Efficient Dynamic Provable Data Possession Protocols with Public Verifiability and Data Privacy

Clémentine Gritti, Willy Susilo, Thomas Plantard and Rongmao Chen
Centre for Computer and Information Security Research
School of Computing and Information Technology
University of Wollongong, Australia
Emails: {cjpg967,rc517}@uowmail.edu.au, {wsusilo,thomaspl}@uow.edu.au

Abstract

An efficient Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy was recently published in ACISP'15. It appears that three attacks menace this scheme. The first one enables the server to store only one block of a file m and still pass the data integrity verification on any number of file blocks. The second attack permits the server to keep the old version of a file block m_i and the corresponding verification metadata T_{m_i}, after the client asked to modify them by sending the new version of these elements, and still pass the data integrity verification. The last attack allows the Third Party Auditor (TPA) to distinguish files when proceeding the data integrity checking, without accessing their contents.

In this paper, we propose several solutions to overcome all the aforementioned issues. For the two first attacks, we give two new constructions of the scheme, one using Index-Hash Tables and the other based on the Merkle Hash Trees. We compare the efficiency of these two new systems with the previous one. For the third attack, we suggest a weaker security model for data privacy without modifying the original scheme and two new constructions either based on the Index-Hash Tables or on the Merkle Hash Trees to enhance the security and to achieve the strongest data privacy notion.

Keywords: Provable Data Possession, Dynamcity, Public Verifiability, Security, Data Privacy.

1. Introduction

Provable Data Possession (PDP) is a protocol that allows a client to verify the integrity of its data stored at an untrusted server without the need to retrieve the entire file. In a Dynamic Provable Data Possession (DPDP) system with Public Verifiability and Data Privacy, three entities are involved: a client who is the owner of the data to be stored, a server that stores the data and a Third Party Auditor (TPA) who may be required when the client wants to check the integrity of its data stored on the server, without acquiring any information about these data. The system is publicly verifiable with the possible help of the

TPA who acts on behalf of the client to check the integrity of the data. The system exhibits data dynamicity at block level such that three essential operations can be done, namely data insertion, data deletion and data modification. Finally, the system is secure at the untrusted server, meaning that a server cannot successfully generate a correct proof of data possession without storing all the file blocks, and data private, meaning that the TPA learns nothing about the data of the client from all available information.

In [5], the authors presented an efficient and practical PDP system by adopting asymmetric pairings to gain in efficiency and reducing the group exponentiation and pairing operations. In their scheme, no exponentiation and only three pairings are required during the proof of data possession check, which outperformed all the existing schemes in the literature at this stage. Furthermore, the TPA on behalf of the client is allowed to request the server for a proof of data possession on as many data blocks as possible at no extra cost.

Nevertheless, we find three attacks on the DPDP scheme with Public Verifiability and Data Privacy [5]. The first one enables the server to store only one block of a file m and still pass the data integrity verification on any number of file blocks. The second attack permits the server to keep the old version of a file block m_i and the corresponding verification metadata T_{m_i}, after the client asked to modify them by sending the new version of these elements, and still pass the data integrity verification.

To overcome these two issues, we have found solutions that forces us to switch from the standard model to the random oracle model. However, the practicality of our improved schemes is not impacted: no exponentiation and four pairings (instead of three) are needed during the proof of data possession check. Solutions include the employ of Index-Hash Tables (IHTs) and Merkle Hash Trees (MHTs) respectively.

The last attack threatens the data privacy in [5]. Indeed, in their security model, the TPA plays the role of the attacker and is allowed to submit two equal-length files m_0 and m_1 to the challenger. The latter chooses a bit $b \in \{0,1\}$ and sends back the verification metadata corresponding to each block of m_b. Given these elements as well as the public key, the attacker is able to discover which file was chosen by the challenger.

To overcome this issue, there are two options. The first one is to relax the security notion: instead of the indistinguishability property, we focus on the one-way property. The second one is to employ Index-Hash Tables (IHTs) or Merkle Hash Trees (MHTs) to achieve the strongest security notion at the cost of efficiency loss.

1.1. Related Work

Ateniese et al. [1] introduced the notion of Provable Data Possession (PDP), which enables a client to check the integrity of its data stored at an untrusted server without retrieving the entire file. Their scheme is designed for static data and used public key-based homomorphic tags for auditing the data file. Nevertheless, the precomputation of the verification metadata imposes heavy computation overhead that can be expensive for the entire file. Thereafter, Ateniese et al. [2] proposed scalable and efficient schemes using symmetric keys in order to improve the efficiency of the audit. This results in lower overhead than their previous scheme

[1]. The scheme partially supports dynamic data operations (block updates, deletions and appends to the stored file); however, it is not publicly verifiable and is limited in number of verification requests.

Subsequently, several works were presented following the models given in [1, 2]. Wang et al. [17] combined a BLS-based homomorphic authenticator with a MHT to achieve a public auditing protocol with fully dynamic data. Yu et al. [21] joined the techniques of Attribute-Based Encryption, Proxy Re-Encryption and Lazy Re-Encryption together to obtain a scheme that guarantees fine-grainedness, scalability and data confidentiality of access control. Hao et al. [6] designed a dynamic public auditing system based on RSA. However, they did not provide any proof of security. Their scheme is shown not to be secure with respect to data privacy in [22]. The authors in the [22] gave an improved scheme that is proved to be data private. Erway et al. [3] proposed a fully Dynamic PDP (DPDP) scheme based on rank-based authenticated dictionary. Unfortunately, their system is very inefficient. Zhu et al. [26] used IHTs to support fully dynamic data and constructed a zero-knowledge PDP. Zhu et al. [25] created a dynamic audit service based on fragment structure, random sampling and IHT that supports timely anomaly detection. Wang et al. [16] proposed a system to ensure the correctness of users' data stored on multiple servers by requiring homomorphic tokens and erasure codes in the auditing process. Le and Markopoulou [7] constructed an efficient dynamic remote data integrity checking scheme based on homomorphic MAC scheme and CPA-secure encryption scheme and specifically designed for network coding based storage cloud. Wang et al. [15] gave a flexible distributed storage integrity auditing protocol utilizing the homomorphic token and the distributed erasure-coded data. Later, Wang et al. [12] designed a privacy-preserving protocol, called Oruta, that allows public auditing on shared data stored in the cloud. They exploited ring signatures to compute the verification information needed to audit the integrity of shared data. The scheme allows public auditing and identity privacy but fails to support large groups and traceability. In a parallel work, Wang et al. [11] presented a privacy-preserving auditing system, called Knox, for data stored in the cloud and shared among a large number of users in the group. They used group signatures to construct homomorphic authenticators. The scheme allows identity privacy, large users' number and traceability but is only for private auditing. Note that in both schemes, the identity of the signer on each block in shared data is kept private from a Third Party Auditor (TPA), who is still able to verify the integrity of shared data without retrieving the entire file. Nevertheless, Yu et al. [24] investigated the active adversary attacks in three auditing protocols for shared data in the cloud, including the two identity privacy preserving auditing systems Oruta [12] and Knox [11], and the aforementioned distributed storage integrity auditing system [15]. They showed that these schemes become insecure when active adversaries are involved in the cloud storage (i.e. they can alter the cloud data without being detected by the auditor in the verification phase). Yang et al. [20] presented a survey of the previous works on data auditing. Subsequently, Wang et al. [14] proposed another privacy-preserving scheme with public auditing. Nevertheless, Fan et al. [4] showed that such scheme cannot guarantee no information leakage since indistinguishability is not achieved. The authors in [4] then gave a new definition of data privacy based on indistinguishability along with an new version of the aforementioned scheme that satisfies the new security model. Wang et al. [13] suggested a protocol, called Panda, that achieves user revocation (by using Proxy Re-Signature model), public verifiability, data dynamicity (by using IHT) and batch auditing. However, Yu et al. [23] showed that Panda is not secure since a server can hide data loss without being detected. The authors in [23] proposed a solution to overcome the issue that keeps the properties of Panda.

Recently, another DPDP scheme with Public Verifiability and Data Privacy was proposed [5], such that the efficiency of this scheme outperformed the one from all the previous related systems. However, the scheme is subject to several attacks, including replay and replace ones.

1.2. Contributions

As we mentioned above, three attacks threatens the scheme in [5]. In this paper, we propose several solution to overcome these problems. For the two first attacks, we give two new constructions of the scheme, one using IHTs and the other one based on the MHTs. We show that the efficiency in the improved systems is slightly affected by comparing it with the one in [5]. For the third attack, we suggest a weaker security model for data privacy for the protocol in [5] and then, we propose two new constructions either based on the IHTs or on the MHTs to achieve the strongest data privacy notion.

2. Preliminaries

In this section, we recall the definition of the DPDP protocol and the security models provided in [5].

2.1. DPDP Scheme Definition

The file to be stored is split into n blocks, and each block is split into s sectors. We let each block and sector be elements of $\mathbb{Z}_p$ for some large prime p. For instance, let the file be b bits long. Then, the file is split into $n = \lceil b/s \cdot \log(p) \rceil$ blocks. The aforementioned intuition comes from [10]. Suppose that the blocks contain $s \geq 1$ elements of $\mathbb{Z}_p$. Therefore, a tradeoff exists between the storage overhead and the communtication overhead. More precisely, the communication complexity rises as $s+1$ elements of $\mathbb{Z}_p$. Finally, a larger value of s yields less storage overhead at cost of a high communication. Moreover, p should be λ bits long, where λ is the security parameter such that $n >> \lambda$.

A Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\mathbf{KeyGen}, \mathbf{TagGen}, \mathbf{PerfOp}, \mathbf{CheckOp}, \mathbf{GenProof}, \mathbf{CheckProof})$ is as follows:

$\mathbf{KeyGen}(\lambda) \rightarrow (pk, sk)$. The probabilistic key generation algorithm is run by the client to setup the scheme. It takes as input the security parameter λ, and outputs a pair of public and secret keys (pk, sk).

TagGen$(pk, sk, m) \to T_m$. The (possibly) probabilistic tag generation algorithm is run by the client to generate the verification metadata. It takes as inputs the public key pk, the secret key sk and a file block m, and outputs a verification metadata T_m. Then, the client stores all the file blocks m in an ordered collection $\mathbb{F}$ and the corresponding verification metadata T_m in an ordered collection $\mathbb{E}$. It forwards these two collections to the server and deletes them from its local storage.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{operation}, l', m_{l'}, T_{m_{l'}})) \to (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data operation (insertion, deletion or modification) requested by the client. It takes as inputs the public key pk, the previous collection $\mathbb{F}$ of all the file blocks, the previous collection $\mathbb{E}$ of all the verification metadata, the type of the data operation to be performed, the index l' denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$ - $l' = \frac{2i+1}{2}$ for insertion, $l' = i$ for deletion and modification), the file block $m_{l'}$ to be inserted, deleted or modified ($m_{l'} = m_{\frac{2i+1}{2}}$ for insertion, $m_{l'}$ is not required for deletion and $m_{l'} = m'_i$ for modification), and the corresponding verification metadata $T_{l'}$ to be inserted, deleted or modified ($T_{m_{l'}} = T_{m_{\frac{2i+1}{2}}}$ for insertion, $T_{m_{l'}}$ is not required for deletion and $T_{m_{l'}} = T_{m'_i}$ for modification), for $l' = 0, \cdots, n$. More precisely, for the operation:

- *insertion:* $m_{\frac{2i+1}{2}}$ is inserted between the existing blocks m_i and m_{i+1} and $T_{m_{\frac{2i+1}{2}}}$ is inserted between the existing verification metadata T_{m_i} and $T_{m_{i+1}}$, for $i = 1, \cdots, n-1$. For $i = 0$, $m_{\frac{1}{2}}$ is appended before m_1 and $T_{m_{\frac{1}{2}}}$ is appended before T_{m_1}. For $i = n$, $m_{\frac{2n+1}{2}}$ is appended after m_n and $T_{m_{\frac{2n+1}{2}}}$ is appended after T_{m_n}.
- *deletion:* m_i is deleted, giving that m_{i-1} is followed by m_{i+1} and T_{m_i} is deleted, giving that $T_{m_{i-1}}$ is followed by $T_{m_{i+1}}$, for $i = 2, \cdots, n-1$. For $i = 1$, m_1 is removed, giving that the file now begins from m_2, and T_{m_1} is removed, giving that the collection of verification metadata now begins from T_{m_2}. For $i = n$, m_n is removed, giving that the file now ends at m_{n-1}, and T_{m_n} is removed, giving that the collection of verification metadata now ends at $T_{m_{n-1}}$.
- *modification:* m'_i replaces m_i and $T_{m'_i}$ replaces T_{m_i}. We assume that the file block m'_i and the corresponding verification metadata $T_{m'_i}$ were provided by the client to the server, such that $T_{m'_i}$ was correctly computed by running the algorithm **TagGen**.

Finally, it outputs the updated file block collection $\mathbb{F}'$ containing $m_{l'}$ for insertion and modification and not containing it for deletion, the updated verification metadata collection $\mathbb{E}'$ containing $T_{m_{l'}}$ for insertion and modification and not containing it for deletion.

CheckOp$(pk, \nu') \to \{\text{"success"}, \text{"failure"}\}$. This algorithm is run by the TPA on behalf of the client to verify the server's behavior during the data operation (insertion, deletion or modification). It takes as inputs the public key pk and the updating proof ν' sent by the server. It outputs "success" if ν' is a correct updating proof; otherwise it outputs "failure".

GenProof$(pk, F, chal, \Sigma) \rightarrow \nu$. This algorithm is run by the server in order to generate a proof of data possession. It takes as inputs the public key pk, an ordered collection $F \subset \mathbb{F}$ of blocks, a challenge $chal$ and an ordered collection $\Sigma \subset \mathbb{E}$ which are the verification metadata corresponding to the blocks in F. It outputs a proof of data possession ν for the blocks in F that are determined by the challenge $chal$.

CheckProof$(pk, chal, \nu) \rightarrow \{$"success", "failure"$\}$. This algorithm is run by the TPA in order to validate the proof of data possession. It takes as inputs the public key pk, the challenge $chal$ and the proof of data possession ν. It outputs "success" if ν is a correct proof of data possession for the blocks determined by $chal$; otherwise it outputs "failure". We assume that the answer is then forwarded to the client.

Correctness. We require that a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π is *correct* if for $(pk, sk) \leftarrow$ **KeyGen**(λ), for $T_m \leftarrow$ **TagGen**(pk, sk, m), for $(\mathbb{F}', \mathbb{E}', \nu') \leftarrow$ **PerfOp**$(pk, \mathbb{F}, \mathbb{E}, info)$, for $\nu \leftarrow$ **GenProof**$(pk, F, chal, \Sigma)$, then "success" $\leftarrow$ **CheckOp**(pk, ν') and "success" $\leftarrow$ **CheckProof**$(pk, chal, \nu)$.

DPDP Scheme Construction. For lack of space, we let the reader refer to [5] for the construction of the DPDP protocol on which we focus in this paper.

2.2. Security Models

2.2.1. Security against the server

The definition of the game mentioned below follows the ones from [1] and [3]. We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi =$ (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**). Let a data possession game between a challenger $\mathcal{B}$ and an adversary $\mathcal{A}$ be as follows.

KeyGen. $(pk, sk) \leftarrow$ **KeyGen**(λ) is run by $\mathcal{B}$. The element pk is given to $\mathcal{A}$.

Adaptive queries. $\mathcal{A}$ makes adaptive queries through the intermediary of two oracles. First, $\mathcal{A}$ is given access to a tag generation oracle $\mathcal{O}_{TG}$ as follows. $\mathcal{A}$ chooses blocks m_i and forwards them to $\mathcal{B}$, for $i = 1, \cdots, n$. $\mathcal{B}$ computes the corresponding verification metadata $T_{m_i} \leftarrow$ **TagGen**(pk, sk, m_i) and gives them to $\mathcal{A}$. Then, $\mathcal{A}$ creates an ordered collection $\mathbb{F} = \{m_1, \cdots, m_n\}$ of file blocks along with an ordered collection $\mathbb{E} = \{T_{m_1}, \cdots, T_{m_n}\}$ of the corresponding verification metadata.

Thereafter, $\mathcal{A}$ is given access to a data operation performance oracle $\mathcal{O}_{DOP}$ as follows. $\mathcal{A}$ submits to $\mathcal{B}$ a block m_i, for $i = 1, \cdots, n$, and the corresponding value $info_i$ about the data operation that $\mathcal{A}$ wants to perform. $\mathcal{A}$ also outputs a new ordered collection $\mathbb{F}'$ of file blocks, a new ordered collection $\mathbb{E}'$ of verification metadata, and the corresponding updating proof ν'. $\mathcal{B}$ checks the value ν' by running the algorithm **CheckOp**(pk, ν') and gives back the resulting answer belonging to $\{$"success", "failure"$\}$ to $\mathcal{A}$. If the answer is "failure", then $\mathcal{B}$ aborts; otherwise, it proceeds. The above interaction between $\mathcal{A}$ and $\mathcal{B}$ can be repeated.

Setup. $\mathcal{A}$ submits file blocks m_i^* along with the corresponding values $info_i^*$, for $i \in \mathcal{I} \subseteq]0, n+1[\cap \mathbb{Q}$. Adaptive queries are again generated by $\mathcal{A}$, such that the first $info_i^*$ specifies a full re-write update (this corresponds to the first time that the client sends a file to the server). $\mathcal{B}$ still verifies the data operations.

Challenge. The final version of the blocks m_i for $i \in \mathcal{I}$ is considered such that these blocks were created according to the data operations requested by $\mathcal{A}$, and verified and accepted by $\mathcal{B}$ in the previous step. $\mathcal{B}$ sets $\mathbb{F} = \{m_i\}_{i \in \mathcal{I}}$ of these file blocks and $\mathbb{E} = \{T_{m_i}\}_{i \in \mathcal{I}}$ of the corresponding verification metadata. It then takes an ordered collection $F = \{m_{i_1}, \cdots, m_{i_k}\} \subset \mathbb{F}$ and the ordered collection $\Sigma = \{T_{m_{i_1}}, \cdots, T_{m_{i_k}}\} \subset \mathbb{E}$ of the corresponding verification metadata, for $i_j \in \mathcal{I}$, $j = 1, \cdots, k$. It generates a resulting challenge *chal* for F and Σ and forwards it to $\mathcal{A}$.

Forge. $\mathcal{A}$ generates a proof of data possession ν on *chal*. Then, $\mathcal{B}$ runs $\textbf{CheckProof}(pk, chal, \nu)$ and gives the answer belonging to {"success", "failure"} to $\mathcal{A}$. If the answer is "success" then $\mathcal{A}$ wins.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\textbf{KeyGen}, \textbf{TagGen}, \textbf{PerfOp}, \textbf{CheckOp}, \textbf{GenProof}, \textbf{CheckProof})$ is said to be secure if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who can win the above data possession game with non-negligible probability, then the challenger $\mathcal{B}$ can extract at least the challenged parts of the file by resetting and challenging the adversary polynomially many times by means of a knowledge extractor $\mathcal{E}$.

2.2.2. Privacy against the TPA

The definition of the game mentioned below follows the one from [22] as an enhancement of the model given in [6]. We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\textbf{KeyGen}, \textbf{TagGen}, \textbf{PerfOp}, \textbf{CheckOp}, \textbf{GenProof}, \textbf{CheckProof})$. Let a data privacy game between a challenger $\mathcal{B}$ and an adversary $\mathcal{A}$ be as follows.

KeyGen. $(pk, sk) \leftarrow \textbf{KeyGen}(\lambda)$ is run by $\mathcal{B}$. The element pk is given to $\mathcal{A}$.

Queries. $\mathcal{A}$ gives to $\mathcal{B}$ two files $m_0 = (m_{0,1}, \cdots, m_{0,n})$ and $m_1 = (m_{1,1}, \cdots, m_{1,n})$ of equal length. $\mathcal{B}$ randomly selects a bit $b \in_R \{0, 1\}$, computes $T_{m_{b,i}} \leftarrow \textbf{TagGen}(pk, sk, m_{b,i})$, for $i = 1, \cdots, n$, and gives them to $\mathcal{A}$. Then, $\mathcal{A}$ creates an ordered collection $\mathbb{F} = \{m_{b,1}, \cdots, m_{b,n}\}$ of file blocks along with an ordered collection $\mathbb{E} = \{T_{m_{b,1}}, \cdots, T_{m_{b,n}}\}$ of the corresponding verification metadata.

Challenge. $\mathcal{A}$ forwards *chal* to $\mathcal{B}$.

Generation of the Proof. $\mathcal{B}$ outputs a proof of data possession $\nu^* \leftarrow \textbf{GenProof}(pk, F, chal, \Sigma)$ for the blocks in F that are determined by the challenge *chal*, where $F = \{m_{b,i_1}, \cdots, m_{b,i_k}\} \subset$

$\mathbb{F}$ is an ordered collection of blocks and $\Sigma = \{T_{m_{b,i_1}}, \cdots, T_{m_{b,i_k}}\} \subset \mathbb{E}$ is an ordered collection of the verification metadata corresponding to the blocks in F, for $1 \leq i_j \leq n$, $1 \leq j \leq k$ and $1 \leq k \leq n$.

Guess. $\mathcal{A}$ returns a bit b'. $\mathcal{A}$ wins if $b' = b$.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\textbf{KeyGen}, \textbf{TagGen}, \textbf{PerfOp}, \textbf{CheckOp}, \textbf{GenProof}, \textbf{CheckProof})$ is said to be data private if there is no probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who can win the above data privacy game with non-negligible advantage equal to $|Pr[b' = b] - \frac{1}{2}|$.

DPDP Scheme Security Proofs. For lack of space, we let the reader refer to [5] for the security proofs of the DPDP scheme on which we focus in this paper.

Note that the security against the server given in [5] is proved based on the Discrete Logarithm assumption. In the proof for the privacy against the TPA found in [5], the analysis is wrong: the affirmation "The probability $Pr[b' = b]$ must be equal to $\frac{1}{2}$ since the verification metadata $T_{m_{b,i}}$, for $i = 1, \cdots, n$, and the proof ν^* are independent of the bit b." is wrong: $T_{m_{b,i}}$ and ν^* actually depend on b.

3. Replace and Replay Attacks

In this section, please refer to the construction of the DPDP system given in [5].

3.1. Replace Attack

In this section, we explain the first successful attack on the scheme given in [5]. The client generates the verification metadata for a file m that it wants to upload on the server.

$\textbf{TagGen}(pk, sk, m) \to T_m$. A file m is split into n blocks m_i, for $i = 1, \cdots, n$. Each block m_i is then split into s sectors $m_{i,j} \in \mathbb{Z}_p$, for $j = 1, \cdots, s$. Therefore, the file m can be seen a $n \times s$ matrix with elements denoted as $m_{i,j}$. The client computes the verification metadata $T_{m_i} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-a} = \prod_{j=1}^{s} h_j^{-a \cdot m_{i,j}}$ for $i = 1, \cdots, n$. Then, it sets $T_m = (T_{m_1}, \cdots, T_{m_n}) \in \mathbb{G}_1^n$.

Then, the client stores all the file blocks m in an ordered collection $\mathbb{F}$ and the corresponding verification metadata T_m in an ordered collection $\mathbb{E}$. It forwards these two collections to the server and deletes them from its local storage.

Yet, the server is asked to generate a proof of data possession. However, let's assume that it only stores the first block m_1 of the file m (it deleted all other blocks) and we show that it can still pass the verification process.

$\textbf{GenProof}(pk, F, chal, \Sigma) \to \nu$. The TPA prepares a challenge $chal$ to send to the server as follows. First, it chooses a subset $I \subseteq]0, n+1[\cap \mathbb{Q}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$. Second, after receiving the challenge $chal$ which indicates the specific blocks for which the client through the TPA wants a proof of data possession, the server sets the ordered collection $F = \{m_1\}_{i \in I} \subset \mathbb{F}$ of blocks (instead of $F = \{m_i\}_{i \in I}$) and an ordered collection $\Sigma = \{T_{m_1}\}_{i \in I} \subset \mathbb{E}$ which are the verification metadata corresponding to the blocks in F (instead of $\Sigma = \{T_{m_i}\}_{i \in I}$). It then selects at random $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1 = h_1^{r_1}, \cdots, R_s = h_s^{r_s}$. It also sets $b_j = \sum_{(i,v_i) \in chal} m_{1,j} \cdot v_i + r_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$ (instead of $b_j = \sum_{(i,v_i) \in chal} m_{i,j} \cdot v_i + r_j$), then $B_j = h_j^{b_j}$ for $j = 1, \cdots, s$, and $c = \prod_{(i,v_i) \in chal} {T_{m_1}}^{v_i}$ (instead of $c = \prod_{(i,v_i) \in chal} T_{m_i}^{v_i}$). Finally, it returns $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c) \in \mathbb{G}_1^{2s+1}$ to the TPA.

CheckProof$(pk, chal, \nu) \rightarrow \{$"success","failure"$\}$. The TPA has to check whether the following equation holds:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \overset{?}{=} e(\prod_{j=1}^{s} B_j, g_2) \tag{1}$$

If Eq. 1 holds, then the TPA returns "success" to the client; otherwise, it returns "failure" to the client.

Correctness. For the proof of data possession:

$$\begin{aligned}
e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) &= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} {T_{m_1}}^{v_i}, g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} \prod_{j=1}^{s} h_j^{m_{1,j} \cdot (-a) \cdot v_i}, g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) \\
&= e\left(\prod_{j=1}^{s} h_j^{\sum_{\substack{(i,v_i)\\ \in chal}} m_{1,j} \cdot v_i + r_j}, g_2\right) \\
&= e(\prod_{j=1}^{s} h_j^{b_j}, g_2) = e(\prod_{j=1}^{s} B_j, g_2)
\end{aligned}$$

Therefore, Eq. 1 holds, although the server is actually storing one block only.

N.B.. This attack is not due to the dynamicity property of our scheme. Such attack could happen even on static data.

3.2. Replay Attack

In this section, we explain the second successful attack on the scheme given in [5]. A client asks the server to modify the file block m_i by sending the new version of the block m_i' and the corresponding verification metadata $T_{m'_i}$. However, the server does not follow the client's request and decides to keep the old version of the block m_i and the corresponding verification metadata T_{m_i}, and deletes m'_i and $T_{m'_i}$.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{modification}, i, m'_i, T_{m'_i})) \to (\mathbb{F}', \mathbb{E}', \nu')$. After receiving the elements i, m'_i and $T_{m'_i}$ from the client, the server prepares the updating proof as follows. It first retrieves the block m_i and the verification metadata T_{m_i} corresponding to the index i, and once it gets these elements, it deletes m'_i and $T_{m'_i}$. It then selects at random $u_1, \cdots, u_s \in_R \mathbb{Z}_p$ and computes $U_1 = h_1^{u_1}, \cdots, U_s = h_s^{u_s}$. It also chooses at random $w_i \in_R \mathbb{Z}_p$ and sets $c_j = m_{i,j} \cdot w_i + u_j \in \mathbb{Z}_p$ (instead of $c_j = m'_{i,j} \cdot w_i + u_j \in \mathbb{Z}_p$) for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = {T_{m_i}}^{w_i}$ (instead of $d = T_{m'_i}^{w_i}$). Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

CheckOp$(pk, \nu') \to \{\text{“success”}, \text{“failure”}\}$. The TPA has to check whether the following equation holds:

$$e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) \overset{?}{=} e(\prod_{j=1}^{s} C_j, g_2) \tag{2}$$

If Eq. 2 holds, then the TPA returns “success” to the client; otherwise, it returns “failure” to the client.

Correctness. If all the algorithms are correctly generated, then the above scheme is correct. For the updating proof:

$$\begin{aligned} e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) &= e({T_{m_i}}^{w_i}, g_2^a) \cdot e(\prod_{j=1}^{s} h_j^{u_j}, g_2) \\ &= e(\prod_{j=1}^{s} h_j^{m_{i,j} \cdot (-a) \cdot w_i}, g_2^a) \cdot e(\prod_{j=1}^{s} h_j^{u_j}, g_2) \\ &= e(\prod_{j=1}^{s} h_j^{m_{i,j} \cdot w_i + u_j}, g_2) \\ &= e(\prod_{j=1}^{s} h_j^{c_j}, g_2) = e(\prod_{j=1}^{s} C_j, g_2) \end{aligned}$$

Therefore, Eq. 2 holds, although the server has not updated the block m'_i and the corresponding verification metadata $T_{m'_i}$.

N.B.. This attack is due to the dynamicity property of our scheme.

4. Attack against Data Privacy

In Section 2.2.2, we recalled the security model for Data Privacy against the TPA from [5]. The data privacy attack on the scheme given in [5] works as follows.

The TPA, who plays the role of the adversary, provides two equal-length messages m_0 and m_1 to the challenger, such that $m_0 = (m_{0,1}, \cdots, m_{0,n})$ and $m_1 = (m_{1,1}, \cdots, m_{1,n})$. The challenger chooses a bit $b \in \{0,1\}$, computes $T_{m_{b,i}} \leftarrow \mathbf{TagGen}(pk, sk, m_{b,i})$, for $i = 1, \cdots, n$, and gives them to the TPA. We recall that $T_{m_{b,i}}$ is equal to $(\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}$.

Note that

$$e(T_{m_{b,i}}, g_2) = e((\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk}, g_2) = e((\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}, g_2) = e(\prod_{j=1}^{s} h_j^{m_{b,i,j}}, (g_2^a)^{-1}).$$

The computation of the last pairing requires only public elements. Therefore, for $b' \in \{0,1\}$, the TPA is able to generate the pairing $e(\prod_{j=1}^{s} h_j^{m_{b',i,j}}, (g_2^a)^{-1})$, given the public key pk and the block of the message that it gave to the challenger, as well as the pairing $e(T_{m_{b,i}}, g_2)$, given the verification metadata sent by the challenger, and finally compares them. If these two pairings are equal, then $b' = b$; otherwise $b' \neq b$.

4.1. A Weaker Security Model

4.1.1. Idea

A first solution to avoid the aforementioned attack is to choose a weaker security model than the one proposed in [5]. The security model given in [5] is based on indistinguishability and unfortunately, the system does not satisfy such property. However, we argue that such model is too strong according to the reality. Indeed, even if the TPA is able to distinguish two files, it still does not learn anything about the contents of these files. Moreover, it may have to check the same blocks several times during different challenge-response audits. For instance, even if the TPA notices that it has verified the same block during two consecutive challenge-response audits, it only knows that this block appeared twice, however it does not know more information about it. We recall that the task of the TPA is to check that the server correctly performs data operations at block level and stores the data. Therefore, for a given operation, the TPA is aware of which block is considered and so, it may be able to differentiate it with another one given another operation. Yet again, it does not have access to more details about these two blocks.

Thus, we argue that a security model based on one-wayness is sufficient for the above scenario. In the case of one-wayness, an attacker (played by the TPA) can not get back the whole data of a given verification metadata with just public parameters at its disposal.

4.1.2. Model

We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\mathbf{KeyGen}, \mathbf{TagGen}, \mathbf{PerfOp}, \mathbf{CheckOp}, \mathbf{GenProof}, \mathbf{CheckProof})$. Let a data privacy game between a challenger $\mathcal{B}$ and an adversary $\mathcal{A}$ be as follows.

KeyGen. $(pk, sk) \leftarrow \mathbf{KeyGen}(\lambda)$ is run by $\mathcal{B}$. The element pk is given to $\mathcal{A}$.

Setup. $\mathcal{B}$ chooses a file $m = (m_1, \cdots, m_n)$ and computes $T_{m_i} \leftarrow \mathbf{TagGen}(pk, sk, m_i)$ for $i = 1, \cdots, n$. Then, it creates an ordered collection $\mathbb{F} = \{m_1, \cdots, m_n\}$ of file blocks along with an ordered collection $\mathbb{E} = \{T_{m_1}, \cdots, T_{m_n}\}$ of the corresponding verification metadata.

Challenge. $\mathcal{A}$ forwards *chal* to $\mathcal{B}$.

Generation of the Proof. $\mathcal{B}$ outputs a proof of data possession $\nu^* \leftarrow \textbf{GenProof}(pk, F, chal, \Sigma)$ for the blocks in F that are determined by the challenge *chal*, where $F = \{m_{i_1}, \cdots, m_{i_k}\} \subset \mathbb{F}$ is an ordered collection of blocks and $\Sigma = \{T_{m_{i_1}}, \cdots, T_{m_{i_k}}\} \subset \mathbb{E}$ is an ordered collectection of the verification metadata corresponding to the blocks in F, for $1 \le i_j \le n$, $1 \le j \le k$ and $1 \le k \le n$. $\mathcal{B}$ sends ν' to $\mathcal{A}$.

Guess. $\mathcal{A}$ outputs $F' = \{m'_{i_1}, \cdots, m'_{i_k}\}$ for $\Sigma = \{T_{m_{i_1}}, \cdots, T_{m_{i_k}}\}$ and wins if $F' = F$.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\textbf{KeyGen}, \textbf{TagGen}, \textbf{PerfOp}, \textbf{CheckOp}, \textbf{GenProof}, \textbf{CheckProof})$ is said to be data private if there is no PPT adversary $\mathcal{A}$ who can win the above data privacy game with non-negligible advantage $Adv_{\mathcal{A}}(\lambda)$. Informally, there is no adversary $\mathcal{A}$ who can recover the file from a given verification metadata tuple with non-negligible probability.

4.1.3. s-Strong Diffie Hellman (SDH) Problem

Let $\mathbb{G}_1$ be a cyclic group of order p. Let g_1 be a generator of $\mathbb{G}_1$. Set $\beta, \gamma \in_R \mathbb{Z}_p$. The s-SDH problem is as follows: for $\beta \in_R \mathbb{Z}_p$, given $g_1, g_1^{\beta}, \cdots, g_1^{\beta^s}$, output $(\gamma, g_1^{\frac{1}{\beta+\gamma}})$ for $\gamma \in_R \mathbb{Z}_p \setminus \{-\beta\}$. The s-SDH problem holds in $\mathbb{G}_1$ if no t-time algorithm has advantage at least ε in solving the s-SDH problem in $\mathbb{G}_1$.

4.1.4. Proof

We provide a new security proof for the construction given in [5] (following the weaker security model). For any PPT adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{B}$ that wants to break the $s+1$-SDH assumption in $\mathbb{G}_1$ by interacting with $\mathcal{A}$ as follows.

KeyGen. $\mathcal{B}$ runs $\textbf{GroupGen}(\lambda) \rightarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and receives the $s+1$-SDH instance $(g_1, g_1^{\beta}, \cdots, g_1^{\beta^{s+1}})$. Then, it chooses a generator g_2 for $\mathbb{G}_2$ and sets the elements $h_1 = g_1^{\beta^1}, \cdots, h_s = g_1^{\beta^s}, h_{s+1} = g_1^{\beta^{s+1}}$. It also randomly chooses the secret key $sk = a \in_R \mathbb{Z}_p$. It sets the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$ and forwards it to $\mathcal{A}$.

Setup. $\mathcal{B}$ implicitly chooses a file $m = (m_1, \cdots, m_n)$ such that for all $i = 1, \cdots, n$ and $j = 1, \cdots, s$, we have $m_{i,j} = \beta$ (each block m_i is divided into s sectors $m_{i,j}$). For that, it computes $T_{m_i} = \prod_{j=1}^{s} h_j^{-sk \cdot m_{i,j}} = \prod_{j=1}^{s} g_1^{-\beta^j \cdot \beta \cdot sk} = \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot sk} = \prod_{j=1}^{s} h_{j+1}^{-sk} = \prod_{j=1}^{s} h_{j+1}^{-a}$, for $i = 1, \cdots, n$, and gives them to $\mathcal{A}$.

Challenge. Without loss of generality, $\mathcal{A}$ generates a challenge on one block only. It chooses a subset $I = \{i^*\} \subseteq \{1, \cdots, n\}$, randomly chooses an element $v_{i^*} = \gamma \in_R \mathbb{Z}_p$ and sets $chal = (i^*, v_{i^*})$. It forwards $chal$ as a challenge to $\mathcal{B}$.

Generation of the Proof. Upon receiving the challenge $chal$, $\mathcal{B}$ selects an ordered collection $F = \{m_{i^*}\}$ of blocks and an ordered collection $\Sigma = \{T_{m_{i^*}}\}$ which are the verification metadata corresponding to the block in F such that $T_{m_{i^*}} = \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot a}$. It then chooses $r_1, \cdots, r_s$ at random in $\mathbb{Z}_p$ and computes $R_1^* = h_1^{r_1}, \cdots, R_s^* = h_s^{r_s}$. It also implicitly sets $b_1 = m_{i^*,1} \cdot v_{i^*} + r_1, \cdots, b_s = m_{i^*,s} \cdot v_{i^*} + r_s$ by computing $B_1^* = h_2^{\gamma} \cdot R_1^* = h_1^{\beta \cdot \gamma + r_1}, \cdots, B_s^* = h_{s+1}^{\gamma} \cdot R_s^* = h_s^{\beta \cdot \gamma + r_s}$. It sets $c^* = T_{m_{i^*}}^{v_{i^*}} = \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot sk \cdot v_{i^*}} = \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot a \cdot \gamma}$ as well. Finally, $\mathcal{B}$ returns $\nu^* = (R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*, c^*)$.

Guess. $\mathcal{A}$ outputs an ordered collection $F' = \{m'_{i^*}\}$.

Analysis. The simulation is perfect, thus the adversary must not be able to recover the ordered collection $F = \{m_{i^*}\} = \{(\beta, \cdots, \beta)\}$ set by $\mathcal{B}$, such that $|m_{i^*}| = s$, despite accepting verification metadata and proof of data possession from the challenger. Because otherwise, $\mathcal{B}$ can efficiently solve the $s+1$-SDH problem in $\mathbb{G}_1$ by receiving γ and β from $\mathcal{A}$. The proof is completed.

4.2. The Strongest Security Model

4.2.1. Idea

A second solution to avoid the above attack while keeping the same security level (i.e. indistinguishability) is to use IHTs or MHTs. Please refer to Sections 5 and 6 for details about such constructions, respectively.

The strongest notion for data privacy has been proposed by Fan et al. [4]. They defined data privacy using an indistinguishability game between a challenger $\mathcal{B}$ (the server as the prover) and an adversary $\mathcal{A}$ (the TPA as the auditor or the verifier). Note that such security model is similar to the one presented in [5]. The game is defined as follows.

Setup. $\mathcal{B}$ runs the key generation algorithm to generate the pair of public and secret keys (pk, sk) and forwards pk to $\mathcal{A}$.

Queries. $\mathcal{A}$ is allowed to make tag generation queries as follows. $\mathcal{A}$ selects a file m and sends it to $\mathcal{B}$. The latter generates the corresponding verification metadata T_m and returns it to $\mathcal{A}$.

Challenge. $\mathcal{A}$ chooses two different files m_0 and m_1 of equal length, such that they have not appeared in the Query phase, and sends them to $\mathcal{B}$. The latter computes T_{m_0} and T_{m_1} by running the algorithm **TagGen**. Then, $\mathcal{B}$ randomly chooses a bit $b \in \{0,1\}$ and sends T_{m_b} back to $\mathcal{A}$. Thereafter, $\mathcal{A}$ creates a challenge $chal$ and gives it to $\mathcal{B}$. The latter generates a proof of data possession ν based on m_b, T_{m_b} and $chal$, and sends ν to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$ and wins the game if $b' = b$.

The advantage of the adversary $\mathcal{A}$ in winning the indistinguishability game is defined as

$$Adv_{\mathcal{A}}(\lambda) = |Pr[b' = b] - \frac{1}{2}|.$$

A proof of data possession ν has indistinguishability if for any PPT adversary $\mathcal{A}$, $Adv_{\mathcal{A}}(\lambda)$ is a negligible function in the security parameter λ.

N.B.. This security model will be considered for the data privacy proofs of the IHT-based and MHT-based constructions.

5. First Solution: IHT-based Construction

5.1. Idea

A solution to avoid the replace attack is to embed the index i of the file block m_i into the verification metadata T_{m_i}. When the TPA on behalf of the client checks the proof of data possession generated by the server, it requires to use all the indices of the challenged file blocks to process the verification. Such idea was proposed for the publicly verifiable scheme in [10].

A solution to avoid the replay attack is to embed the version number vnb_i of the file block m_i into the verification metatdata T_{m_i}. The first time that the client sends the file block m_i to the server, the number vnb_i is set to be equal to 1 (meaning that the first version of the file block is uploaded) and is appended to the index i. When the client wants to modify the file block m_i with m'_i, it specifies the number $vnb_i = 2$ (meaning that the second version of file block is uploaded) and generates the verification metadata $T_{m'_i}$ accordingly. When the TPA on behalf of the client checks that the block was correctly updated by the server, it has to use both the index i and the version number vnb_i of the file block.

We stress that the index i of the file block m_i is unique. More precisely, when a block is inserted, a new index is created that has not been used and when a block is modified, the index does not change. However, when a block is deleted, its index does not disappear in order to let the scheme remain secure. To explain why, we consider that the index of a deleted block is removed. Let $m = (m_1, \cdots, m_{10})$ be a file stored on the server. The client first requests to the server to delete the file block m_5. Thus, the index 5 disappears. Later, the client asks to insert a block $m'_{\frac{4+6}{2}} = m'_5$ between the file blocks m_4 and m_6. However, the server might have not properly deleted the previous file block m_5 when the cliend asked for, and so the server may not replace the not-yet-deleted block m_5 by the block m'_5 that the client wants to insert, and can still pass the data integrity verification using the not-yet-deleted block m_5. In order to elude this situation, the index i is kept as “used” even if the block m_i is deleted and when a file block should be added between m_{i-1} and m_{i+1}, then the client can choose either an index equal to $\frac{2i-1}{2}$ for $m_{\frac{2i-1}{2}}$ or $\frac{2i+1}{2}$ for $m_{\frac{2i+1}{2}}$.

In our construction, we specify that the client deletes the file blocks and the corresponding verification metadata from its local storage once these elements are sent on the server. We also implicitly let the TPA know the indices of the file blocks that are currently stored on the server in order to challenge the server on a certain data amount of the stored data. To be quite clear, we let the client conserve a table of indices of the file blocks m_i kept on the server along with their version numbers vnb_i. Such table is then forwarded to the TPA. Refering to the aforementioned example with $m = (m_1, \cdots, m_{10})$, we suppose that the block m_5 has been deleted, the block $m_{\frac{4+5}{2}} = m_{\frac{9}{2}}$ has been added, and the block m_6 has been modified twice. Let the table stored on the client's local storage be as follows:

Index i	Version Number vnb_i	Comments
1	1	-
$\cdots$	$\cdots$	$\cdots$
4	1	-
9/2	1	-
5	-	**DELETED**
6	3	-
7	1	-
$\cdots$	$\cdots$	$\cdots$
10	1	-

The above IHT table is composed of several columns: one for the block index, one for the version number and one for some auxiliary comments. A column for random values can be added, if the verification metadata should be randomized to enhance the data privacy against the TPA. We stress that each record in the IHT is different from another to ensure that data blocks and their corresponding verification metadata cannot be forged. An example of an IHT-based PDP scheme can be found in [25].

The IHT gives better security levels against the untrusted server and TPA (in the random oracle model instead of the standard model), although this leads into an increase of the complexity of the system. Indeed, the communication and the computation costs grow for all the entities involved in the protocol. We recall that the TPA is required to help the client that does not have the necessary resources to audit the server efficiently and regularly. However, the task of the TPA should remain simple in verifying that the proof of data possession is consistent with the given challenge and that the updating proof with the requested data operation. Nevertheless, adding IHTs obliges the TPA to provide more local storage and more communication burden. Such additional costs make that the intervention of the TPA is not necessarly advantageous One may think that the TPA should be more trusted since it keeps more sensitive information elements; however, note that the TPA already has access to the indices of the data blocks in the original scheme given in [5].

Finally, we show below that such changes only slightly affect the efficiency and the practicality of the protocol compared to the one presented in [5].

5.2. Construction

We now give a new construction based on IHTs in order to overcome the two aforementioned attacks. The IHT-based Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi = (\mathbf{KeyGen}, \mathbf{TagGen}, \mathbf{PerfOp}, \mathbf{CheckOp}, \mathbf{GenProof}, \mathbf{CheckProof})$

is as follows:

KeyGen$(\lambda) \rightarrow (pk, sk)$. Let **GroupGen**$(\lambda)$ be an algorithm that, on input the security parameter λ, generates the cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order $p = p(\lambda)$ with bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Let g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Let the hash function $H : \mathbb{Q} \times \mathbb{N} \rightarrow \mathbb{G}_1$ be a random oracle. Then, the client randomly chooses s elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$. Moreover, it selects at random $a \in_R \mathbb{Z}_p$ and sets its public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a, H)$ and its secret key $sk = a$.

TagGen$(pk, sk, m) \rightarrow T_m$. A file m is split into n blocks m_i, for $i = 1, \cdots, n$. Each block m_i is then split into s sectors $m_{i,j} \in \mathbb{Z}_p$, for $j = 1, \cdots, s$. We suppose that $|m| = b$ and $n = \lceil b/s \cdot \log(p) \rceil$. Therefore, the file m can be seen a $n \times s$ matrix with elements denoted as $m_{i,j}$. The client computes the verification metadata

$$T_{m_i} \quad = \quad (H(i, vnb_i) \cdot \prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = H(i, vnb_i)^{-a} \cdot \prod_{j=1}^{s} h_j^{-a \cdot m_{i,j}}$$

Yet, it sets $T_m = (T_{m_1}, \cdots, T_{m_n}) \in \mathbb{G}_1^n$.

Then, the client stores all the file blocks m in an ordered collection $\mathbb{F}$ and the corresponding verification metadata T_m in an ordered collection $\mathbb{E}$. It forwards these two collections to the server and deletes them from its local storage.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{operation}, l', m_{l'}, T_{m_{l'}})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. After receiving the elements l', $m_{l'}$ and $T_{m_{l'}}$ from the client, the server prepares the updating proof as follows. It first selects at random $u_1, \cdots, u_s \in_R \mathbb{Z}_p$ and computes $U_1 = h_1^{u_1}, \cdots, U_s = h_s^{u_s}$. It also chooses at random $w_{l'} \in_R \mathbb{Z}_p$ and sets $c_j = m_{l',j} \cdot w_i + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_{l'}}^{w_{l'}}$. Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

More precisely, for the operation:

- *insertion:* $(l', m_{l'}, T_{m_{l'}}) = \frac{2i+1}{2}, m_{\frac{2i+1}{2}}, T_{m_{\frac{2i+1}{2}}})$ and $vnb_{l'} = vnb_{\frac{2i+1}{2}} = 1$.
- *deletion:* $(l', m_{l'}, T_{m_{l'}}) = (l', \_, \_)$ and $vnb_{l'} = vnb_i = \_$, meaning that the elements $m_{l'}$, $T_{m_{l'}}$ and $vnb_{l'}$ are not required. Indeed, the server uses the block m_i and the corresponding verification metadata T_{m_i} that are kept on its storage to generate ν'.
- *modification:* $(l', m_{l'}, T_{m_{l'}}) = (i, m'_i, T_{m'_i})$ and $vnb'_i = vnb_i + 1$.

CheckOp$(pk, \nu') \rightarrow$ {"success", "failure"}. The TPA has to check whether the following equation holds:

$$e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) \quad \stackrel{?}{=} \quad e(H(l', vnb_{l'}), g_2) \cdot e(\prod_{j=1}^{s} C_j, g_2) \tag{3}$$

If Eq. 3 holds, then the TPA returns “success” to the client; otherwise. it returns “failure” to the client.

GenProof$(pk, F, chal, \Sigma) \to \nu$. The TPA on behalf of the client prepares a challenge *chal* to send to the server as follows. First, it chooses a subset $I \subseteq]0, n+1[\cap \mathbb{Q}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$.

Second, after receiving the challenge *chal* provided by the TPA, the server sets the ordered collection $F = \{m_i\}_{i \in I} \subset \mathbb{F}$ of blocks and an ordered collection $\Sigma = \{T_{m_i}\}_{i \in I} \subset \mathbb{E}$ which are the verification metadata corresponding to the blocks in F. It then selects at random $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1 = h_1^{r_1}, \cdots, R_s = h_s^{r_s}$. It also sets $b_j = \sum_{(i,v_i) \in chal} m_{i,j} \cdot v_i + r_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $B_j = h_j^{b_j}$ for $j = 1, \cdots, s$, and $c = \prod_{(i,v_i) \in chal} T_{m_i}^{v_i}$. Finally, it returns $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c) \in \mathbb{G}_1^{2s+1}$ to the TPA.

CheckProof$(pk, chal, \nu) \to \{\text{“success”}, \text{“failure”}\}$. The TPA has to check whether the following equation holds:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \overset{?}{=} e(\prod_{\substack{(i,v_i) \\ \in chal}} H(i, vnb_i), g_2) \cdot e(\prod_{j=1}^{s} B_j, g_2) \tag{4}$$

If Eq. 4 holds, then the TPA returns “success” to the client; otherwise. it returns “failure” to the client.

Correctness. For the proof of data possession:

$$\begin{aligned}
e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) &= e\left(\prod_{\substack{(i,v_i) \\ \in chal}} T_{m_i}^{v_i}, g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i) \\ \in chal}} (H(i, vnb_i)^{-a} \cdot \prod_{j=1}^{s} h_j^{m_{i,j} \cdot (-a) \cdot v_i}), g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i) \\ \in chal}} H(i, vnb_i), g_2\right) \cdot e(\prod_{j=1}^{s} h_j^{\sum_{\substack{(i,v_i) \\ \in chal}} m_{i,j} \cdot v_i + r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i) \\ \in chal}} H(i, vnb_i), g_2\right) \cdot e(\prod_{j=1}^{s} h_j^{b_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i) \\ \in chal}} H(i, vnb_i), g_2\right) \cdot e(\prod_{j=1}^{s} B_j, g_2)
\end{aligned}$$

For the updating proof:

$$\begin{aligned}
e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) &= e(T_{m_{l'}}^{w_{l'}}, g_2^a) \cdot e(\prod_{j=1}^{s} h_j^{u_j}, g_2) \\
&= e(H(i, vnb_{l'})^{-a} \cdot \prod_{j=1}^{s} h_j^{m_{l',j} \cdot (-a) \cdot w_{l'}}, g_2^a) \cdot e(\prod_{j=1}^{s} h_j^{u_j}, g_2) \\
&= e(H(l', vnb_{l'}), g_2) \cdot e(\prod_{j=1}^{s} h_j^{m_{l',j} \cdot w_{l'} + u_j}, g_2) \\
&= e(H(l', vnb_{l'}), g_2) \cdot e(\prod_{j=1}^{s} h_j^{c_j}, g_2) \\
&= e(H(l', vnb_{l'}), g_2) \cdot e(\prod_{j=1}^{s} C_j, g_2)
\end{aligned}$$

5.3. Security against the server

5.3.1. Discrete Logarithm (DL) Problem

Let $\mathbb{G}_1$ be a multiplicative cyclic group of prime order $p = p(\lambda)$ (where λ is the security parameter). The DL problem is as follows: for $a \in \mathbb{Z}_p$, given $g_1, g_1^a \in \mathbb{G}_1$, output a. The DL problem holds in $\mathbb{G}_1$ if no t-time algorithm has advantage at least ε in solving the DL problem in $\mathbb{G}_1$.

5.3.2. Proof

For any PPT adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{B}$ that wants to break the DL problem by interacting with $\mathcal{A}$ as follows.

KeyGen. $\mathcal{B}$ runs $\mathbf{GroupGen}(\lambda) \rightarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and selects two generators g_1 and g_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then, it randomly chooses s elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$ and an element $a \in_R \mathbb{Z}_p$. Let a hash function $H : \mathbb{Q} \times \mathbb{N} \rightarrow \mathbb{G}_1$ be controlled by $\mathcal{B}$ as follows. Upon receiving a query (i_l, vnb_{i_l}) to the random oracle H for some $l \in [1, q_H]$:

- If $((i_l, vnb_{i_l}), \theta_l, W_l)$ exists in L_H, return W_l.
- Otherwise, choose $\theta_l \in_R \mathbb{Z}_p$ at random and compute $W_l = g_1^{\theta_l}$. Put $((i_l, vnb_{i_l}), \theta_l, W_l)$ in L_H and return W_l as answer.

$\mathcal{B}$ sets the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a, H)$ and forwards it to $\mathcal{A}$. It sets the secret key $sk = a$ and keeps it.

Adaptive queries. $\mathcal{A}$ has first access to the tag generation oracle $\mathcal{O}_{TG}$ as follows. It first adaptively selects blocks m_i, for $i = 1, \cdots, n$. $\mathcal{B}$ splits each block m_i into s sectors $m_{i,j}$. Then, it computes $T_{m_i} = (W \cdot \prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (W \cdot \prod_{j=1}^{s} h_j^{m_{i,j}})^{-a}$, for $i = 1, \cdots, n$, such that if $((i, vnb_i), \theta, W)$ exists in L_H, then the value W is returned. Otherwise, an element $\theta \in_R \mathbb{Z}_p$

is chosen at random, $W = g^{\theta}$ is computed, and $((i, vnb_i), \theta, W)$ is put in L_H. It gives this tuple to $\mathcal{A}$. The latter sets an ordered collection $\mathbb{F} = \{m_1, \cdots, m_n\}$ of blocks and an ordered collection $\mathbb{E} = \{T_{m_1}, \cdots, T_{m_n}\}$ which are the verification metadata corresponding to the blocks in $\mathbb{F}$. $\mathcal{A}$ has also access to the data operation performance oracle $\mathcal{O}_{DOP}$ as follows. Repeatedly, $\mathcal{A}$ selects a block $m_{l'}$ and the corresponding element $info_{l'}$ and forwards them to $\mathcal{B}$. l' denotes the rank where $\mathcal{A}$ wants the data operation to be performed: l' is equal to $\frac{2i+1}{2}$ for an insertion and to i for a deletion or a modification. Moreover, $m_{l'} = \perp$ in the case of a deletion, since only the rank is needed to perform this kind of operation. The version number $vnb_{l'}$ increases by one in the case of a modification. Then, $\mathcal{A}$ outputs a new ordered collection $\mathbb{F}'$ (containing the updated version of the block $m_{l'}$), a new ordered collection $\mathbb{E}'$ (containing the updated version of the verification metadata $T_{m_{l'}}$) and a corresponding updating proof $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d)$, such that $w_{l'}$ is randomly chosen from $\mathbb{Z}_p$, $d = T_{m_{l'}}^{w_{l'}}$, and for $j = 1, \cdots, s$, u_j is randomly chosen from $\mathbb{Z}_p$, $U_j = h_j^{r_j}$, $c_j = m_{l',j} \cdot w_{l'} + u_j$ and $C_j = h_j^{c_j}$. $\mathcal{B}$ runs the algorithm **CheckOp** on the value ν' and sends the answer to $\mathcal{A}$. If the answer is "failure", then the challenger aborts; otherwise, it proceeds.

Setup. $\mathcal{A}$ selects blocks m_i^* and the corresponding elements $info_i^*$, for $i \in \mathcal{I} \subseteq]0, n+1[\cap \mathbb{Q}$, and forwards them to the challenger who checks the data operations. In particular, the first $info_i^*$ indicates a full re-write.

Challenge. $\mathcal{B}$ chooses a subset $I \subseteq \mathcal{I}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$. It forwards $chal$ as a challenge to $\mathcal{A}$.

Forge. Upon receiving the challenge $chal$, the resulting proof of data possession on the correct stored file m should be $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ and pass the Eq. **??**. However, $\mathcal{A}$ generates a proof of data possession on an incorrect stored file $\tilde{m}$ as $\tilde{\nu} = (R_1, \cdots, R_s, \tilde{B}_1, \cdots, \tilde{B}_s, \tilde{c})$, such that r_j is randomly chosen from $\mathbb{Z}_p$, $R_j = h_j^{r_j}$, $\tilde{b}_j = \sum_{(i,v_i) \in chal} \tilde{m}_{i,j} \cdot v_i + r_j$ and $\tilde{B}_j = h_j^{\tilde{b}_j}$, for $j = 1, \cdots, s$. It also sets $\tilde{c} = \prod_{(i,v_i) \in chal} T_{\tilde{m}_i}^{v_i}$. Finally, it returns $\tilde{\nu} = (R_1, \cdots, R_s, \tilde{B}_1, \cdots, \tilde{B}_s, \tilde{c})$ to $\mathcal{B}$. If the proof of data possession still pass the verification, then $\mathcal{A}$ wins. Otherwise, it fails. We define $\Delta b_j = \tilde{b}_j - b_j$, for $j = 1, \cdots, s$. At least one element of $\{\Delta b_j\}_{j=1,\cdots,s}$ is non-zero.

Analysis. We prove that if the adversary can win the game, then a solution to the DL problem is found, which contradicts the assumption that the DL problem is hard in $\mathbb{G}_1$. Let assume that the server wins the game. Then, according to Eq. 4, we have $e(c, g_2^a) \cdot e(\prod_{j=1}^s R_j, g_2) = e(\prod_{\substack{(i,v_i) \\ \in chal}} H(i, vnb_i), g_2) \cdot e(\prod_{j=1}^s \tilde{B}_j, g_2)$. Since the proof $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ is a correct one, we also have $e(c, g_2^a) \cdot e(\prod_{j=1}^s R_j, g_2) = e(\prod_{\substack{(i,v_i) \\ \in chal}} H(i, vnb_i), g_2) \cdot e(\prod_{j=1}^s B_j, g_2)$. Therefore, we get that $\prod_{j=1}^s \tilde{B}_j = \prod_{j=1}^s B_j$. We can re-write as $\prod_{j=1}^s h_j^{\tilde{b}_j} = \prod_{j=1}^s h_j^{b_j}$ or even as $\prod_{j=1}^s h_j^{\Delta b_j} = 1$. For two elements $g, h \in \mathbb{G}_1$, there exists $x \in \mathbb{Z}_p$ such that $h = g^x$ since $\mathbb{G}_1$ is a cyclic group. Without loss of generality, given $g, h \in \mathbb{G}_1$, each h_j could randomly and correctly be generated by computing $h_j = g^{y_j} \cdot h^{z_j} \in \mathbb{G}_1$ such that y_j and z_j are random values of $\mathbb{Z}_p$. Then, we have $1 = \prod_{j=1}^{s} h_j^{\Delta b_j} = \prod_{j=1}^{s}(g^{y_j} \cdot h^{z_j})^{\Delta b_j} = g^{\sum_{j=1}^{s} y_j \cdot \Delta b_j} \cdot h^{\sum_{j=1}^{s} z_j \cdot \Delta b_j}$. Clearly, we can find a solution to the DL problem. More specifically, given $g, h = g^x \in \mathbb{G}_1$, we can compute $h = g^{\frac{\sum_{j=1}^{s} y_j \cdot \Delta b_j}{\sum_{j=1}^{s} z_j \cdot \Delta b_j}} = g^x$ unless the denominator is zero. However, as we defined in the game, at least one element of $\{\Delta b_j\}_{j=1,\cdots,s}$ is non-zero. Since z_j is a random element of $\mathbb{Z}_p$, the denominator is zero with probability equal to $1/p$, which is negligible. Thus, if $\mathcal{A}$ wins the game, then a solution of the DL problem can be found with probability equal to $1-\frac{1}{p}$, which contradicts the fact that the DL problem is assumed to be hard in $\mathbb{G}_1$. Therefore, for $\mathcal{A}$, it is computationally infeasible to win the game and generate an incorrect proof of data possession which can pass the verification.

In addition, the simulation of the random oracle H is not entirely perfect. Let's consider the event that $\mathcal{A}$ has queried before the *Challenge* phase (i^*, vnb_i^*) to H. This event happens with probability $1/p$. Except for the case above, the simulation of H is perfect. We recall that $\mathcal{A}$ makes at most q_H random oracle queries. The simulation of the tag generation oracle $\mathcal{O}_{TG}$ is perfect. The simulation of the data operation performance oracle $\mathcal{O}_{DOP}$ is almost perfect except when $\mathcal{B}$ aborts. This happens the data operation was not correclty performed. As previously, we can prove that if $\mathcal{A}$ can pass the updating proof, then a solution to the DL problem is found. Following the above analysis and according to Eq. 3, if $\mathcal{A}$ generates an incorrect updating proof which can pass the verification, then a solution of the DL problem can be found with probability equal to $1-\frac{1}{p}$, which contradicts the fact that the DL problem is assumed to be hard in $\mathbb{G}_1$. Therefore, for $\mathcal{A}$, it is computationally infeasible to generate an incorrect updating proof which can pass the verification. The proof is completed.

5.4. Data Privacy against the TPA

5.4.1. Decisional s-Strong Diffie Hellman (DSDH) Problem

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic groups of order p. Let g_1 be a generator of $\mathbb{G}_1$ and g_2 be a generator of $\mathbb{G}_2$. Choose $\beta \in_R \mathbb{Z}_p$ and $b \in_R \{0,1\}$. The s-DSDH problem instance consists of the tuple $(Z, g_1^{\gamma}, g_1, g_1^{\beta}, \cdots, g_1^{\beta^s}, g_2, g_2^{\beta})$, such that if $b = 0$, set $Z = g_1^{\beta^{s+1}}$ for some integer s; otherwise, set $Z \in_R \mathbb{G}_1$. The s-DSDH problem is to guess b and holds in $(\mathbb{G}_1, \mathbb{G}_2)$ if no t-time algorithm has advantage at least ε in solving the s-DSDH problem in $(\mathbb{G}_1, \mathbb{G}_2)$.

5.4.2. Proof

For any PPT adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{B}$ that wants to break the $s+1$-DSDH assumption by interacting with $\mathcal{A}$ as follows.

KeyGen. $\mathcal{B}$ runs **GroupGen**$(\lambda) \rightarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and receives the $s+1$-DSDH instance $(g_1, g_1^{\beta}, \cdots, g_1^{\beta^{s+1}}, g_2, g_2^{\beta})$. The challenger sets $\mu = 0$ when the output Z of the $s+1$-DSDH assumption is equal to $g_1^{\beta^{s+2}}$; otherwise, it sets $\mu = 1$ when the output Z of the $s+1$-DSDH assumption is a random element in $\mathbb{G}_1$. Then, it sets the elements $h_1 = g_1^{\beta}, \cdots, h_s = g_1^{\beta^s}, h_{s+1} = g_1^{\beta^{s+1}}$ and implicitly fixes the secret key $sk = a = \beta$ by setting $g_2^{a} = g_2^{\beta}$. The challenger $\mathcal{B}$ also gives $\mathcal{A}$ access to hash function $H : \mathbb{Q} \times \mathbb{N} \to \mathbb{G}_1$, such that H is controlled by $\mathcal{B}$ as follows: upon receiving a query (i_l, vnb_{i_l}) to the random oracle H for some $l \in [1, q_H]$:

- If $((i_l, vnb_{i_l}), \theta_l, W_l)$ exists in L_H, return W_l.
- Otherwise, choose $\theta_l \in_R \mathbb{Z}_p$ at random and compute $W_l = h_1^{-\theta_l}$. Put $((i_l, vnb_{i_l}), \theta_l, W_l)$ in L_H and return W_l as answer.

It sets the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, h_1, \cdots, h_s, g_2^a, H)$ and forwards it to $\mathcal{A}$.

Queries. $\mathcal{A}$ makes tag generation queries as follows: $\mathcal{A}$ first selects a file $m = (m_1, \cdots, m_n)$ and sends it to $\mathcal{B}$. Then, the challenger splits each block m_i into s sectors $m_{i,j}$. Then, it computes $T_{m_i} = W \cdot \prod_{j=1}^{s} g_1^{\beta^j \cdot (-\beta) \cdot m_{i,j}} = W \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{i,j}}$, such that if $((i, vnb_i), \theta, W)$ exists in L_H, then the value W is returned; otherwise, an element $\theta \in_R \mathbb{Z}_p$ is chosen at random, $W = h_1^{-\theta}$ is computed, and $((i, vnb_i), \theta, W)$ is put in L_H. It gives the corresponding verification metadata $T_m = (T_{m_1}, \cdots, T_{m_n})$ to $\mathcal{A}$.

Challenge. $\mathcal{A}$ first gives to the challenger two files $m_0 = (m_{0,1}, \cdots, m_{0,n})$ and $m_1 = (m_{1,1}, \cdots, m_{1,n})$ of equal length. $\mathcal{B}$ randomly selects a bit $b \in_R \{0, 1\}$ and for $i = 1, \cdots, n$, splits each block $m_{b,i}$ into s sectors $m_{b,i,j}$. Then, it computes $T_{m_{b,i}} = W_b \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{b,i,j}}$, such that if $((i, vnb_{b,i}), \theta_b, W_b)$ exists in L_H, then the value W_b is returned; otherwise, an element $\theta_b \in_R \mathbb{Z}_p$ is chosen at random, $W_b = h_1^{-\theta_b}$ is computed, and $((i, vnb_{b,i}), \theta_b, W_b)$ is put in L_H. It gives the verification metadata $T_{m_b} = (T_{m_{b,1}}, \cdots, T_{m_{b,n}})$ to $\mathcal{A}$.

Without loss of generality, $\mathcal{A}$ generates a challenge on one block only. It chooses a subset $I = \{i^*\} \subseteq \{1, \cdots, n\}$, randomly chooses an element $v_{i^*} \in_R \mathbb{Z}_p$ and sets $chal = (i^*, v_{i^*})$. It forwards $chal$ as a challenge to $\mathcal{B}$. Upon receiving the challenge $chal$, $\mathcal{B}$ selects an ordered collection $F_b = \{m_{b,i^*}\}$ of blocks and an ordered collection $\Sigma_b = \{T_{m_{b,i^*}}\}$ which are the verification metadata corresponding to the blocks in F_b such that $T_{m_{b,i^*}} = W_b \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{b,i^*,j}}$. It then randomly chooses $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1^* = h_1^{\beta^2 \cdot r_1} = h_3^{r_1} = g_1^{\beta^3 \cdot r_1}, \cdots, R_{s-1}^* = h_{s-1}^{\beta^2 \cdot r_{s-1}} = h_{s+1}^{r_{s-1}} = g_1^{\beta^{s+1} \cdot r_{s-1}}$ and $R_s^* = Z^{r_s}$. It implicitly fixes $b_1 = \beta^2 \cdot m_{b,i^*,1} \cdot v_{i^*} + r_1, \cdots, b_{s-1} = \beta^2 \cdot m_{b,i^*,s-1} \cdot v_{i^*} + r_{s-1}$ by computing $B_1^* = h_1^{b_1} = h_1^{\beta^2 \cdot m_{b,i^*,1} \cdot v_{i^*}} \cdot h_1^{\beta^2 \cdot r_1} = h_3^{m_{b,i^*,1} \cdot v_{i^*}} \cdot h_3^{r_1} = g_1^{\beta^3 \cdot m_{b,i^*,1} \cdot v_{i^*}} \cdot g_1^{\beta^3 \cdot r_1}, \cdots, B_{s-1}^* = h_{s-1}^{b_{s-1}} = h_{s-1}^{\beta^2 \cdot m_{b,i^*,s-1} \cdot v_{i^*}} \cdot h_{s-1}^{\beta^2 \cdot r_{s-1}} = h_{s+1}^{m_{b,i^*,s-1} \cdot v_{i^*}} \cdot h_{s+1}^{r_{s-1}} = g_1^{\beta^{s+1} \cdot m_{b,i^*,s-1} \cdot v_{i^*}} \cdot g_1^{\beta^{s+1} \cdot r_{s-1}}$ and $B_s^* = Z^{m_{b,i^*,s} \cdot v_{i^*}} \cdot Z^{r_s}$. It sets $c^* = T_{m_{b,i^*}}^{v_{i^*}} = W_b^{v_{i^*}} \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{b,i^*,j} \cdot v_{i^*}}$ as well. Finally, $\mathcal{B}$ returns $\nu^* = (R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*, c^*)$.

If $\mu = 0$ then $Z = g_1^{\beta^{s+2}}$. Therefore, the proof of data possession is a valid random proof for the file block m_b. Otherwise, if $\mu = 1$, then Z is a random value in $\mathbb{G}_1$. Since Z is random, the values R_s and B_s will be random elements of $\mathbb{G}_1$ from the adversary's view and the proof of data possession contains no information about m_b.

Guess. The adversary returns a bit b'. If $b = b'$, the challenger will output $\mu' = 0$ to indicate that it was given a $s+1$-DSDH tuple; otherwise it will output $\mu' = 1$ to indicate that it was given a random tuple.

Analysis. We prove that the verification metadata and the proof of data possession given to $\mathcal{A}$ are correctly distributed. In the case where $\mu = 1$, $\mathcal{A}$ gains no information about b. Therefore, we have $Pr[b \neq b' | \mu = 1] = 1/2$. Since $\mathcal{B}$ guesses $\mu' = 1$ when $b \neq b'$, we have $Pr[\mu = \mu' | \mu = 1] = 1/2$. If $\mu = 0$, then $\mathcal{A}$ sees an upload of m_b. The adversary's advantage in this situation is negligible by definition, i.e. equal to a given ϵ. Therefore, we have $Pr[b \neq b' | \mu = 0] = 1/2 + \epsilon$. Since the challenger guesses $\mu' = 0$ when $b = b'$, we have $Pr[\mu = \mu' | \mu = 0] = 1/2 + \epsilon$. The value $T_{m_{b,i}}$ is equal to $(H(i, vnb_{b,i}) \cdot \prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk}$ and $sk = a$ is kept secret from $\mathcal{A}$. Moreover, the above simulation is almost perfect since the simulation of the random oracle H is not entirely perfect. Let's consider the event that $\mathcal{A}$ has queried before the *Challenge* phase (i^*, vnb_i^*) to H. This event happens with probability $1/p$. Except for the case above, the simulation of H is perfect. We recall that $\mathcal{A}$ makes at most q_H random oracle queries. In addition, $R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*$ are statically indistinguishable with the current output corresponding to m_0 or m_1. Thus, the answers given to $\mathcal{A}$ are correctly distributed. The proof is completed.

5.5. Performance

We compare the IHT-based scheme with the original scheme proposed in [5]. First, the client and the TPA obviously have to store more information by keeping the IHT. Nevertheless, we stress that in any case, the client and the TPA should maintain an index list. Indeed, they need some information about the stored data in order to select some data blocks to be challenged. We recall that the challenge consists of pairs (index, random element). By appending an integer and sometimes an auxiliary comment (only in case of deletions) to each index, the extra burden does not seem excessive. Therefore, such table does slightly affect the client's as well as the TPA's local storage. The communication between the client and the TPA rather increases since the client should send more elements to the TPA in order to keep the table updated.

Second, the client has to perform extra computation when generating the verification metatdata: for each file block m_i, it has to compute $H(i, vnb_i)$. However, the communication between the client and the server overhead does not inscrease.

Third, the TPA need to compute an extra pairing $e(H(i, vnb_i), g_2)$ in order to check that the server correclty performed a data operation requested by the client. The TPA also has to compute $|I|$ multiplications in $\mathbb{G}_1$ and one extra pairing when checking the proof of data possession: for each challenge $chal = \{(i, v_i)\}_{i \in I}$, it calculates $\prod_{(i,vi) \in chal} H(i, vnb_i)$ as well as the pairing $e(\prod_{(i,vi) \in chal} H(i, vnb_i), g_2)$. This gives a constant total of four pairings in order to verify the data integrity instead of three, that is not a big loss in term of efficiency and practicality.

Finally, apart the storage of a light table and the computation of an extra pairing by the TPA for the verification of both the updating proof and the proof of data possession, the new construction for the DPDP protocol is still practical by adopting asymmetric pairings to gain efficiency and by still reducing the group exponentiation and pairing operations. In addition, this scheme still allows the TPA on behalf of the client to request the server for a proof of data possession on as many data blocks as possible at no extra cost, as in the scheme given in [5].

6. Second Solution: MHT-based Construction

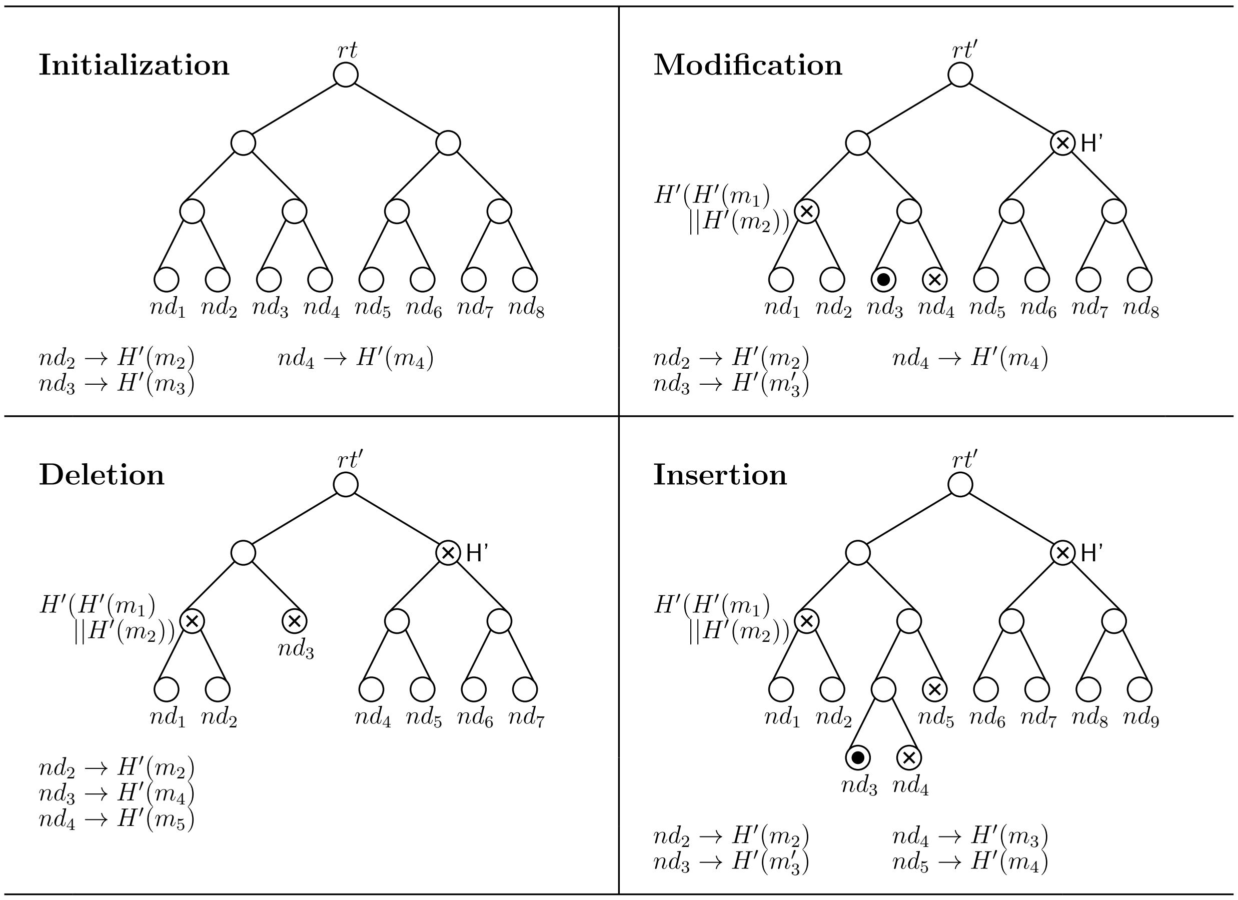
6.1. Idea

The second solution to avoid the replace and replay attacks is to implement a MHT for each file such that the file blocks are ordered. The MHT [9] is similar to a binary tree in the way that each node nd has at most two children. Following the update algorithm, each internal (non-leaf) node has always two children. The construction of a MHT is as follows. For leaf node nd_i based on the file block m_i, the assigned value is equal to $H'(m_i)$, where the hash function $H' : \{0,1\}^* \rightarrow \mathbb{G}_1$ is seen as a random oracle. Note that the hash values are affected to leaf nodes in the increasing order of the blocks, i.e. nd_1 corresponds to the hash of the first block m_1, nd_2 corresponds to the hash of the first block m_2, and so on. A parent node of nd_i and nd_{i+1} has a value computed as $H'(H'(m_i)||H'(m_{i+1}))$, where $||$ is the concatenation sign (for an odd index i). The auxiliary authentication information (AAI) Ω_i of a leaf node nd_i for the file block m_i is a set of hash values chosen from its upper levels, so that the root value rt can be computed through (m_i, Ω_i).

When the client desires to add, remove or change a data block on its stored data, it has first to inform the server of such wish. The client's request R contains the type of operation that has to be performed (insertion, deletion or modification) as well as the position where the operation will be done. Using such information, the server is able to select the appopriate elements from the current version of the MHT and set the AAI Ω_i as the tuple of all these elements. More precisely, the elements are the hash values assigned on the nodes of the MHT: the elements can be found either on the leaves or the internal nodes, at different level, in function of what needs the client to create the updated version of the MHT according to the data operation.

Let $m = (m_1, \cdots, m_8)$ be a file stored on the server. In Figure 1, we highlight which leaves/internal nodes are required in order to insert a data block m'_3 after the data block m_2, to delete the block m_3 and to modify the block m_3 by replacing it with m'_3. Each hash value $H'(m_i)$ is assigned to a leaf node nd_i, for $i = 1, \cdots, n$, in the increasing order. The blocks that sustain the operations are alloted to leaf nodes with a disk inside. The hash values that should be included into Ω_3 are affected to leaves or internal nodes with a cross inside. Such values will allow the client to compute the new version of the MHT, including the new root rt' that will be then signed.

The AAI Ω_i is also required by the TPA to check the proof of data possession. More precisely, in addition to the proof of data possession ν, the server also forwards values rt_{server}, $H'(m_i)$ and Ω_i for $i \in I$, according to the blocks indicated by $chal$. With such elements, the TPA is able to construct the current MHT and verifies that the root that it obtained is equal to rt_{server}. If such verification is successful, then the TPA proceeds to validate the proof of



N.B.: $\mathsf{H}' = H'(H'(H'(m_5)||H'(m_6))||H'(H'(m_7)||H'(m_8)))$

Figure 1: MHTs for the file $m = (m_1, \cdots, m_8)$ at the Initialization phase, at the Modification phase (changing the block m_3 into m_3'), at the Deletion phase (removing the block m_3), and at the Insertion phase (adding the block m_3' after the block m_2).

data possession itself.

We recall that the client generates the verification metadata for each block of a file $m = (m_1, \cdots, m_n)$, and sends both the file blocks and the corresponding verification metadata to the server. Now, the client has to also construct a MHT for such a file: given a hash function $H' : \{0,1\}^* \to \mathbb{G}_1$, it computes $H'(m_i)$ for $i = 1, \cdots, n$, and assigns the values $H'(m_i)$ to each leaf of the MHT in the increasing order. Then, it calculates the hash values of the internal nodes until computing the root rt of the MHT, following the construction definition of such a tree. Given a digital signature scheme $\mathbf{SS} = (\mathbf{SS.KeyGen}, \mathbf{SS.Sign}, \mathbf{SS.Verify})$, it signs the root rt using the signing secret key $\mathbf{SS.}sk \leftarrow \mathbf{SS.KeyGen}(\lambda)$ and obtains the signature $\sigma_{rt} \leftarrow \mathbf{SS.Sign}(\mathbf{SS.}sk, rt)$. Finally, it sends H' and σ_{rt} to the server. The client also computes the verifying public key $\mathbf{SS.}pk \leftarrow \mathbf{SS.KeyGen}(\lambda)$ and shares it with the TPA.

Upon receiving the ordered collection $\mathbb{F} = \{m_i\}_{i=1,\cdots,n}$ of the file blocks, the ordered collection $\mathbb{E} = \{T_{m_i}\}_{i=1,\cdots,n}$ of the corresponding verification metadata, the hash function H' and the signature σ_{rt}, the server first constructs the MHT such that each hash value $H'(m_i)$ is assigned to each leaf of the MHT in the increasing order. It then obtains a root rt_{server} and sends it to the client. The latter then runs $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt}, rt_{server})$ and gets an answer in $\{0,1\}$. If the answer is equal to 1, then the client knows that the server correctly downloaded its files (the roots of their respective MHTs are identical), and proceeds. Otherwise, then the client knows that the server did not obtain the same MHT, thus does not correctly stores the data, and aborts.

Data Operations. When the client wants to add a block m after the block m_i, remove m_i or modify m_i by replacing it with m'_i, for $i = 1, \cdots, n$, it first sends a request to the server containing information such as the type of operation to be performed and its location (index of the block).

Upon each request, the server needs to return the AAI Ω_i of the block to be updated (in order to reconstruct the current MHT and build the new one after the data update) and the last signed root value σ_{rt} provided by the client. The latter is so able to authenticate the AAI Ω_i by verifying the given signed root value σ_{rt} with the root rt of the reconstructed MHT. If the AAI are successfully authenticated, then the client proceeds; otherwise, it aborts.

Thereafter, in order to get the new MHT, the client computes the value $H'(m)$ for insertion or $H'(m'_i)$ for modification. Note that it does not need to compute $H(m_i)$ for deletion. In the new MHT, for the operation:

- *insertion:* the block m "takes" the position of and "becomes" the block m_{i+1}, the block m_{i+1} "takes" the position of and "becomes" the block m_{i+2} and so on, until the block m_n that has a new position and "becomes" the block m_{n+1}.
- *deletion:* the block m_{i+1} "takes" the position of and "becomes" the deleted block m_i, the block m_{i+2} "takes" the position of and "becomes" the block m_{i+1} and so on, until the block m_n that "takes" the position of and "becomes" the block m_{n-1}.
- *modification:* m'_i simply "takes" the position of m_i.

The client then signs the updated root rt' obtained in the new MHT by running **SS.Sign**, gets the signature $\sigma_{rt'}$ and forwards it to the server, in addition to (m, T_m) for insertion and $(m'_i, T_{m'_i})$ for modification. The server builds the MHT following the client's operation request and gives the resulting rt'_{server} to the client. Yet, the client runs $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt'}, rt'_{server})$ to get an answer in $\{0,1\}$. If the answer is equal to 1, then the client knows that the server correctly updated the data (the roots of their respective MHTs are identical), and proceeds (in particular, it can delete from its local storage $(m, T_m, \Omega_i, \sigma_{rt'})$ for insertion, $(\Omega_i, \sigma_{rt'})$ for deletion or $(m'_i, T_{m'_i}, \Omega_i, \sigma_{rt'})$ for modification). Otherwise, then the client knows that the server did not obtain the same MHT, thus did not correctly perform the operation on the data, and thus aborts.

Note that the client may ask the TPA to challenge the server after each update process in order to check the server's behavior.

6.2. Construction

Let $\Pi = (\mathbf{KeyGen}, \mathbf{TagGen}, \mathbf{PerfOp}, \mathbf{CheckOp}, \mathbf{GenProof}, \mathbf{CheckProof})$ be a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy such as the one given in [5]. Let $\mathbf{SS} = (\mathbf{SS.KeyGen}, \mathbf{SS.Sign}, \mathbf{SS.Verify})$ be a secure Digital Signature scheme. The MHT-based Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\mathbf{MHT.}\Pi = (\mathbf{MHT.KeyGen}, \mathbf{MHT.TagGen}, \mathbf{MHT.PerfOp}$, $\mathbf{MHT.GenProof}, \mathbf{MHT.CheckProof})$ is as follows:

$\mathbf{MHT.KeyGen}(\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$. The client first runs $\mathbf{KeyGen}(\lambda) \rightarrow (pk, sk)$ and $\mathbf{SS.KeyGen}(\lambda)$ $\rightarrow (\mathbf{SS.}pk, \mathbf{SS.}sk)$. More precisely, let $\mathbf{GroupGen}(\lambda)$ be an algorithm that, on input the security parameter λ, generates the cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order $p = p(\lambda)$ with bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Let g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then, $s+1$ elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$ and $a \in_R \mathbb{Z}_p$ are choosen randomly.

The client sets its public key $\mathsf{pk} = (pk, \mathbf{SS.}pk) = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, h_1, \cdots, h_s, g_2^a, \mathbf{SS.}pk)$ and its secret key $\mathsf{sk} = (sk, \mathbf{SS.}sk) = (a, \mathbf{SS.}sk)$.

$\mathbf{MHT.TagGen}(\mathsf{pk}, \mathsf{sk}, m) \rightarrow \mathsf{T}_m$. The client runs $\mathbf{TagGen}(pk, sk, m) \rightarrow T'_m = (T'_{m_1}, \cdots, T'_{m_n})$ $\in \mathbb{G}_1^n$ such that $T'_{m_i} = (\prod_{j=1}^s h_j^{m_{i,j}})^{-sk} = (\prod_{j=1}^s h_j^{m_{i,j}})^{-a} = \prod_{j=1}^s h_j^{-a \cdot m_{i,j}}$ for $i = 1, \cdots, n$. It also chooses a hash function $H' : \{0,1\}^* \rightarrow \{0,1\}^*$ seen as a random oracle Then, it creates the MHT according to the file m as follows. For $i = 1, \cdots, n$, the client computes $H'(m_i)$ and assigns this value to the i-th leaf. Once the n leaves refer to the n hashed values, the client starts to construct the resulting MHT, and obtains the root rt. Finally, the client signs the root by running $\mathbf{SS.Sign}(\mathbf{SS.}sk, rt) \rightarrow \sigma_{rt}$. Using the hash values, it computes the verification metadata as follows: $T_{m_i} = H'(m_i)^{-sk} \cdot T'_{m_i} = (H'(m_i) \cdot \prod_{j=1}^s h_j^{m_{i,j}})^{-sk} =$ $(H'(m_i) \cdot \prod_{j=1}^s h_j^{m_{i,j}})^{-a} = H'(m_i)^{-a} \cdot \prod_{j=1}^s h_j^{-a \cdot m_{i,j}}$ for $i = 1, \cdots, n$.

Then, the client stores all the file blocks m in an ordered collection $\mathbb{F}$ and the corresponding verification metadata T_m in an ordered collection $\mathbb{E}$. It forwards these two collections and (H', σ_{rt}) to the server.

Once the server received $(\mathbb{F}, \mathbb{E}, H')$, it generates the MHT corresponding to the data uploaded by the client. It sends the resulting root rt_{server} to the client.

Upon getting the root rt_{server}, the client runs $\mathbf{SS.Verify}(\mathbf{SS.}pk, \sigma_{rt}, rt_{server}) \rightarrow answer$. If $answer = 0$, then the client stops the process. Otherwise, it proceeds and deletes $(\mathbb{F}, \mathbb{E}, \sigma_{rt})$ from its local storage and keeps H' for further data operations.

$\mathbf{MHT.PerfOp}(\mathsf{pk}, \mathbb{F}, \mathbb{E}, R = (\text{operation}, i), info = (m_i, T_{m_i}, \sigma_{rt'})) \rightarrow (\mathbb{F}', \mathbb{E}', rt_{server})$. First, the client sends a request R to the server. Such request $R = (\text{operation}, i)$ should contain at least the type of operation and the position where such operation will be performed.

Upon receiving the request R, the server selects the AAI Ω_i from the MHT that the client needs in order to generate the root rt' of the updated MHT. It sends Ω_i to the client.

Once the client received Ω_i, it first authenticates it: if Ω_i is not the current AAI, then it aborts; otherwise, it constructs the updated MHT following the update that it wants the server to perform on its stored data. It calculates the new root rt' and signs it by running

$\mathbf{SS.Sign}(\mathbf{SS}.sk, rt') \rightarrow \sigma_{rt'}$. Then, the client sends $info = (m_i, T_{m_i}, \sigma_{rt'})$ (note that m_i and T_{m_i} are not necessary in case of deletion).

After receiving the element $info$ from the client, the server first updates the MHT, calculates the new root rt'_{server} and sends this value to the client.

Upon getting the root rt'_{server}, the client runs $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt'}, rt'_{server}) \rightarrow answer'$. If $answer' = 0$, then the client stops the process. Otherwise, it proceeds and deletes $(m_i, T_{m_i}, \sigma_{rt'})$ from its local storage.

$\mathbf{MHT.GenProof}(\mathsf{pk}, F, chal, \Sigma) \rightarrow (\nu, rt_{server}, \{H'(m_i), \Omega_i\}_{i \in I})$. After a time period to be agreed between the client and the TPA, the latter prepares a challenge $chal$ to be sent to the server as follows: it chooses a subset $I \subseteq [1, n_{max}]$ (n_{max} is the maximum number of blocks after operations), randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$.

Then, after receiving the challenge $chal$ which indicates the specific blocks for which the TPA on behalf of the client wants a proof of data possession, the server runs **GenProof** $(pk, F, chal, \Sigma) \rightarrow \nu$ such that $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c) \in \mathbb{G}_1^{2s+1}$. More precisely, it sets the ordered collection $F = \{m_i\}_{i \in I} \subset \mathbb{F}$ of blocks and an ordered collection $\Sigma = \{T_{m_i}\}_{i \in I} \subset \mathbb{E}$ which are the verification metadata corresponding to the blocks in F. It then selects at random $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1 = h_1^{r_1}, \cdots, R_s = h_s^{r_s}$. It also sets $b_j = \sum_{(i,v_i) \in chal} m_{i,j} \cdot v_i + r_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $B_j = h_j^{b_j}$ for $j = 1, \cdots, s$, and $c = \prod_{(i,v_i) \in chal} T_{m_i}^{v_i}$.

Moreover, the server prepares the latest version of the stored root's signature σ_{rt} provided by the client, the root rt_{server} of the current MHT as well as the hash values $H'(m_i)$ and the AAI Ω_i for the challenged blocks, such that the current MHT can be constructed using $\{H'(m_i), \Omega_i\}_{i \in I}$. Finally, it returns $(\nu, \sigma_{rt}, rt_{server}, \{H'(m_i), \Omega_i\}_{i \in I})$ to the TPA.

$\mathbf{MHT.CheckProof}(\mathsf{pk}, chal, \nu, \sigma_{rt}, rt_{server}, \{H'(m_i), \Omega_i\}_{i \in I}) \rightarrow \{\text{"success"}, \text{"failure"}\}$. After receiving the set $\{H'(m_i), \Omega_i\}_{i \in I}$ from the server, the TPA first constructs the MHT and calculates the root rt_{TPA}. It then checks that $rt_{server} = rt_{TPA}$: if not, then it aborts; otherwise, it runs $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt}, rt_{server}) \rightarrow answer$. If $answer = 0$, then the TPA stops the process. Otherwise, it proceeds and checks whether the following equation holds:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \overset{?}{=} e(\prod_{\substack{(i,v_i) \\ \in chal}} H'(m_i), g_2) \cdot e(\prod_{j=1}^{s} B_j, g_2) \tag{5}$$

If Eq. 5 holds, then the TPA returns "success" to the client; otherwise, it returns "failure" to the client.

Correctness. Supposing that the correctness holds for the systems Π and **SS**, if all the algorithms of $\mathbf{MHT}.\Pi$ are correctly generated, then the above scheme is correct. For the proof of data possession:

$$\begin{aligned}
e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) &= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} T_{m_i}^{v_i}, g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} (H'(m_i)^{-a} \cdot \prod_{j=1}^{s} h_j^{m_{i,j}\cdot(-a)\cdot v_i}), g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} H'(m_i), g_2\right) \cdot e(\prod_{j=1}^{s} h_j^{\sum_{\substack{(i,v_i)\\ \in chal}} m_{i,j}\cdot v_i + r_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} H'(m_i), g_2\right) \cdot e(\prod_{j=1}^{s} h_j^{b_j}, g_2) \\
&= e\left(\prod_{\substack{(i,v_i)\\ \in chal}} H'(m_i), g_2\right) \cdot e(\prod_{j=1}^{s} B_j, g_2)
\end{aligned}$$

6.3. Comments

Let $\mathbf{SS} = (\mathbf{SS.KeyGen}, \mathbf{SS.Sign}, \mathbf{SS.Verify})$ be a secure digital signature scheme. One concrete and simple example in the random oracle model is $\mathbf{SS.Sign}(\mathbf{SS}.sk, rt) = (H'(rt))^a = \sigma_{rt}$, where $H' : \{0,1\}^* \to \mathbb{G}_1$ is the hash function used to construct the MHT and $a = sk \leftarrow \mathbf{KeyGen}(\lambda)$ from the DPDP Π with Public Verifiability and Data Privacy.

A possible security concern is the non-authentication of the TPA. We treat our scheme as publicly verifiable, meaning that anyone has the possibility to check that the server correctly stores the data. Therefore, a malicious user (not necessarily a client of the server) is able to proceed such verification. To avoid such experience, the client can choose and authenticate a particular TPA and the server then is able to verify that the person who wants to audit it, is the TPA selected by the client [19]. More precisely, client, TPA and server proceed as follows. When the client is uploading the data on the server, it also chooses a TPA to check on its behalf the integrity of its data, and asks to the TPA its identity. The latter encrypts its identity ID_{TPA} under the client's public key pk and sends the resulting ciphertext to the client. Then, the client decrypts the latter to recover ID_{TPA} and includes $\mathbf{SS.Sign}(\mathbf{SS}.sk, ID_{TPA})$ into the elements (the data, the verification metadata, the signature of the MHT root and the hash function H') that are sent to the server. Later, during each challenge-response process, the TPA encrypts its identity ID_{TPA} under the server's public key pk_{server} and sends the resulting ciphertext to the server. Then, the server uses its associated secret key sk_{server} to recover the identity ID_{TPA} and runs $\mathbf{SS.Verify}(\mathbf{SS.}pk, \mathbf{SS.Sign}(\mathbf{SS.}sk, ID_{TPA}), ID_{TPA})$ to check the validity of the identity. We assume that the server and a malicious non-authenticated TPA cannot collude together, i.e. the server does not accept to communicate with a non-authenticated TPA.

Note that the updating proof is no longer needed since the client has to authenticate the AAI sent by the server anyway, and to verify that the updated MHT root rt'_{server} is identical to the client's one rt'. We also argue that the TPA can be required to challenge the server after each update requested by the client.

6.4. Security against the Server

6.4.1. Discrete Logarithm (DL) Problem

Please refer to the paragraph 5.3.1 for the definition.

6.4.2. Proof

For any PPT adversary $\mathcal{A}$ who winds the game, there is a challenger $\mathcal{B}$ that wants to break the DL problem by interacting with $\mathcal{A}$ as follows.

KeyGen. $\mathcal{B}$ runs $\mathbf{GroupGen}(\lambda) \to (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and selects two generators g_1 and g_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then, it randomly chooses s elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$ and an element $a \in_R \mathbb{Z}_p$. Let a hash function $H' : \{0,1\}^* \to \mathbb{G}_1$ be controlled by $\mathcal{B}$ as follows. Upon receiving a query m_{i_l} to the random oracle H' for some $l \in [1, q_{H'}]$:

- If (m_{i_l}, θ_l, W_l) exists in $L_{H'}$, return W_l.
- Otherwise, choose $\theta_l \in_R \mathbb{Z}_p$ at random and compute $W_l = g_1^{\theta_l}$. Put (m_{i_l}, θ_l, W_l) in $L_{H'}$ and return W_l as answer.

Note that H' is supposed to be collision resistant and the digital signature scheme **SS** to be unforgeable. $\mathcal{B}$ gives $\mathcal{A}$ the public key pk that contains the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$ as well as the public key $\mathbf{SS.}pk \leftarrow \mathbf{SS.KeyGen}(\lambda)$. It sets the secret key $sk = a$ and keeps it, as well as $\mathbf{SS.}sk \leftarrow \mathbf{SS.KeyGen}(\lambda)$.

Adaptive queries. $\mathcal{A}$ has access to the tag generation oracle $\mathcal{O}_{TG}$ as follows. It first adaptively selects blocks m_i, for $i = 1, \cdots, n$. $\mathcal{B}$ splits each block m_i into s sectors $m_{i,j}$. Then, it computes $T_{m_i} = (W \cdot \prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (W \cdot \prod_{j=1}^{s} h_j^{m_{i,j}})^{-a}$, for $i = 1, \cdots, n$, such that if (m_i, θ, W) exists in $L_{H'}$, then the value W is returned. Otherwise, an element $\theta \in_R \mathbb{Z}_p$ is chosen at random, $W = g^\theta$ is computed, and (m_i, θ, W) is put in $L_{H'}$. $\mathcal{B}$ also creates the MHT resulting from the blocks m_i and gets the corresponding root rt. It signs rt by running $\sigma_{rt} \leftarrow \mathbf{SS.Sign}(\mathbf{SS.}sk, rt)$. It gives the verification metadata T_{m_i} and their corresponding hash values W, along with σ_{rt} to $\mathcal{A}$. $\mathcal{A}$ sets an ordered collection $\mathbb{F} = \{m_1, \cdots, m_n\}$ of blocks and an ordered collection $\mathbb{E} = \{T_{m_1}, \cdots, T_{m_n}\}$ which are the verification metadata corresponding to the blocks in $\mathbb{F}$. It has also access to the data operation performance oracle $\mathcal{O}_{DOP}$ as follows. Repeatedly, $\mathcal{A}$ selects a block $m_{l'}$ and the corresponding element $info_{l'}$

and forwards them to $\mathcal{B}$. The signature of the root $\sigma_{rt'}$ is included in $info_{l'}$, from the signature algorithm **SS.Sign**(**SS.**sk, rt'). l' denotes the rank where $\mathcal{A}$ wants the data operation to be performed. Moreover, $m_{l'} = \perp$ in the case of a deletion, since only the rank is needed to perform this kind of operation. Then, $\mathcal{A}$ outputs a new ordered collection $\mathbb{F}'$ (containing the updated version of the block $m_{l'}$), a new ordered collection $\mathbb{E}'$ (containing the updated version of the verification metadata $T_{m_{l'}}$) and a new root $rt'_{\mathcal{A}}$ corresponding to the updated MHT. $\mathcal{B}$ runs the algorithm **SS.Verify** on the values $\sigma_{rt'}$ and $rt'_{\mathcal{A}}$ and aborts if the answer is equal to 0; proceeds otherwise.

Setup. $\mathcal{A}$ selects blocks m_i^* and the corresponding elements $info_i^*$, for $i \in \mathcal{I}$, and forwards them to the challenger who checks the data operations. In particular, the first $info_i^*$ indicates a full re-write.

Challenge. $\mathcal{B}$ chooses a subset $I \subseteq \mathcal{I}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$. It forwards $chal$ as a challenge to $\mathcal{A}$.

Forge. Upon receiving the challenge $chal$, the resulting proof of data possession on the correct stored file m should be $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ and pass the Eq. 5. However, $\mathcal{A}$ generates a proof of data possession on an incorrect stored file $\tilde{m}$ as $\tilde{\nu} = (R_1, \cdots, R_s, \tilde{B}_1, \cdots, \tilde{B}_s, \tilde{c})$, such that r_j is randomly chosen from $\mathbb{Z}_p$, $R_j = h_j^{r_j}$, $\tilde{b}_j = \sum_{(i,v_i) \in chal} \tilde{m}_{i,j} \cdot v_i + r_j$ and $\tilde{B}_j = h_j^{\tilde{b}_j}$, for $j = 1, \cdots, s$. It also sets $\tilde{c} = \prod_{(i,v_i) \in chal} T_{\tilde{m}_i}^{v_i}$. Finally, it returns $\tilde{\nu} = (R_1, \cdots, R_s, \tilde{B}_1, \cdots, \tilde{B}_s, \tilde{c})$ to $\mathcal{B}$. If the proof of data possession still pass the verification, then $\mathcal{A}$ wins. Otherwise, it fails. We define $\Delta b_j = \tilde{b}_j - b_j$, for $j = 1, \cdots, s$. At least one element of $\{\Delta b_j\}_{j=1,\cdots,s}$ is non-zero.

Analysis. We prove that if the adversary can win the game, then a solution to the DL problem is found, which contradicts the assumption that the DL problem is hard in $\mathbb{G}_1$. Let assume that the server wins the game. Then, according to Eq. 5, we have $e(c, g_2^a) \cdot e(\prod_{j=1}^s R_j, g_2) = e(\prod_{\substack{(i,v_i) \\ \in chal}} H'(m_i), g_2) \cdot e(\prod_{j=1}^s \tilde{B}_j, g_2)$. Since the proof $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ is a correct one, we also have $e(c, g_2^a) \cdot e(\prod_{j=1}^s R_j, g_2) = e(\prod_{\substack{(i,v_i) \\ \in chal}} H'(m_i), g_2) \cdot e(\prod_{j=1}^s B_j, g_2)$. Therefore, we get that $\prod_{j=1}^s \tilde{B}_j = \prod_{j=1}^s B_j$. We can re-write as $\prod_{j=1}^s h_j^{\tilde{b}_j} = \prod_{j=1}^s h_j^{b_j}$ or even as $\prod_{j=1}^s h_j^{\Delta b_j} = 1$. For two elements $g, h \in \mathbb{G}_1$, there exists $x \in \mathbb{Z}_p$ such that $h = g^x$ since $\mathbb{G}_1$ is a cyclic group. Without loss of generality, given $g, h \in \mathbb{G}_1$, each h_j could randomly and correctly be generated by computing $h_j = g^{y_j} \cdot h^{z_j} \in \mathbb{G}_1$ such that y_j and z_j are random values of $\mathbb{Z}_p$. Then, we have $1 = \prod_{j=1}^s h_j^{\Delta b_j} = \prod_{j=1}^s (g^{y_j} \cdot h^{z_j})^{\Delta b_j} = g^{\sum_{j=1}^s y_j \cdot \Delta b_j} \cdot h^{\sum_{j=1}^s z_j \cdot \Delta b_j}$. Clearly, we can find a solution to the DL problem. More specifically, given $g, h = g^x \in \mathbb{G}_1$, we can compute $h = g^{\frac{\sum_{j=1}^s y_j \cdot \Delta b_j}{\sum_{j=1}^s z_j \cdot \Delta b_j}} = g^x$ unless the denominator is zero. However, as we defined in the game, at least one element of $\{\Delta b_j\}_{j=1,\cdots,s}$ is non-zero. Since z_j is a random element of $\mathbb{Z}_p$, the denominator is zero with probability equal to $1/p$, which is negligible. Thus, if $\mathcal{A}$ wins the game, then a solution of the DL problem can be found with probability equal to $1-\frac{1}{p}$, which contradicts the fact that the DL problem is assumed to be hard in $\mathbb{G}_1$. Therefore, for $\mathcal{A}$, it is computationally infeasible to win the game and generate an incorrect proof of data possession which can pass the verification.

Moreover, the simulation of the random oracle H' is not entirely perfect. Let's consider the event that $\mathcal{A}$ has queried before the *Challenge* phase m_i^* to H'. This event happens with probability $1/p$. Except for the case above, the simulation of H' is perfect. We recall that $\mathcal{A}$ makes at most $q_{H'}$ random oracle queries. The simulation of the tag generation oracle $\mathcal{O}_{TG}$ is perfect. The simulation of the data operation performance oracle $\mathcal{O}_{DOP}$ is also perfect. The proof is completed.

6.5. Data Privacy against the TPA

6.5.1. Decisional s-Strong Diffie Hellman (DSDH) Problem

Please refer to the paragraph 5.4.1 for the definition.

6.5.2. Proof

We presume that the digital signature scheme **SS** is unforgeable and the hash function H' is collision resistant. For any PPT adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{B}$ that wants to break the $s+1$-DSDH assumption by interacting with the adversary $\mathcal{A}$ as follows.

KeyGen. $\mathcal{B}$ runs $\mathbf{GroupGen}(\lambda) \to (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and receives the $s+1$-DSDH instance $(g_1, g_1^{\beta}, \cdots, g_1^{\beta^{s+1}}, g_2, g_2^{\beta})$. $\mathcal{B}$ sets $\mu = 0$ when the output Z of the $s+1$-DSDH assumption is equal to $g_1^{\beta^{s+2}}$; otherwise, it sets $\mu = 1$ when the output Z of the $s+1$-DSDH assumption is a random element in $\mathbb{G}_1$. Then, it sets the elements $h_1 = g_1^{\beta}, \cdots, h_s = g_1^{\beta^s}, h_{s+1} = g_1^{\beta^{s+1}}$ and implicitly fixes the secret key $sk = a = \beta$ by setting $g_2^a = g_2^{\beta}$. $\mathcal{B}$ also controls the hash function $H' : \{0,1\}^* \to \mathbb{G}_1$ as follows. Upon receiving a query m_{i_l} to the random oracle H' for some $l \in [1, q_{H'}]$:

- If $(m_{i_l}, \theta_l, V_l, W_l)$ exists in $L_{H'}$, return V_l and W_l.
- Otherwise, choose $\theta_l \in_R \mathbb{Z}_p$ at random and compute $V_l = g_1^{-\theta_l}$ and $W_l = h_1^{-\theta_l}$. Put $(m_{i_l}, \theta_l, V_l, W_l)$ in $L_{H'}$ and return W_l as answer.

It sets the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$. The challenger has also access to a secure digital signature scheme **SS** and runs the algorithm $\mathbf{SS.KeyGen}(\lambda)$ to obtain the public and secret key pair $(\mathbf{SS.}pk, \mathbf{SS.}sk)$. $\mathcal{B}$ sets the public key $\mathsf{pk} = (pk, \mathbf{SS.}pk)$ and forwards it to $\mathcal{A}$. It keeps the secret keys sk and $\mathbf{SS.}sk$ and the hash function H'.

Queries. $\mathcal{A}$ makes tag generation queries as follows: $\mathcal{A}$ first selects a file $m = (m_1, \cdots, m_n)$ and sends it to $\mathcal{B}$. Then, the challenger splits each block m_i into s sectors $m_{i,j}$. Then, it computes $T_{m_i} = W \cdot \prod_{j=1}^{s} g_1^{\beta^j \cdot (-\beta) \cdot m_{i,j}} = W \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{i,j}}$, such that if (m_i, θ, V, W) exists in $L_{H'}$, then the value W is returned; otherwise, an element $\theta \in_R \mathbb{Z}_p$ is chosen at random, $V = g_1^{\theta}$ and $W = h_1^{-\theta}$ are computed, and (m_i, θ, V, W) is put in $L_{H'}$. $\mathcal{B}$ also creates the MHT resulting from the file m using the values V such that if (m_i, θ, V, W) exists in $L_{H'}$, then the value V is returned; otherwise, an element $\theta \in_R \mathbb{Z}_p$ is chosen at random, $V = g_1^{\theta}$ and $W = h_1^{-\theta}$ are computed, and (m_i, θ, V, W) is put in $L_{H'}$. It finally gets the corresponding root rt. It gives the verification metadata $T_m = (T_{m_1}, \cdots, T_{m_n})$ along with the root's signature $\sigma_{rt} \leftarrow \mathbf{SS.Sign}(\mathbf{SS.}sk, rt)$ to $\mathcal{A}$.

Challenge. $\mathcal{A}$ first gives to the challenger two files $m_0 = (m_{0,1}, \cdots, m_{0,n})$ and $m_1 = (m_{1,1}, \cdots, m_{1,n})$ of equal length. $\mathcal{B}$ randomly selects a bit $b \in_R \{0,1\}$ and for $i = 1, \cdots, n$, splits each block $m_{b,i}$ into s sectors $m_{b,i,j}$. Then, it computes $T_{m_{b,i}} = W_b \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{b,i,j}}$, such that if $(m_{b,i}, \theta_b, V_b, W_b)$ exists in $L_{H'}$, then the value W_b is returned; otherwise, an element $\theta_b \in_R \mathbb{Z}_p$ is chosen at random, $V_b = g_1^{-\theta_b}$ and $W_b = h_1^{-\theta_b}$ are computed, and $(m_{b,i}, \theta_b, V_b, W_b)$ is put in $L_{H'}$. $\mathcal{B}$ also creates the MHT resulting from the file m_b using the values V such that if $(m_{b,i}, \theta_b, V_b, W_b)$ exists in $L_{H'}$, then the value V_b is returned; otherwise, an element $\theta_b \in_R \mathbb{Z}_p$ is chosen at random, $V_b = g_1^{-\theta_b}$ and $W_b = h_1^{-\theta_b}$ are computed, and $(m_{b,i}, \theta_b, V_b, W_b)$ is put in $L_{H'}$. It finally gets the corresponding root rt_b. It gives the verification metadata $T_{m_b} = (T_{m_{b,1}}, \cdots, T_{m_{b,n}})$ along with the root's signature $\sigma_{rt_b} \leftarrow \mathbf{SS.Sign}(\mathbf{SS.}sk, rt_b)$ to $\mathcal{A}$.

Without loss of generality, $\mathcal{A}$ generates a challenge on one block only. It chooses a subset $I = \{i^*\} \subseteq \{1, \cdots, n\}$, randomly chooses an element $v_{i^*} \in_R \mathbb{Z}_p$ and sets $chal = (i^*, v_{i^*})$. It forwards $chal$ as a challenge to $\mathcal{B}$. Upon receiving the challenge $chal$, $\mathcal{B}$ selects an ordered collection $F_b = \{m_{b,i^*}\}$ of blocks and an ordered collection $\Sigma_b = \{T_{m_{b,i^*}}\}$ which are the verification metadata corresponding to the blocks in F_b such that $T_{m_{b,i^*}} = W_b \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{b,i^*,j}}$. It then randomly chooses $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1^* = h_1^{\beta^2 \cdot r_1} = h_3^{r_1} = g_1^{\beta^3 \cdot r_1}, \cdots, R_{s-1}^* = h_{s-1}^{\beta^2 \cdot r_{s-1}} = h_{s+1}^{r_{s-1}} = g_1^{\beta^{s+1} \cdot r_{s-1}}$ and $R_s^* = Z^{r_s}$. It implicitly fixes $b_1 = \beta^2 \cdot m_{b,i^*,1} \cdot v_{i^*} + r_1, \cdots, b_{s-1} = \beta^2 \cdot m_{b,i^*,s-1} \cdot v_{i^*} + r_{s-1}$ by computing $B_1^* = h_1^{b_1} = h_1^{\beta^2 \cdot m_{b,i^*,1} \cdot v_{i^*}} \cdot h_1^{\beta^2 \cdot r_1} = h_3^{m_{b,i^*,1} \cdot v_{i^*}} \cdot h_3^{r_1} = g_1^{\beta^3 \cdot m_{b,i^*,1} \cdot v_{i^*}} \cdot g_1^{\beta^3 \cdot r_1}, \cdots, B_{s-1}^* = h_{s-1}^{b_{s-1}} = h_{s-1}^{\beta^2 \cdot m_{b,i^*,s-1} \cdot v_{i^*}} \cdot h_{s-1}^{\beta^2 \cdot r_{s-1}} = h_{s+1}^{m_{b,i^*,s-1} \cdot v_{i^*}} \cdot h_{s+1}^{r_{s-1}} = g_1^{\beta^{s+1} \cdot m_{b,i^*,s-1} \cdot v_{i^*}} \cdot g_1^{\beta^{s+1} \cdot r_{s-1}}$ and $B_s^* = Z^{m_{b,i^*,s} \cdot v_{i^*}} \cdot Z^{r_s}$. It sets $c^* = T_{m_{b,i^*}}^{v_{i^*}} = W_b^{v_{i^*}} \cdot \prod_{j=1}^{s} g_1^{-\beta^{j+1} \cdot m_{b,i^*,j} \cdot v_{i^*}}$ as well. Finally, $\mathcal{B}$ returns $\nu^* = (R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*, c^*)$ along with (V_b, Ω_{i^*}), where Ω_{i^*} is the AAI needed to create the MHT based on V_b.

If $\mu = 0$ then $Z = g_1^{\beta^{s+2}}$. Therefore, the proof of data possession is a valid random proof for the file m_b. Otherwise, if $\mu = 1$, then Z is random value in $\mathbb{G}_1$. Since Z is random, the values R_s and B_s will be random elements of $\mathbb{G}_1$ from the adversary's view and the proof of data possession contains no information about m_b.

Guess. $\mathcal{A}$ returns a bit b'. If $b = b'$, $\mathcal{B}$ will output $\mu' = 0$ to indicate that it was given a $s+1$-DSDH tuple; otherwise it will output $\mu' = 1$ to indicate that it was given a random tuple.

Analysis. We first recall that the digital signature scheme **SS** is assumed to be unforgeable and the hash function H' is supposed to be collision resistant. We prove that the verification metadata and the proof of data possession given to $\mathcal{A}$ are correctly distributed. In the case where $\mu = 1$, $\mathcal{A}$ gains no information about b. Therefore, we have $Pr[b \neq b' | \mu = 1] = 1/2$. Since $\mathcal{B}$ guesses $\mu' = 1$ when $b \neq b'$, we have $Pr[\mu = \mu' | \mu = 1] = 1/2$. If $\mu = 0$, then $\mathcal{A}$ sees an upload of m_b. The adversary's advantage in this situation is negligible by definition, i.e. equal to a given ϵ. Therefore, we have $Pr[b \neq b' | \mu = 0] = 1/2 + \epsilon$. Since $\mathcal{B}$ guesses $\mu' = 0$ when $b = b'$, we have $Pr[\mu = \mu' | \mu = 0] = 1/2 + \epsilon$. The value $T_{m_{b,i}}$ is equal to $(H'(m_{b,i}) \cdot \prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk}$ and $sk = a$. The elements $\mathbf{SS}.sk$ and H' are kept secret from $\mathcal{A}$. The simulation of the random oracle H' is entirely perfect since $\mathcal{A}$ does have access to it. In addition, $R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*$ are statically indistinguishable with the actual outputs corresponding to m_0 or m_1. Thus, the answers given to $\mathcal{A}$ are correctly distributed. The proof is completed.

6.6. Performance

We compare the MHT-based scheme with the original one presented in [5]. The client is able to verify the first upload and the updates without the help of the TPA. It authenticates the AAI given by the server, uses its MHT root and compares it with the one provided by the server. The TPA is required for data integrity checks: it regularly challenges the server to prove that the latter correctly stores the data of the client. The client and the TPA share a list containing block indices (or at least the maximum number n_{max} of blocks given after the operations that the client wanted the server to perform on its data) and the server stores the data, the verification metadata, the signature of the client's root (also regularly updated after the data operations) and the hash function H'. The server might construct the MHT each time this is needed or stores the latest version of the entire tree.

Obviously, the communication and the computation overheads grow. First of all, when the client is uploading the file for the first time, it has to compute all the elements in order to construct the MHT resulting from the file and compares the root with the one from the server's MHT. However, such elements are hash values, that is, elements easily computable.

Moreover, the server stores more elements: the MHT and the signature of the client's root for each file, along with the file itself and the corresponding verification metadata as in [5]. Note that for each time the server is asked to perform an operation, it has to update the MHT accordingly. Nevertheless, as suggested in [5], the server has huge computation and storage resources, thus this should not be a constraint for it. Note that the server can only store m_i, T_{m_i}, σ_{rt} and H' and constructs the MHT whenever necessary; however, this option does not seem more attractive.

Then, the communication burden increases between the client and the server, especially for the data operation process. More precisely, the client has first to inform the server that it wants to make an operation on its data and asks the necessary information. From this request, the server sends some AAI to the client in order to start the data operation process. Upon receiving such information, the client has to authenticate it and then create the MHT according to the data update to be performed. Then, as for the first upload, the client sends the resulting information back to the server. Using such elements, the server can perform the operation on the stored data and on the MHT, and obtain a new version of the root that is forwarded back to the client. Such root is compared with the one given by the client (under the form of a signature) on the client's side. However, we no longer need the help of the TPA to check updates. Thus, the server no longer has to generate an updating proof through the algorithm **PerfOp** and the TPA no longer has to run the algorithm **CheckOp** to verify the correctness of the updating proof, compared to the scheme in [5]. Overall, implementing a MHT-based scheme seems to fairly affect the computation and communication overheads.

In the MHT-based system, we assume that the client deletes all the elements related to the file (blocks, verification metadata, root signatures, MHTs) from its local storage, except a list containing block indices that are needed for requesting a data integrity verification. Such list may be shared with the TPA since the latter works on behalf of the client in the challenge-response process. In order to reduce the communication overhead between the client and the server when the former prepares an update operation, we can force the client to store on its local storage the current MHT or a list of the hash values corresponding to all the leaves of the current MHT. Doing this, the client does not need to ask for AAI before updating the data and so, the server does not need to send it back to the client. Note that the latter no longer needs to authenticate it, since it keeps the current version of the MHT.

If the client keeps the entire MHT, the hash values at internal nodes and the root are already computed; however it has to store $2^{k+1}-1$ elements for $n=2^k$. Moreover, when an operation is performed, the client does not need to calculate all the hash values at internal nodes, but only the ones on the path from the modified leaves until the root, as well as the root itself. If the client keeps only a list of hash values corresponding to all the leaves of the MHT, meaning that it stores only n elements, it has to compute all the intermediary hash values at upper levels and the root each time it inserts, deletes or modifies data blocks. This option is possible if the client possesses enough resources for storage and computation.

The MHT-based construction seems less practical and efficient. Communication and computation burdens appear in order to obtain the desired security standards against the server and the TPA. The communication overheads increase between the client and the server. The computation overheads for the client raises also, although the client is limited in resources. The storage space of the server should be bigger, since the server has to create and possibly stores MHTs for each client it has. The TPA has to provide more computational resources for each client in order to ensure valid data integrity checks. Nevertheless, experiments might show that the time gap between the algorithms in the scheme proposed in [5] and the ones in the MHT-based scheme is acceptable.

Comparison with the Existing Schemes. The MHT is an Authenticated Data Structure (ADS) that allows the client and the TPA to check that the server correctly stores and updates the data blocks.

Erway et al. [3] proposed the first DPDP scheme. The verification of the data updates is based on a modified ADS, called Rank-based Authentication Skip List (RASL). This provides authentication of the data block indices, which ensures security in regards to data block dynamicity. However, public verifiability is not reached. Note that such ADS with bottom-up levelling limits the insertion operations. For instance, if the leaf nodes are at level 0, any data insertion that creates a new level *below* the level 0 will bring necessary updates of all the level hash values and the client might not be able to verify.

Wang et al. [18] first presented a DPDP with Public Verifiability using MHT. However, security proofs and technical details lacked. The authors revised the aforementioned paper [18] and proposed a more complete paper [19] that focuses on dynamic and publicly verifiable PDP systems based on BLS signatures. To achieve the dynamicity property, they employed MHT. Nevertheless, because the check of the block indices is not done, the server can delude the client by corrupting a challenged block as follows: it is able to compute a valid proof with other non-corrupted blocks. Thereafter, in a subsequent work [17], Wang et al. suggested to add randomization to the above system [19], in order to guarantee that the server cannot deduce the contents of the data files from the proofs of data possession.

Liu et al. [8] constructed a PDP protocol based on MHT with top-down levelling. Such protocol satisfies dynamicity and public verifiability. They opted for such design to let leaf nodes be on different levels. Thus, the client and the TPA have both to remember the total number of data blocks and check the block indices from two directions (leftmost to rightmost and vice versa) to ensure that the server does not delude the client with another node on behalf of a file block during the data integrity checking process.

In this paper, the dynamic and publicly verifiable PDP scheme is based on MHT with bottom-up levelling, such that data block indices are authenticated. Such tree-based construction guarantees security for both dynamicity and public verifiability properties. In the next section, we explain how we can ensure that the server cannot successfully generate a correct proof of data possession without storing all the file blocks and the TPA cannot get any information about the challenged file blocks.

7. Conclusion

We provided solutions to solve the technical issues of the Dynamic PDP scheme with Public Verifiability and Data Privacy proposed in [5]. These solutions manage to overcome replay and replace attacks by including Index-Hash Tables (IHTs) or Merkle Hash Trees (MHTs) into the original construction given in [5], as well as indistinguishability-based data privacy model problems by either presenting a weaker security model for the scheme presented in [5] or by proving the initial security level for both the IHT-based and the MHT-based constructions.

[1] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D., 2007. Provable data possession at untrusted stores. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. CCS '07. ACM, New York, NY, USA, pp. 598–609.

[2] Ateniese, G., Di Pietro, R., Mancini, L. V., Tsudik, G., 2008. Scalable and efficient provable data possession. In: Proceedings of the 4th International Conference on Secu-

rity and Privacy in Communication Netowrks. SecureComm '08. ACM, New York, NY, USA, pp. 9:1–9:10.

[3] Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R., 2009. Dynamic provable data possession. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. CCS '09. ACM, New York, NY, USA, pp. 213–222.

[4] Fan, X., Yang, G., Mu, Y., Yu, Y., Apr. 2015. On indistinguishability in remote data integrity checking 58 (4), 823–830.

[5] Gritti, C., Susilo, W., Plantard, T., 2015. Efficient dynamic provable data possession with public verifiability and data privacy. In: Proceedings of the 20th Australasian Conference on Information Security and Privacy. ACISP '15. Springer-Verlag, Berlin, Heidelberg.

[6] Hao, Z., Zhong, S., Yu, N., Sep. 2011. A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability. IEEE Trans. on Knowl. and Data Eng. 23 (9), 1432–1437.

[7] Le, A., Markopoulou, A., 2012. Nc-audit: Auditing for network coding storage. CoRR abs/1203.1730.

[8] Liu, C., Ranjan, R., Yang, C., Zhang, X., Wang, L., Chen, J., 2014. Mur-dpa: Top-down levelled multi-replica merkle hash tree based secure public auditing for dynamic big data storage on cloud. IACR Cryptology ePrint Archive 2014, 391.

[9] Merkle, R. C., 1979. Secrecy, authentication, and public key systems. Ph.D. thesis, Stanford, CA, USA, aAI8001972.

[10] Shacham, H., Waters, B., 2008. Compact proofs of retrievability. In: Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology. ASIACRYPT '08. Springer-Verlag, Berlin, Heidelberg, pp. 90–107.

[11] Wang, B., Li, B., Li, H., 2012. Knox: Privacy-preserving auditing for shared data with large groups in the cloud. In: Proceedings of the 10th International Conference on Applied Cryptography and Network Security. ACNS'12. Springer-Verlag, Berlin, Heidelberg, pp. 507–525.

[12] Wang, B., Li, B., Li, H., 2012. Oruta: privacy-preserving public auditing for shared data in the cloud. IEEE Transactions on Cloud Computing 2 (1), 43–56.

[13] Wang, B., Li, B., Li, H., 2015. Panda: Public auditing for shared data with efficient user revocation in the cloud. IEEE T. Services Computing 8 (1), 92–106.

[14] Wang, C., Chow, S. S., Wang, Q., Ren, K., Lou, W., 2013. Privacy-preserving public auditing for secure cloud storage. IEEE Transactions on Computers 62 (2), 362–375.

[15] Wang, C., Wang, Q., Ren, K., Cao, N., Lou, W., Jan. 2012. Toward secure and dependable storage services in cloud computing. IEEE Trans. Serv. Comput. 5 (2), 220–232.

[16] Wang, C., Wang, Q., Ren, K., Lou, W., 2009. Ensuring data storage security in cloud computing. In: in Proc. of IWQoS'09.

[17] Wang, C., Wang, Q., Ren, K., Lou, W., 2010. Privacy-preserving public auditing for data storage security in cloud computing. In: Proceedings of the 29th Conference on Information Communications. INFOCOM'10. IEEE Press, Piscataway, NJ, USA, pp. 525–533.

[18] Wang, Q., Wang, C., Li, J., Ren, K., Lou, W., 2009. Enabling public verifiability and data dynamics for storage security in cloud computing. In: Proceedings of the 14th European Conference on Research in Computer Security. ESORICS'09. Springer-Verlag, Berlin, Heidelberg, pp. 355–370.

[19] Wang, Q., Wang, C., Ren, K., Lou, W., Li, J., May 2011. Enabling public auditability and data dynamics for storage security in cloud computing. IEEE Trans. Parallel Distrib. Syst. 22 (5), 847–859.

[20] Yang, K., Jia, X., Jul. 2012. Data storage auditing service in cloud computing: Challenges, methods and opportunities. World Wide Web 15 (4), 409–428.

[21] Yu, S., Wang, C., Ren, K., Lou, W., 2010. Achieving secure, scalable, and fine-grained data access control in cloud computing. In: Proceedings of the 29th Conference on Information Communications. INFOCOM'10. IEEE Press, Piscataway, NJ, USA, pp. 534–542.

[22] Yu, Y., Au, M. H., Mu, Y., Tang, S., Ren, J., Susilo, W., Dong, L., 2014. Enhanced privacy of a remote data integrity-checking protocol for secure cloud storage. International Journal of Information Security, 1–12.

[23] Yu, Y., Ni, J., Au, M., Mu, Y., Wang, B., Li, H., 2014. On the security of a public auditing mechanism for shared cloud data service. Services Computing, IEEE Transactions on PP (99), 1–1.

[24] Yu, Y., Niu, L., Yang, G., Mu, Y., Susilo, W., 2014. On the security of auditing mechanisms for secure cloud storage. Generation Computer Systems: the international journal of grid computing: theory, methods and applications 30 (1), 127–132.

[25] Zhu, Y., Ahn, G.-J., Hu, H., Yau, S. S., An, H. G., Hu, C.-J., 2013. Dynamic audit services for outsourced storages in clouds. IEEE Transactions on Services Computing 6 (2), 227–238.

[26] Zhu, Y., Wang, H., Hu, Z., Ahn, G.-J., Hu, H., Yau, S. S., 2011. Dynamic audit services for integrity verification of outsourced storages in clouds. In: Proceedings of the 2011

ACM Symposium on Applied Computing. SAC '11. ACM, New York, NY, USA, pp. 1550–1557.