the corresponding signature $\sigma_i' = (H(m_i') \cdot u^{m_i'})^{\alpha}$. Then, he constructs an *update request* message "$update = (\mathcal{M}, i, m_i', \sigma_i')$" and sends to the server, where $\mathcal{M}$ denotes the modification operation. Upon receiving the request, the server runs $ExecUpdate(F, \Phi, update)$. Specifically, the server (i) replaces the block m_i with m_i' and outputs F'; (ii) replaces the σ_i with σ_i' and outputs Φ'; (iii) replaces $H(m_i)$ with $H(m_i')$ in the Merkle hash tree construction and generates the new root R' (see the example in Fig. 4). Finally, the server responses the client with a proof for this operation, $P_{update} = (\Omega_i, H(m_i), sig_{sk}(H(R)), R')$, where Ω_i is the AAI for authentication of m_i. After receiving the proof for modification operation from server, the client first generates root R using $\{\Omega_i, H(m_i)\}$ and authenticates the AAI or R by checking $e(sig_{sk}(H(R)), g) \stackrel{?}{=} e(H(R), g^{\alpha})$. If it is not true, output *FALSE*, otherwise the client can now check whether the server has performed the modification as required or not, by further computing the new root value using $\{\Omega_i, H(m_i')\}$ and comparing it with R'. If it is not true, output *FALSE*, otherwise output *TRUE*. Then, the client signs the new root metadata R' by $sig_{sk}(H(R'))$ and sends it to the server for update.

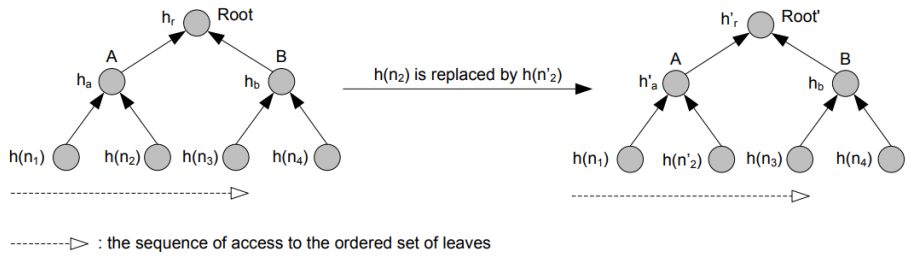


Fig. 4. Example of MHT update under block modification operation. Here, n_i and n_i' are used to denote $H(m_i)$ and $H(m_i')$, respectively.

Data Insertion

数据插入，是指在数据文件F中的某些指定位置之后插入新的块。

Suppose the client wants to insert block m^* after the i-th block m_i. The protocol procedures are similar to the data modification case (see Fig. 3, now m'_i can be seen as m^*). At start, based on m^* the client generates the corresponding signature $\sigma^* = (H(m^*) \cdot u^{m^*})^\alpha$. Then, he constructs an *update request* message "$update = (\mathcal{I}, i, m^*, \sigma^*)$" and sends to the server, where $\mathcal{I}$ denotes the insertion operation. Upon receiving the request, the server runs $ExecUpdate(F, \Phi, update)$. Specifically, the server (i) stores m^* and adds a leaf $h(H(m^*))$ "after" leaf $h(H(m_i))$ in the Merkle hash tree and outputs F'; (ii) adds the σ^* into the signature set and outputs Φ'; (iii) generates the new root R' based on the updated Merkle hash tree. Finally, the server responses the client with a proof for this operation, $P_{update} = (\Omega_i, H(m_i), sig_{sk}(H(R)), R')$, where Ω_i is the AAI for authentication of m_i in the old tree. An example of block insertion is illustrated in Fig. 5, to insert $h(H(m^*))$ after leaf node $h(H(m_2))$, only node $h(H(m^*))$ and an internal node C is added to the original tree, where $h_c = h(h(H(m_2))||h(H(m^*)))$. After receiving the proof for insert operation from server, the client first generates root R using $\{\Omega_i, H(m_i)\}$ and authenticates the AAI or R by checking if $e(sig_{sk}(H(R)), g) = e(H(R), g^\alpha)$. If it is not true, output *FALSE*, otherwise the client can now check whether the server has performed the insertion as required or not, by further computing the new root value using $\{\Omega_i, H(m_i), H(m^*)\}$ and comparing it with R'. If it is not true, output *FALSE*, otherwise output *TRUE*. Then, the client signs the new root metadata R' by $sig_{sk}(H(R'))$ and sends it to the server for update.

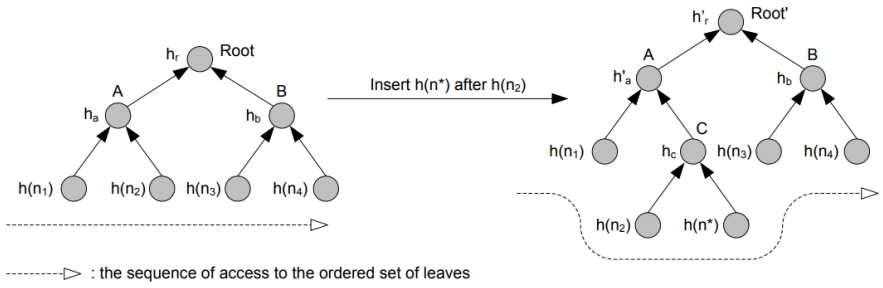


Fig. 5. Example of MHT update under block insertion operation. Here, n_i and n^* are used to denote $H(m_i)$ and $H(m^*)$, respectively.

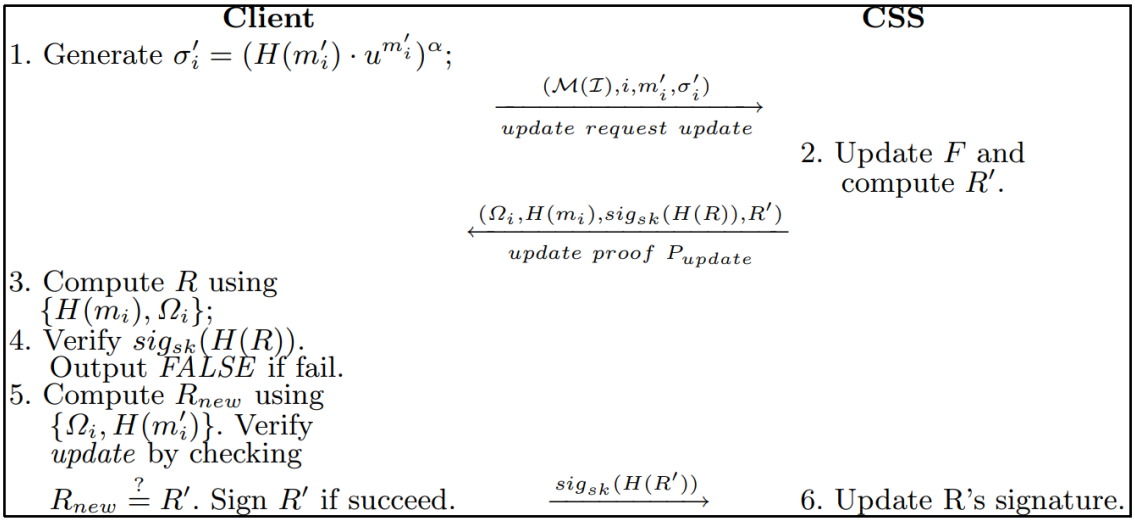


Fig. 3. The protocol for provable data update (Modification and Insertion)

Data Deletion

数据删除只是数据插入的相反操作。对于单块删除，是指删除指定的块并将所有块向前移动一个块。

-***Data Deletion***: Data deletion is just the opposite operation of data insertion. For single block deletion, it refers to deleting the specified block and moving all the latter blocks one block forward. Suppose the server receives the *update* request for deleting block m_i, it will delete m_i from its storage space, delete the leaf node $h(H(m_i))$ in the MHT and generate the new root metadata R' (see the example in Fig. 6). The details of the protocol procedures are similar to that of data modification and insertion, which are thus omitted here.

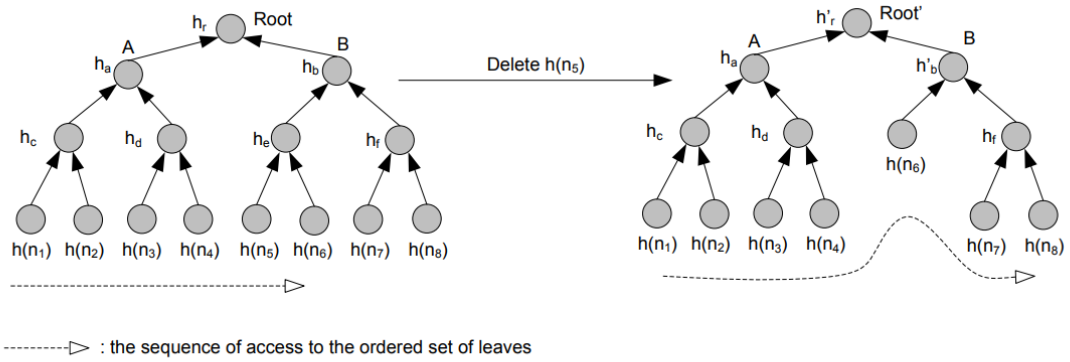


Fig. 6. Example of MHT update under block deletion operation

4 Security Analysis

Theorem 1. *If the signature scheme is existentially unforgeable and the computational Diffie-Hellman problem is hard in bilinear groups, no adversary against the soundness of our public-verification scheme could cause verifier to accept in a proof-of-retrievability protocol instance with non-negligible probability, except by responding with correctly computed values.*

Theorem 2. *Suppose a cheating prover on an n-block file F is well-behaved in the sense above, and that it is ϵ-admissible. Let $\omega = 1/\sharp B + (\rho n)^{\ell}/(n-c+1)^{c}$. Then, provided that $\epsilon - \omega$ is positive and non-negligible, it is possible to recover a ρ-fraction of the encoded file blocks in $O(n/(\epsilon-\rho))$ interactions with cheating prover and in $O(n^2 + (1+\epsilon n^2)(n)/(\epsilon-\omega))$ time overall.*

Theorem 3. *Given a fraction of the n blocks of an encoded file F, it is possible to recover the entire original file F with all but negligible probability.*