Improvements on Efficient Dynamic Provable Data Possession scheme 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
{cjpg967,rc517}@uowmail.edu.au, {wsusilo,thomaspl}@uow.edu.au

June 30, 2015

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 processing the data integrity checking.

In this paper, we propose several solution 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 current scheme and a new construction to enhance the security and to achieve the strongest data privacy notion.

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 (PDP) 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 [4], the authors presented an efficient practical PDP system by adopting asymmetric pairings to gain efficiency and reduce 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 clearly outperforms all the existing schemes in the literature. 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. Hence, the authors claimed that their scheme was better than the existing schemes in terms of practicality.

Nevertheless, we find two attacks on the Dynamic Provable Data Possession (DPDP) scheme with Public Verifiability and Data Privacy [4]. 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 find a solution that forces us to switch from the standard model to the random oracle model. However, the practicality of our scheme is not impacted: no exponentiation and four pairings intead of three are needed during the proof of data possession check.

Another attack threatens the data privacy in [4]. Indeed, in our 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 metatdata corresponding to each block of m_b. Given these elements as well as the public key, the attacker is able to discover which fle was chosen by the challenger.

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

2 Preliminaries

2.1 DPDP: Definition and Construction

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 [8]. 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).

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, 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 $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$ and its secret key $sk = a$.

$\mathbf{TagGen}(pk, sk, m) \rightarrow T_m$. 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.

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

$\mathbf{PerfOp}(pk, \mathbb{F}, \mathbb{E}, info = (\text{insertion}, \frac{2i+1}{2}, m_{\frac{2i+1}{2}}, T_{m_{\frac{2i+1}{2}}})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data insertion 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 "insertion" of the data operation to be performed, the index $\frac{2i+1}{2}$ denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$), the file block $m_{\frac{2i+1}{2}}$ to be inserted, and the corresponding verification metadata $T_{m_{\frac{2i+1}{2}}}$ to be inserted, for $i = 0, \cdots, n$. More precisely, $m_{\frac{2i+1}{2}}$ 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}. Finally, it outputs the updated file block collection $\mathbb{F}'$ containing $m_{\frac{2i+1}{2}}$, the updated verification metadata collection $\mathbb{E}'$ containing $T_{m_{\frac{2i+1}{2}}}$, and the related updating proof ν'. The server sends ν' to the TPA.

After receiving the elements $\frac{2i+1}{2}$, $m_{\frac{2i+1}{2}}$ and $T_{m_{\frac{2i+1}{2}}}$ from the client, for $i = 0, \cdots, n$, 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 $w_{\frac{2i+1}{2}} \in_R \mathbb{Z}_p$ and sets $c_j = m_{\frac{2i+1}{2},j} \cdot w_{\frac{2i+1}{2}} + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_{\frac{2i+1}{2}}}^{w_{\frac{2i+1}{2}}}$. Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{deletion}, i)) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data deletion 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 "deletion" of the data operation to be performed, and the index i denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$). The server deletes the existing file block m_i, and the corresponding verification metadata T_{m_i}, for $i = 1, \cdots, n$. More precisely, 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 $T_{m_{n-1}}$. Finally, it outputs the updated file block collection $\mathbb{F}'$ that does not contain m_i anymore, the updated verification metadata collection $\mathbb{E}'$ that does not contain T_{m_i} anymore, and the related updating proof ν'. The server sends ν' to the TPA. The deletion operation stops when the number of blocks is equal to 0.

After receiving an index $i = 1, \cdots, n$ 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 $w_i \in_R \mathbb{Z}_p$ and sets $c_j = m_{i,j} \cdot w_i + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_i}^{w_i}$, where m_i and T_{m_i} are the existing file block and verification metadata to be deleted respectively. Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{modification}, i, m_i', T_{m_i'})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data 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 "modification" of the data operation to be performed, the index i denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$), the file block m_i' which replaces the existing block m_i, and the corresponding verification metadata $T_{m_i'}$ which replaces the existing verification metadata T_{m_i}, for $i = 1, \cdots, n$. 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**. It outputs the updated verification metadata collection $\mathbb{F}'$ replacing m_i by m_i', the updated verification metadata collection $\mathbb{E}'$ replacing T_{m_i} by $T_{m_i'}$, and the related updating proof ν'. The server sends ν' to the TPA. We allow the client to make full re-write updates, meaning that all the file blocks $m_1, \cdots, m_n$ are replaced by $m_1', \cdots, m_n'$ and all the verification metadata $T_{m_1}, \cdots, T_{m_n}$ are replaced by $T_{m_1'}, \cdots, T_{m_n'}$.

After receiving the elements i, m_i' and $T_{m_i'}$ 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 $w_i \in_R \mathbb{Z}_p$ and sets $c_j = m_{i,j}' \cdot w_i + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_i'}^{w_i}$. 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$(pk, \nu') \rightarrow \{$"success", "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". We assume that the answer is then forwarded to the client. We omit this part of the process.

The TPA has to check whether the following equation holds:

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

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

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 $chal$.

We assume that a first challenge $chal_C$ is generated by the client and forwarded to the TPA. Then, the TPA generates a challenge $chal$ from $chal_C$ and sends it to the server. In particular, if the client wants to check the integrity of its data without the help of the TPA, then $chal_C = chal$. We omit the process done by the client at this point.

After receiving a challenge $chal_C$ from the client, the TPA prepares a challenge $chal$ to send to the server as follows. First, it chooses a subset $I \subseteq]0, n+1[\mathbb{Q}$, 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 $F = \{m_i\}_{i \in I} \subset \mathbb{F}$ 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$(pk, chal, \nu) \rightarrow \{$"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. We omit this part of the process.

The TPA has to check whether the following equation holds:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \quad \stackrel{?}{=} \quad e(\prod_{j=1}^{s} B_j, g_2) \tag{2}$$

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

Correctness. 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**$(pk, chal, \nu)$.

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

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

For the proof of data possession, we have:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \quad = \quad e\left(\prod_{\substack{(i,v_i) \\ \in chal}} T_{m_i}^{v_i}, g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) = e(\prod_{j=1}^{s} h_j^{\sum\limits_{\substack{(i,v_i) \\ \in chal}} m_{i,j} \cdot v_i + r_j}, g_2) = e(\prod_{j=1}^{s} h_j^{b_j}, g_2) = e(\prod_{j=1}^{s} B_j, g_2)$$

2.2 Security Models and Proofs

2.2.1 Assumption: 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 $\mathbb{G}_1$.

2.2.2 Security against the server

The definition of the scheme, mentioned below, follows the ones from [1] and [2]. We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**). Let a data possession game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ be as follows:

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

Adaptive queries. $\mathcal{A}$ makes adaptive queries through the intermediary of two oracles. The adversary is given access to a tag generation oracle $\mathcal{O}_{TG}$ as follows. $\mathcal{A}$ chooses a first block m_1 and forwards it the challenger. $\mathcal{C}$ computes the corresponding verification metadata $T_{m_1} \leftarrow$ **TagGen**(pk, sk, m_1) and gives it to the adversary. The adversary keeps on the same queries process with $\mathcal{C}$ for the verification metadata $T_{m_2} \leftarrow$ **TagGen**$(pk, sk, m_2), \cdots, T_{m_n} \leftarrow$ **TagGen** (pk, sk, m_n), where the blocks $m_2, \cdots, m_n$ are chosen by $\mathcal{A}$. Then, the adversary 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, the adversary is given access to a data operation performance oracle $\mathcal{O}_{DOP}$ as follows. $\mathcal{A}$ submits to the challenger a block m_i, for $i = 1, \cdots, n$, and the corresponding value $info_i$ about the data operation that the adversary wants to perform. The adversary runs the algorithm **PerfOp** and outputs a new file blocks ordered collection $\mathbb{F}'$, a new metadata ordered collection $\mathbb{E}'$, and the corresponding updating proof ν'. $\mathcal{C}$ checks the value ν' by running the algorithm **CheckOp**(pk, ν') and gives back the resulting answer belonging to {"success", "failure"} to the adversary. If the answer is "failure", then the challenger aborts; otherwise, it proceeds. The above interaction between $\mathcal{A}$ and $\mathcal{C}$ can be repeated.

Setup. The adversary 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 the adversary, 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). The challenger verifies the data operations.

Challenge. The final version of the blocks $m_i \in \mathcal{I}$ is considered such that these blocks were created according to the data operations requested by the adversary, and verified and accepted by the challenger in the previous step. The challenger 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. $\mathcal{C}$ then takes an ordered collection $F = \{m_{i_1}, \cdots, m_{i_k}\} \subset \mathbb{F}$ and the corresponding verification metadata ordered collection $\Sigma = \{T_{m_{i_1}}, \cdots, T_{m_{i_k}}\} \subset \mathbb{E}$, 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. The adversary generates a proof of data possession ν on *chal*. Then, the challenger runs **CheckProof**$(pk, chal, \nu)$ and gives the answer belonging to {"success", "failure"} to $\mathcal{A}$. If the answer is "success" then the adversary wins.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **Check- Proof**) 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 $\mathcal{C}$ 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 $\mathcal{E}$.

Proof. For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{C}$ that interacts with the adversary $\mathcal{A}$ as follows.

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

Adaptive queries. $\mathcal{A}$ has access to the tag generation oracle $\mathcal{O}_{TG}$ as follows. It first adaptively selects blocks m_i, for $i = 1, \cdots, n$. $\mathcal{C}$ splits each block m_i, for $i = 1, \cdots, n$ into s sectors $m_{i,j}$. Then, it computes $T_{m_i} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-a}$, for $i = 1, \cdots, n$, and gives them to $\mathcal{A}$. The adversary 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 access to the data operation performance oracle $\mathcal{O}_{DOP}$ as follows. Repeatedly, the adversary selects a block m_l and the corresponding element $info_l$

and forwards them to the challenger. 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. Then, $\mathcal{A}$ outputs a new file blocks ordered collection $\mathbb{F}'$ (containing the updated version of the block m_l), a new verification metadata 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 $C_j = h_j^{c_j}$. $\mathcal{C}$ 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. The adversary 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. The challenger 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. 2. 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 the challenger. 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. 2, we have $e(c, g_2^a) \cdot e(\prod_{j=1}^s R_j, g_2) = e(\prod_{j=1}^s \tilde{B}_j, g_2)$. Since the proof $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ is a correct one, we also have $e(c, g_2^a) \cdot e(\prod_{j=1}^s R_j, g_2) = e(\prod_{j=1}^s B_j, g_2)$. 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 the adversary 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 the adversary, 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 tag generation oracle $\mathcal{O}_{TG}$ is perfect. The simulation of the data operation performance oracle $\mathcal{O}_{DOP}$ is almost perfect except when the challenger aborts. This happens the data operation was not correclty performed. As previously, we can prove that if the adversary can pass the updating proof, then a solution to the DL problem is found. Following the above analysis and according to Eq. 1, if the adversary 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 the adversary, it is computationally infeasible to generate an incorrect updating proof which can pass the verification. The proof is completed.

2.2.3 Privacy against the TPA

The definition of the scheme, mentioned below, follows the one from [12]. We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**). Let a data privacy game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ be as follows:

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

Queries. $\mathcal{A}$ 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. $\mathcal{C}$ randomly selects a bit $b \in_R \{0, 1\}$, computes $T_{m_{b,i}} \leftarrow$ **TagGen**$(pk, sk, m_{b,i})$ for $i = 1, \cdots, n$ and

gives them to $\mathcal{A}$. Then, the adversary 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. The adversary forwards *chal* to $\mathcal{C}$.

Generation of the Proof. The challenger outputs a proof of data possession $\nu^* \leftarrow$ **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 collectection 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. The adversary returns a bit b'. $\mathcal{A}$ wins if $b' = b$.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **Check- Proof**) 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 $|Pr[b' = b] - \frac{1}{2}|$.

Proof. For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{C}$ that interacts with the adversary $\mathcal{A}$ as follows.

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

Queries. $\mathcal{A}$ 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. $\mathcal{C}$ 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 $T_{m_{b,i}} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}$, for $i = 1, \cdots, n$, and gives them to $\mathcal{A}$.

Challenge. The adversary chooses a subset $I \subseteq \{1, \cdots, n\}$, 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 $\mathcal{C}$.

Generation of the Proof. Upon receiving the challenge *chal*, the challenger selects an ordered collection $F = \{m_i\}_{i \in I}$ of blocks and an ordered collection $\Sigma = \{T_{m_i}\}_{i \in I}$ which are the verification metadata corresponding to the blocks in F such that $T_{m_i} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-a}$, for $i \in I$. It then randomly chooses $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1^* = h_1^{r_1}, \cdots, R_s^* = h_s^{r_s}$. It also randomly selects $b_1, \cdots, b_s \in \mathbb{Z}_p$ and computes $B_1^* = h_1^{b_1}, \cdots, B_s^* = h_s^{b_s}$. It sets $c^* = \prod_{(i,v_i) \in chal} T_{m_i}^{v_i}$ as well. Finally, the challenger returns $\nu^* = (R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*, c^*)$.

Guess. The adversary returns a bit b'.

Analysis. 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. We now prove that the verification metadata and the proof of data possession given to the adversary are correctly distributed. The value $T_{m_{b,i}}$ is equal to $(\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}$. Since $sk = a$ is kept secret from $\mathcal{A}$, the above simulation is perfect. For a block file m_b, there exists $v_{b,i}$, for $(i, v_{b,i}) \in chal_b$, such that $b_{b,j} = \sum_{(i,v_{b,i}) \in chal_b} m_{b,i,j} \cdot v_{b,i} + r_{b,j}$. In addition, $R_{b,1}, \cdots, R_{b,s}, B_{b,1}, \cdots, B_{b,s}$ are statically indistinguishable with the actual outputs corresponding to m_0 or m_1. Thus, the answers given to the adversary are correctly distributed. The proof is completed.

3 Replace and Replay Attacks

3.1 Replace Attack

As usual, the client generates the verification metadata for a file m that it wants to upload on the server.

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

Then, the server is asked to generate a proof of data possession. However, 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.

GenProof$(pk, F, chal, \Sigma) \rightarrow \nu$. After receiving a challenge $chal_C$ from the client, the TPA prepares a challenge $chal$ to send to the server as follows. First, it chooses a subset $I \subseteq]0, n+1[_{\mathbb{Q}}$, 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 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. 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 $j = 1, \cdots, s$, then $B_j = h_j^{b_j}$ for $j = 1, \cdots, s$, and $c = \prod_{(i,v_i) \in chal} {T_{m_1}}^{v_i}$. 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:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \stackrel{?}{=} e(\prod_{j=1}^{s} B_j, g_2) \tag{3}$$

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

Correctness. For the proof of data possession, we have:

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

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

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' amd $T_{m_i'}$.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{modification}, i, m_i', T_{m_i'})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. 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 got 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$(pk, \nu') \rightarrow \{$"success", "failure"$\}$. The TPA has to check whether the following equation holds:

$$e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) \stackrel{?}{=} e(\prod_{j=1}^{s} C_j, g_2) \tag{4}$$

If Eq. 7 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, we have:

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

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

3.3 The First Solution for the Replace and Replay Attacks: Index-Hash Table

3.3.1 Idea of the First Solution

A solution to avoid the first 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 in the publicly verifiable scheme proposed in [8].

A solution to avoid the second 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 append 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) when generating the verification metadata $T_{m_i'}$. 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 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 with a certain percentage of the data. To be quite clear, we let the client conserve a table of indices of the file blocks that are kept on the server along with their version numbers vnb_i. Such a table is then either forward to the TPA or used when the client sends a challenge $chal_C$ to the TPA in order to send a challenge

chal to the server. 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	-

Such table is called Index-Hash Table (IHT). This table can be seen as a summary of the hash value of each block as well as a history of the data changes made by the client. Our IHT is composed of several columns: one for the block index, one for the version number, and one for some 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 [**?**].

The IHT gives a better security level against the untrusted server, although this leads into an increase of the complexity of our system.

3.3.2 The New Construction

We now explain how overcome the two aforementioned attacks. Let the hash function $H : \mathbb{Q} \times \mathbb{N} \to \mathbb{G}_1$ be a random oracle. Such a hash function H is included in the public key pk during the **Setup** phase. Then, the verification metadata $T_m = (T_{m_1}, \cdots, T_{m_n})$ of the file $m = (m_1, \cdots, m_n)$ is generated as follows. For each block m_i, we compute

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

This solution is easy to implement and does not devaluate the practicality of our scheme, although the scheme becomes secure in the random oracle model instead of the standard model.

This time, the server has to use the correct blocks to generate the proof of data possession. Indeed, the TPA use the hashes of the indices that are challenged to check this proof.

GenProof$(pk, F, chal, \Sigma) \to \nu$. After receiving a challenge $chal_C$ from the client, the TPA prepares a challenge *chal* to send to the server as follows. First, it chooses a subset $I \subseteq]0, n+1[_{\mathbb{Q}}$, 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 $F = \{m_i\}_{i \in I} \subset \mathbb{F}$ 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$(pk, chal, \nu) \to \{$"success", "failure"$\}$. The TPA has to check whether the following equation holds:

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

If Eq. 8 holds, then the TPA returns "success" to the client; otherwise. it returns "failure" to the client.

Correctness.

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

This also works for the operation "modification" and the updating proof: the verification metadata $T_{m_i'}$ of the modified block m_i' contains the version number $vnb_i' = vnb_i + 1$, where vnb_i is the version number of the file block m_i to be modified.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{modification}, i, m_i', T_{m_i'})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. After receiving the elements i, m_i' and $T_{m_i'}$ from the client, the server prepares the updating proof as follows. It first 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 $c_j = m_{i,j}' \cdot w_i + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_i'}^{w_i}$. 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$(pk, \nu') \rightarrow \{$"success", "failure"$\}$. The TPA has to check whether the following equation holds:

$$
e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) \stackrel{?}{=} e(H(i, vnb_i'), g_2) \cdot e(\prod_{j=1}^{s} C_j, g_2) \tag{6}
$$

If Eq. 7 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, we have:

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

The security of our model still holds, but in the random oracle model instead of the standard model.

3.3.3 Security against the server

During the **KeyGen** phase, the challenger $\mathcal{B}$ gives $\mathcal{A}$ the public key pk that contains a 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 $((i_l, vnb_{i_l}), \omega_l, W_l)$ exists in L_H, return W_l.
- Otherwise, choose $\omega_l \in_R \mathbb{Z}_p$ at random and compute $W_l = g_1^{\omega_l}$. Put $((i_l, vnb_{i_l}), \omega_l, W_l)$ in L_H and return W_l as answer.

During the Adaptive Queries phase, $\mathcal{A}$ has access to the tag generation oracle $\mathcal{O}_{TG}$ as follows. It first adaptively selects blocks m_i, for $i = 1, \cdots, n$. $\mathcal{C}$ splits each block m_i, for $i = 1, \cdots, n$ 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 $((i, vnb_i), \omega, W)$ exists in L_H, then the value W is returned. Otherwise, an element $\omega \in_R \mathbb{Z}_p$ is chosen at random, and $W = g^{\omega}$ is computed, and $((i, vnb_i), \omega, W)$ is put in L_H. It gives them to $\mathcal{A}$. The adversary 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 access to the data operation performance oracle $\mathcal{O}_{DOP}$ as follows. Repeatedly, the adversary selects a block $m_{l'}$ and the corresponding element $info_{l'}$ and forwards them to the challenger. 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 file blocks collection $\mathbb{F}'$ (containing the updated version of the block m_l), a new ordered verification metadata 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 $C_j = h_j^{c_j}$. $\mathcal{C}$ 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.

Analysis. 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. Except for the case above, the simulation of H is perfect. This event happens with probability $1/p$. We recall that the adversary makes at most q_H random oracle queries.

Adding the above analysis step into the paragraph 2.2.2, we obtain that the advantage of the adversary in winning the Data Possession Game is still negligible. The proof is completed.

3.3.4 Data Privacy against the TPA

We give the Data Privacy proof in Section 4, since other modifications and explanations have to be brought.

3.3.5 Performance

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 a comment (only in case of deletions) to each idnex, the extra burden does not seem excessive. Therefore, such a table does slightly affect the client's as well as the TPA's local storage and the communication between the client and the TPA increases a little bit 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 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 proof of data possession, the new version of our PDP scheme is still practical by adopting asymmetric pairings to gain efficiency and by still reducing the group exponentiation and pairing operations. In addition, our fresh protocol 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.

3.4 The Second Solution for the Replace and Replay Attacks: Merkle Hash Tree

3.4.1 Idea of the Second Solution

The second solution to avoid the replace and replay attacks is to implement a Merkle Hash Tree (MHT) for each file to order the file blocks. The MHT [7] 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. We construct the MHT 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 $H' : \{0,1\}^* \to \{0,1\}^*$ is seen as a random oracle. Note that hashed values are affected to leaf nodes in the ascending 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 $H'(H'(m_i)||H'(m_{i+1}))$. The auxiliary information Ω_i of a leaf node nd_i for a 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).

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 $H' : \{0,1\}^* \to \{0,1\}^*$, 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 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 $\mathbf{SS}.sk \leftarrow \mathbf{SS.KeyGen}$ and obtains the signature $\sigma_{rt} \leftarrow \mathbf{SS.Sign}(rt, \mathbf{SS}.sk)$. Finally, it sends H', the verifying public key $\mathbf{SS}.pk \leftarrow \mathbf{SS.KeyGen}$ and σ_{rt} to the server.

Upon receiving the file blocks $\{m_i\}_{i=1,\cdots,n}$, the corresponding verification metadata $\{T_{m_i}\}_{i=1,\cdots,n}$, the hash function H', the verifying public key $\mathbf{SS}.pk$ and 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 runs $\mathbf{SS.Verify}(\sigma_{rt}, \mathbf{SS}.pk)$ and sends the answer to the client. 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 the same), 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.

We stress that the hash function H' is an element only shared between the client and the server. If such a function is made public, then the scheme is no longer data private: the TPA may use the hash function on

random data blocks and compare the hash result with the one received through the proof of data possession.

We now explain how the operations are performed.

Insertion. When the client wants to add a block m after the block m_i for $i = 1, \cdots, n$, it first sends a request to the server telling that it wants to insert a block and where. Second, the server sends back the auxiliary information that is needed to update the MHT. This auxiliary information contains some hash values. Then, the client computes the value $H'(m)$ and using the auxiliary information, calculates the root rt' of the update MHT. In this new version of the MHT, 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}. It signs the root rt' to obtain the signature $\sigma_{rt'}$ and forwards m, T_m and $\sigma_{rt'}$ to the server. Later, the server reconstructs the MHT following the request of the client and gives back the answer from $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt'}, rt'_{server})$ to the client. 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 the same), and proceeds. 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 aborts.

Deletion. When the client wants to remove the block m_i for $i = 1, \cdots, n$, it first sends a request to the server tellin which block it wants to delete. Second, the server sends back the auxiliary information that is needed to update the MHT. This auxiliary information contains some hash values. In this case, the client does need to compute $H'(m)$. Then, the client calculates the root rt' of the update MHT using the auxiliary information. In this new version of the MHT, 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}. It signs the root rt' to obtain the signature $\sigma_{rt'}$ and forwards m, T_m and $\sigma_{rt'}$ to the server. Later, the server reconstructs the MHT following the request of the client and gives back the answer from $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt'}, rt'_{server})$ to the client. 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 the same), and proceeds. 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 aborts.

Modification. This is the simplest operation. When the client wants to modify a block m_i by replacing it with m'_i for $i = 1, \cdots, n$, it first sends a request to the server telling which block it wants to modify. Second, the server sends back the auxiliary information that is needed to update the MHT. This auxiliary information contains some hash values. Then, the client computes the value $H'(m'_i)$ and using the auxiliary information, calculates the root rt' of the update MHT. It signs the root rt' to obtain the signature $\sigma_{rt'}$ and forwards m'_i, $T_{m'_i}$ and $\sigma_{rt'}$ to the server. Later, the server reconstructs the MHT following the request of the client and gives back the answer from $\mathbf{SS.Verify}(\mathbf{SS}.pk, \sigma_{rt'}, rt'_{server})$ to the client. 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 the same), and proceeds. 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 aborts.

3.4.2 The New 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. Let $\mathbf{SS} = (\mathbf{SS.KeyGen}, \mathbf{SS.Sign}, \mathbf{SS.Verify})$ be a secure digital signature scheme. The Merkle Hash Tree based Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\mathbf{MHT}.\Pi = (\mathbf{MHT.KeyGen}, \mathbf{MHT.TagGen}, \mathbf{MHT.PerfOp}, \mathbf{MHT.CheckOp}, \mathbf{MHT.GenProof}, \mathbf{MHT.CheckProof})$ is as follows:

$\mathbf{MHT.KeyGen}(\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$. The client first runs $\mathbf{KeyGen}(\lambda) \rightarrow (pk, sk)$ and $\mathbf{SS.KeyGen}(\lambda) \rightarrow (\mathbf{SS}.pk, \mathbf{SS}.sk)$. It sets its public key $\mathsf{pk} = (pk, \mathbf{SS}.pk) = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a, \mathbf{SS}.pk)$ 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 $i = 1, \cdots, n$. Then, it creates the Merkle Hash Tree (MHT) according to the file m as follows. It first chooses a hash function $H' : \{0,1\}^* \rightarrow \{0,1\}^*$ seen as a random oracle. 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: $\mathbf{SS.Sign}(\mathbf{SS}.sk, rt) \rightarrow \sigma_{rt}$.

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 $(\mathbb{F}, \mathbb{E}, H', \sigma_{rt})$, it generates the MHT corresponding to the uploaded data by the client. Upon getting the root rt_{server}, it runs $\mathbf{SS.Verify}(\mathbf{SS}.sk, \sigma rt, rt_{server}) \rightarrow answer$ and sends $answer$ to the client. If $answer = 0$, then the client stops the process. Otherwise, it deletes $(\mathbb{F}, \mathbb{E}, H', \sigma_{rt})$ from its local storage.

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

Upon receiving the request R, the server selects the auxiliary information Ω_i from the MHT that the client needs in order to generate the root rt' of the update MHT. It sends Ω_i to the client. We give details about the value Ω_i below the construction.

Once the client received Ω_i, it constructs the update MHT following the updates that it wants the server to perform on its stored data. It calculates the new root rt' and signs it: $\mathbf{SS.Sign}(\mathbf{SS}.sk, rt') \rightarrow \sigma_{rt'}$. Then, the client sends $\sigma_{rt'}$ along with $info = (\text{operation}, i, m_i, T_{m_i}))$ (note that m_i and T_{m_i} are not necessary in case of deletion).

After receiving the elements $\sigma_{rt'}$ and $info$ from the client, the server first updates the MHT and calculates the new root rt'_{server}, it runs $\mathbf{SS.Verify}(\mathbf{SS}.sk, \sigma rt', rt'_{server}) \rightarrow answer'$ and sends $answer'$ to the client. If $answer' = 0$, then the client stops the process. Otherwise, it deletes $(m_i, T_{m_i}, \sigma_{rt'})$ from its local storage. Then, the server runs $\mathbf{PerfOp}(pk, \mathbb{F}, \mathbb{E}, info = (\text{operation}, i, m_i, T_{m_i})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$ such that $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$. Finally, it returns ν' to the TPA.

$\mathbf{MHT.CheckOp}(\mathsf{pk}, \nu') \rightarrow \{\text{“success”}, \text{“failure”}\}$. The TPA runs $\mathbf{CheckOp}(pk, \nu') \rightarrow \{\text{“success”}, \text{“failure”}\}$, i.e. it has to check whether the following equation holds:

$$e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) \quad \stackrel{?}{=} \quad e(\prod_{j=1}^{s} C_j, g_2) \tag{7}$$

If Eq. 7 holds, then the TPA returns “success” to the client; otherwise, it returns “failure” to the client.

$\mathbf{MHT.GenProof}(\mathsf{pk}, F, chal, \Sigma) \rightarrow \nu$. After possibly receiving a challenge $chal_C$ from the client or after a time period to be agreed between the client and the TPA, the TPA prepares a challenge $chal$ to send to the server as follows: it chooses a subset $I \subseteq]0, n+1[_{\mathbb{Q}}$, 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 $\mathbf{GenProof}(pk, F, chal, \Sigma) \rightarrow \nu$ such that $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c) \in \mathbb{G}_1^{2s+1}$. Finally, it returns ν to the TPA.

$\mathbf{MHT.CheckProof}(\mathsf{pk}, chal, \nu) \rightarrow \{\text{“success”}, \text{“failure”}\}$. The TPA runs $\mathbf{CheckProof}(pk, chal, \nu) \rightarrow \{\text{“success”}, \text{“failure”}\}$, i.e. it has to check whether the following equation holds:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \quad \stackrel{?}{=} \quad e(\prod_{j=1}^{s} B_j, g_2) \tag{8}$$

If Eq. 8 holds, then the TPA returns “success” to the client; otherwise. it returns “failure” to the client.

Correctness. Since 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.

Auxiliary Information 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 at block level) 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 auxiliary information Ω_i as the tuple of all these elements. More precisely, the elements are the hashed values assigned on the nodes of the MHT: the elements can be found either on the leaf 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. As we mentioned in the first part of this section, the auxiliary information Ω_i of a leaf node nd_i for a file block m_i is a set of hash values chosen from every of its upper level, so that the root value rt can be computed through (m_i, Ω_i).

Let $m = (m_1, \cdots, m_8)$ be a file stored on the server. In Figure 1, we highlight which leaf/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 hashed value $H'(m_i)$ is assigned to a leaf node nd_i, for $i = 1, \cdots, n$, in the ascending order. The blocks that sustain the operations are alloted to leaf nodes with a disk inside. The hashed values that should be included into Ω_3 are affected to leaf 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.

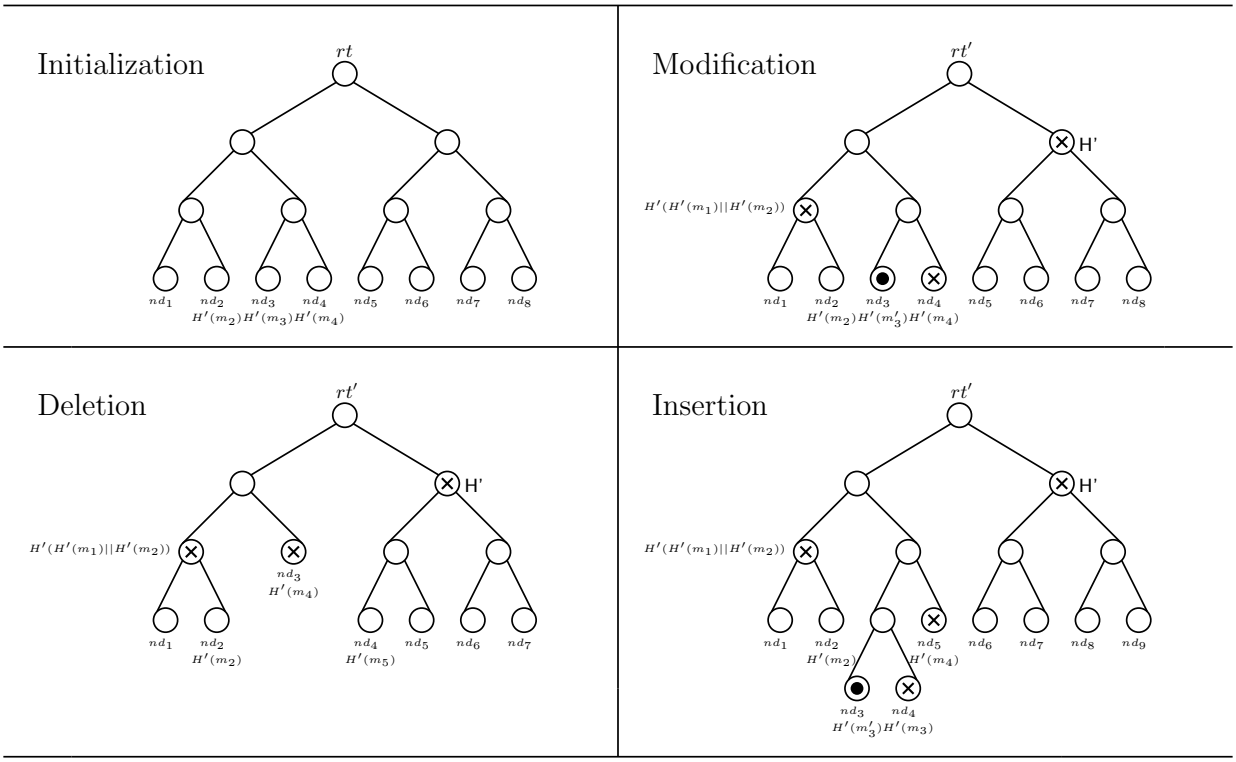

Figure 1: Merkle Hash Trees for the file $m = (m_1, \cdots, m_n)$ at the Initialization phase, at the Modification phase (changing the block m_3 into m'_3), at the Deletion phase (removing the block m_3), and at the Insertion phase (adding the block m'_3 after the block m_2).

3.4.3 Performance

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. However, such elements are only hash values, that is, elements easily computable.

The server stores more elements: the MHT for each file, along with the file itself and the corresponding verification metadata as previously. In addition, each time that the server is asked to perform an operation, it has to update the MHT accordingly. Nevertheless, as suggested in [4], the server has huge computation and storage resources, thus this should not be a constraint for it.

Then, the communication burden increases between the client and the server, especially for data operation processes. 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 auxiliary information to the client in order to start the data operation process. Upon receiving such information, the client has to create the MHT according to the updated version of the data. 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 to obtain a new version of them and gives the updating proof back to the TPA. Upon receiving the updating proof, the TPA checks it and sends the result to the client. Therefore, implementing a MHT-based scheme negatively affects the communication overhead.

Comparison with the Existing Schemes. In DPD system, three entities are involved: a client, a server and a TPA. 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. With the help of the node and some auxiliary information, the data block can be authenticated.

Erway et al. [2] proposed the first Dynamic PDP scheme. The verification of the data updates is based on a modified ADS, 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 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. [11] gave a dynamic and publicly verifiable PDP system 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 [10], Wang et al. suggested to add randomization to the above system [11], in order to guarantee cannot deduce the contents of the data files from proofs of data possession.

Liu et al. [6] 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.

Counter to the above systems, our dynamic and publicly verifiable PDP scheme is based on MHT with bottom-up levelling, such that data block indices are authenticated. In the next section, we explain how we can ensure that the TPA cannot get any information or just few details about the challenged file blocks. Such notion is called Data Privacy.

4 Attack against Data Privacy

We recalled the security model for Data Privacy against the TPA in Section 2.2.3. The definition of the model follows the one from [12], that is an enhancement of the model provided in [5]. The data privacy attack works as follows.

The TPA, who plays the role of the adversary, provides two equl-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 $(\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}$ for $i = 1, \cdots, n$.

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 last pairing requires only public elements to be computed. 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 one 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 The First Solution for the Data Privacy Attack: a Weaker Security Model

4.1.1 Idea of the First Solution

A first solution to avoid the aforementioned attack is to choose a weaker security model than the one proposed in [4]. The security model given in [4] 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, if we allow the TPA to be 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 processes. For instance, even if the TPA notices that it has verified the same block during two consecutive challenge-response processes, it only knows that this block appeared twice but it does not know more information about it. We recall that the TPA checks that the server correctly performed a data operation at block level, therefore this means that the TPA is aware of on which block the operation happened and so, it may be able to differentiate this block with another one on which another operation occured. Yet again, it does not have access too more details about these two blocks.

Thus, we argue that a security model based on one-wayness is sufficient for our purposes. 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 The New Security 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 $\mathcal{C}$ and an adversary $\mathcal{A}$ be as follows:

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

Setup. $\mathcal{C}$ chooses a file $m = m_1||\cdots||m_n$, computes $T_{m_i} \leftarrow \mathbf{TagGen}(pk, sk, m_i)$ for $i = 1, \cdots, n$. Then, it creates an ordered collection $\mathbb{F} = \{m_{b,1}, \cdots, m_{b,n}\}$ 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. The adversary forwards *chal* to $\mathcal{C}$.

Generation of the Proof. The challenger outputs a proof of data possession $\nu^* \leftarrow \mathbf{GenProof}(pk, F, chal, \Sigma)$ 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 $1 \leq i_j \leq n$, $1 \leq j \leq k$ and $1 \leq k \leq n$. $\mathcal{B}$ sends (ν', Σ) to the adversary.

Guess. The adversary 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 $\Pi = (\mathbf{KeyGen}, \mathbf{TagGen}, \mathbf{PerfOp}, \mathbf{CheckOp}, \mathbf{GenProof}, \mathbf{CheckProof})$ 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 $Adv_{\mathcal{A}}(\lambda)$. 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 The New Security Proof for the Original Scheme

The proof will be provided in the full version of this paper.

4.2 The Second Solution for the Data Privacy Attack: Index-Hash Table

4.2.1 Idea of the Second Solution

A second solution to avoid the above attack while keeping the same security level (i.e. indistinguishability) is to employ Index-Hash Tables (IHT). Please refer to Section for details about such construction.

In [3], the authors gave a data privacy model based on indistinguishability. They also noticed that two papers [11, 9] were not data private under the security model presented in [3], meaning that some information about the file is leaked to the TPA. More precisely, the TPA is able to distinguish files that are challenged. Note that such a security model is similar to one presented in [4].

The strongest notion for data privacy has been proposed by Fan et al. [3]. 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).

Setup. $\mathcal{B}$ runs the **KeyGen** algorithm to generate the pair of matching 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 Queries phase, and sends them to $\mathcal{B}$. The latter computes T_{m_0} and T_{m_1} by running the **TagGen** algorithm. 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

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

A proof of data possession ν has indistinguishability if for any PPT adversary $\mathcal{A}$, $Adv_{\mathcal{A}}(1^\lambda)$ is a negligible function of the security parameter λ.

4.2.2 The New Security Proof for the IHT-based Scheme

The proof will be provided in the full version of this paper.

5 Conclusion

We provided solutions to solve the technical issues of DPDP scheme with Public Verifiability and Data Privacy proposed in [4]. These solutions manage to overcome replay and replace attacks by including Index-Hash Tables (IHTs) or Merkle Hash Trees (MHTs) into the original construction, as well as indistinguishability-based data privacy model problems by either presenting a weaker security model for the original scheme or by proving such security level for the IHT-based construction.

References

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

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

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

[4] C. Gritti, W. Susilo, and T. Plantard. 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, Berlin, Heidelberg, 2015. Springer-Verlag.

[5] Z. Hao, S. Zhong, and N. Yu. 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, Sept. 2011.

[6] C. Liu, R. Ranjan, C. Yang, X. Zhang, L. Wang, and J. Chen. 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, 2014.

[7] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford, CA, USA, 1979. AAI8001972.

[8] H. Shacham and B. Waters. 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, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag.

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

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

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

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