DD-POR: Dynamic Operations and Direct Repair in Network Coding-based Proof of Retrievability

Kazumasa Omote and Tran Phuong Thao

Japan Advanced Institute of Science and Technology (JAIST),
1-1 Asahidai, Nomi, Ishikawa, Japan, 923-1211
{omote, tpthao}@jaist.ac.jp

Abstract. POR (Proof of Retrievability) is a protocol by which clients can distribute their data to cloud servers and can check if the data stored in the servers is available and intact. Based on the POR, the network coding is applied to improve network throughput. Although many network coding-based PORs have been proposed, most of them have not achieved the following practical features: direct repair and dynamic operations. In this paper, we propose the DD-POR (Dynamic operations and Direct repair in network coding-based POR) to address these shortcomings. When a server is corrupted, the DD-POR can support the direct repair in which the data stored in the corrupted server can be repaired using the data provided directly from the healthy servers. The client is thus free from the burden of data repair. Furthermore, the DD-POR allows the client to efficiently perform dynamic operations, i.e., modification, insertion and deletion.

Keywords: Proof of Retrievability, Network Coding, Direct Repair, Dynamic Operations

1 Introduction

Since the amount of data is increasing exponentially, data storage and data management become burdensome tasks of the clients. Therefore, storage providers called clouds are proposed to allow the clients to store, manage, and share their data portably and easily from anywhere via the Internet. However, because cloud providers could not be untrustworthy, this system introduces three security challenges: data availability, data integrity and data confidentiality. Ensuring data availability and data integrity is the primary requirement before ensuring data confidentiality because data availability and data integrity are the prerequisites of the existence of a system. This paper thus focuses on data availability and data integrity. To allow the client to check whether the data stored in the cloud servers is available and intact, researchers proposed Proof of Retrievability (POR) [1–3], which is the challenge-response protocol between a client and a server. Based on the POR, the following three approaches are commonly used: replication, erasure coding, and network coding. In the replication [4–6], the client stores a file replica in each server. The client can perform periodic server checks. If a server is corrupted, the client will uses a healthy replica to repair the corruption. The drawback of this approach is the high storage cost for the redundant replicas. To address this drawback, the erasure coding was proposed [7–10]. Instead of storing file replicas as in the replication, the client stores file blocks in each server. Hence, the storage cost can be reduced. However, the drawback of this approach is that to repair a corrupted server, the client must reconstruct the original file before generating the new coded blocks. The computation cost is thus increased during data repair. To address this drawback, the network coding was proposed [11–13] in which the client does not need to reconstruct the original file before repairing the corruption. Instead, the client retrieves the coded blocks in the healthy servers to generate the new file blocks. Therefore, this paper focuses on the network coding. Furthermore, the data stored in the servers cannot be checked without an additional information: Message Authentication Code (MAC) (used for a symmetric key setting), or signature (used for an asymmetric key setting). A MAC is also called a *tag*. This paper is based on a symmetric key setting which is well-known to be more efficient than an asymmetric key setting. The MAC is thus used in the proposed scheme.

Network Coding-based POR. Dimakis et al. [14] were the first to apply the network coding to distributed storage systems and achieve a reduction in the communication overhead of the repair

component. Li et al. [15] introduced the tree-structure data regeneration with the linear network coding to achieve an efficient regeneration traffic and bandwidth capacity by using an undirected-weighted maximum spanning tree. Chen et al. then proposed the Remote Data Checking for Network Coding-based distributed storage system (RDC-NC) [16] which provides an elegant data repair by recoding encoded blocks in healthy servers during repair. H.Chen et al. proposed the NC-Cloud scheme [17] to improve cost-effective repair using the functional minimum-storage regenerating (FMSR) codes and lighten the encoding requirement of storage nodes during repair. However, most of these schemes have the following shortcomings. Firstly, the schemes can only support the indirect repair. That is, to repair a corrupted server, the client must require the healthy servers to provide aggregated coded blocks and aggregated tags, and send them back to the client. The client then checks the provided coded blocks using the tags, and computes the new coded blocks and new tags to replace the corruption. The client sends these new coded blocks and tags to the new server. Such a repair mechanism is a troublesome task for the client. Because the data repair is performed very often during the system lifetime, the client thus incurs high computation and communication costs. Secondly, the schemes do not consider the dynamic operations. That is, the client can only perform the data check and data retrieval, but cannot perform the modification, insertion and deletion. A few PORs were proposed to deal with the dynamic operations, e.g, [18–21]. However, all these schemes are based on the erasure coding, not the network coding.

There are two most notable schemes which are related to our proposed scheme. The first one is the MD-POR [22], which can support the direct repair, but cannot support the dynamic operations. The second one is the NC-Audit [23], which also considered the direct repair and dynamic operations. However, when the direct repair is supported, this scheme cannot prevent the pollution attack which is a common attack of the network coding. This is because the new server cannot check the provided coded blocks. Furthermore, the authors only discuss about the dynamic operations without clear details. For example, for the modification, the authors discuss how to update the tag without mentioning how to update the coded blocks which are related to the modified file block. For the insertion, the authors mentioned that the insertion does not work in their scheme. For the deletion, there is no concrete explanation.

Contribution. This paper proposes the DD-POR scheme with the following contributions:

- Direct repair: when a server is corrupted, the healthy servers will provide their coded blocks and tags directly to the new server without sending them back to the client. Then, the new server can check them to prevent the pollution attack, and can compute the new coded blocks and the tags for itself. The client is thus free from the repair process.
- Dynamic operations: the client not only can check and retrieve the data, but also can modify, insert and delete the data.
- Symmetric key setting: our scheme does not use any public key for the efficiency. The direct repair feature introduces a difficulty that how to allow the new server which is untrusted to check and compute the new coded blocks and the tags without using a public key, our scheme can address this problem by using an orthogonal key technique called InterMac [24].

Roadmap. The backgrounds of the POR, network coding and InterMac are described in Section 2. The adversarial model is presented in Section 3. The DD-POR scheme is proposed in Section 4. The security and efficiency analyses are given in Section 5 and 6. The conclusion is drawn in Section 7.

2 Background

2.1 The POR Framework

The POR [1–3] is a challenge-response protocol between a verifier V (client) and a prover P (server), and consists of the functions defined below.

- $\mathsf{KGen}(\lambda) \rightarrow \kappa$: run by V. This function takes a security parameter s as the input and outputs a secret key κ (For a asymmetric key system, κ is a public/private key pair.)
- $\mathsf{Encode}(F, \kappa) \rightarrow F'$: run by V. This function takes an original file F and κ as input, and outputs an encoded file F^*, and then sends F^* to P.
- $\mathsf{Check}() \rightarrow \{\mathsf{accept}/\mathsf{deny}\}$: run by both V and P. Firstly, V generates a challenge c and sends c to P. P then computes a response r and sends r back to V. V finally verifies P based on c and r.

– Repair(): run by V. When a corruption is detected, V executes this function to repair the corruption. The technique of repair depends on each specific scheme, i.e, replication, erasure coding or network coding.

2.2 Network Coding in Distributed Storage System

The network coding [11–13] is proposed for cost-efficiency in data transmission. The model system consists of a client and multiple servers. The client owns a file F and wants to store the redundant coded blocks in the servers in a way that the client can reconstruct F and can repair the coded blocks in a corrupted server. The client firstly divides F into m blocks: $F = v_1||\cdots||v_m \in \mathbb{F}_q^z$. Each $v_k \in \mathbb{F}_q^z$ where $k \in \{1, \cdots, m\}$ and $\mathbb{F}_q^z$ denotes a z-dimensional finite field over a prime q. The client then augments v_k with a vector of length m which contains a single '1' in the position k and $(m-1)$ single '0's elsewhere. The resulting block is called *augmented block* (says, w_k). w_k has the following form:

$$w_k = (v_k, \overbrace{\underbrace{0, \cdots, 0, 1}_{k}, 0, \cdots, 0}^{m}) \in \mathbb{F}_q^{z+m} \tag{1}$$

Thereafter, the client randomly chooses m coefficients $\alpha_1, \cdots, \alpha_m \in \mathbb{F}_q$ to compute coded blocks using the linear combination $c = \sum_{k=1}^{m} \alpha_k \cdot w_k \in \mathbb{F}_q^{z+m}$. The clients stores these coded blocks in the servers. To reconstruct F, any m coded blocks are required to solve m augmented blocks $w_1, \cdots, w_m$ using the accumulated coefficients contained in the last m coordinates of each coded block. After the m augmented blocks are solved, m file blocks $v_1, \cdots, v_m$ are obtained from the first coordinate of each augmented block. Finally, F is reconstructed by concatenating the file blocks. Note that the matrix consisting of the coefficients used to construct any m coded blocks should have full rank. R. Koetter et al. [13] proved that if the prime q is chosen large enough and the coefficients are chosen randomly, the probability for the matrix having full rank is high. When a server is corrupted, the client repairs it by retrieving the coded blocks from the healthy servers and linearly combining them to regenerate the new coded blocks. An example of the data repair is given in Figure 1.

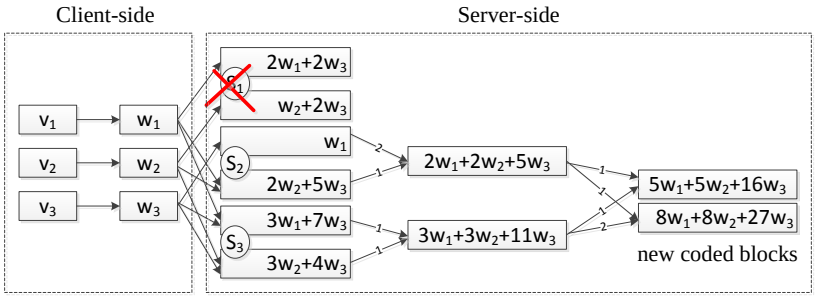

Fig. 1: *The client stores the coded blocks in the server S_1, S_2, S_3. Suppose that S_1 is corrupted, the client repair it by the linear combinations of the coded blocks from S_2 and S_3.*

2.3 InterMac

When we deal with the direct repair, the difficulty is how to allow the new server which is untrusted to check the provided coded blocks using its key without learning the secret key of the client. The InterMac [24] is a suitable technique to generate such a key for the new server. Basically, the InterMac is proposed to generate a vector which is orthogonal to a given set of vectors. Formally, given the set of vectors $\{w_1, \cdots, w_m\}$ and an integer $p \in \mathbb{Z}_q^*, p \neq 1$, the algorithm outputs a vector k_p s.t. $k_p \cdot w_k = 0, \forall k \in \{1, \cdots, m\}$.
(1) The main algorithm InterMac is described as follows:
InterMac$(\{w_1, \cdots, w_m\}, p) \rightarrow k_p$:
– Find the span π of $w_1, \cdots, w_m \in \mathbb{F}_q^{z+m}$.
– Construct the matrix M in which $\{w_1, \cdots, w_m\}$ are the rows of M.
– Find the null-space of M, denoted by $\pi_M^{\perp}$, which is the set of all vectors $u \in \mathbb{F}_q^{z+m}$ s.t. $M \cdot u^T = 0$.
– Find the basis vectors of $\pi_M^{\perp}$, denoted by $b_1, \cdots, b_z \in \mathbb{F}_q^{z+m}$ // Theorem 1 will explain why the number of the basis vectors is z.

– Let $B = \{b_1, \cdots, b_z\}$
– Compute $k_p \leftarrow \mathsf{Kg}(B)$.

(2) The sub-algorithm Kg used in $\mathsf{InterMac}$ is given as follows:
$\mathsf{Kg}(B = \{b_1, \cdots, b_z\}) \rightarrow k_p$:
– Let f be a Pseudorandom function s.t. $\mathcal{K} \times \mathcal{P} \times [1, z] \rightarrow \mathbb{F}_q$.
– Generate $r_i \leftarrow f(k, p, i) \in \mathbb{F}_q, \forall i \in \{1, \cdots, z\}$ where $k \in \mathcal{K}, p \in \mathcal{P}$.
– Compute $k_p \leftarrow \sum_{i=1}^{z} r_i \cdot b_i \in \mathbb{F}_q^{z+m}$.

Theorem 1. *Given $\{w_1, \cdots, w_m\} \in \mathbb{F}_q^{z+m}$, the number of basis vectors of $\pi_M^{\perp}$ is z.*

Proof. $\mathsf{rank}(M) = m$. Let π_M be the space spanned by the rows of M. For any $m \times (z + m)$ matrix, the rank-nullity theorem gives: $\mathsf{rank}(M) + \mathsf{nullity}(M) = z + m$ where $\mathsf{nullity}(M)$ is the dimension of $\pi_M^{\perp}$. Therefore, $\mathsf{dim}(\pi_M^{\perp}) = (z + m) - m = z$. In other words, the number of basis vectors of $\pi_M^{\perp}$ is z. In the $\mathsf{InterMac}$, we denote the basis vectors by $\{b_1, \cdots, b_z\}$. □

3 Adversarial Model

In our scheme, the client is trusted, and the servers are untrusted. Assume that the servers do not collude with each other. The servers can perform two types of attacks:
(1) In the check phase: the servers disrupts the system or modifies the data. These attacks can be commonly prevented by the tags, we thus do not focus on these attacks.
(2) In the repair phase: the servers can perform: (i) the pollution attack which is a common attack of the network coding, and (ii) the curious attack which is a special attack of the direct repair. We focus on these in the security analysis.

– *Pollution Attack.* A malicious server firstly uses a valid coded block to pass the check phase, but then injects an invalid coded block in the repair phase to prevent data repair. An example is given as follows:
  • Encode: the client encodes the augmented blocks (w_1, w_2, w_3) to six coded blocks: c_{11}, c_{12} (stored in the server S_1), c_{21}, c_{22} (stored in the server S_2), and c_{31}, c_{32} (stored in the server S_3). Suppose that S_1 will perform a pollution attack.
  • Check: S_3 is corrupted.
  • Repair: S_3 should be repaired by two coded blocks: c'_{31} (which is a linear combination of c_{11} and c_{12}) and c'_{32} (which is a linear combination of c_{21} and c_{22}). However, S_1 is not detected because this time is the repair phase, not the check phase. The client still thinks S_1 is healthy, and thus the client requests the coded blocks from S_1 and S_2. S_1 will provide an invalid coded block c''_{31} to the client instead of c'_{31}.
– *Curious Attack.* This attack is performed by the new server in the repair phase. Every repair time, the new server is given a key k_r constructed from the secret key k_C of the client and a variant k_p. Having k_r, the new server tries to obtain k_C in order to pass the check phases in the later epochs.

4 Our Proposed Scheme

4.1 Notations

Throughout the DD-POR scheme, the following notations and definitions are used:
– $\mathcal{C}$ denotes the client.
– F denotes the original file.
– m denotes the number of file blocks.
– n denotes the number of servers.
– d denotes the number of coded blocks in each server.
– k denotes the file block index ($k \in \{1, \cdots, m\}$).
– i denotes the server index ($i \in \{1, \cdots, n\}$).
– j denotes the coded block index in each server ($j \in \{1, \cdots, d\}$).
– v_k denotes the file block ($k \in \{1, \cdots, m\}$).
– w_k denotes the augmented block of v_k.

– t_{w_k} denote the tag of w_k.
– S_i denotes the server.
– c_{ij} denotes the j-th coded block stored in S_i.
– $t_{c_{ij}}$ denotes the tag of c_{ij}.
– l denotes the number of healthy servers used for data repair.
– S_r denotes the corrupted server.
– S'_r denotes the new server which is used to replace S_r.
– $\mathbb{F}_q^z$ is the z-dimensional finite field $\mathbb{F}$ over a prime q.

4.2 Construction

Setup.
(1) *Create augmented blocks*: $\mathcal{C}$ divides F into m blocks $F = v_1||\cdots||v_m$. Each $v_k \in \mathbb{F}_q^z$ where $k \in \{1, \cdots, m\}$. $\mathcal{C}$ creates m augmented blocks as Eq. 1.
(2) *Keygen*: $\mathcal{C}$ generates two types of keys:
– *The key of the client (k_C)*: $k_C \xleftarrow{rand} \mathbb{F}_q^{z+m}$.
– *The one-time key of the new server every repair time (k_r)*: $\mathcal{C}$ generates $p \xleftarrow{rand} \mathbb{Z}_q^*, p \neq 1$ (p is generated every repair time). $\mathcal{C}$ computes a key $k_p \leftarrow \mathsf{InterMac}(\{w_1, \cdots, w_m\}, p)$. The property of k_p is that $k_p \cdot w_k = 0$ where $k \in \{1, \cdots, m\}$. Let $k_r = k_C + k_p \in \mathbb{F}_q^{z+m}$. k_C is static, only k_p is re-computed every repair time. k_r is sent to the new server only when the repair phase is executed.

Encode.
(1) $\mathcal{C}$ computes a tag for each augmented block as follows:
– For $\forall k \in \{1, \cdots, m\}$, $\mathcal{C}$ computes tag: $t_{w_k} = w_k \cdot k_C \in \mathbb{F}_q$.
(2) $\mathcal{C}$ computes nd coded blocks and nd corresponding tags as follows:
For $\forall i \in \{1, \cdots, n\}, \forall j \in \{1, \cdots, d\}$:
– $\mathcal{C}$ generates m coefficients: $\alpha_{ijk} \xleftarrow{rand} \mathbb{F}_q$ where $k \in \{1, \cdots, m\}$.
– $\mathcal{C}$ computes code block: $c_{ij} = \sum_{k=1}^{m} \alpha_{ijk} \cdot w_k \in \mathbb{F}_q^{z+m}$.
– $\mathcal{C}$ compute tag: $t_{c_{ij}} = \sum_{k=1}^{m} \alpha_{ijk} \cdot t_{w_k} \in \mathbb{F}_q$.
(3) $\mathcal{C}$ sends d pairs of $\{c_{ij}, t_{c_{ij}}\}$ where $j \in \{1, \cdots, d\}$ to the server S_i.

Check.
(1) $\mathcal{C}$ requires S_i to provide its aggregated coded block and aggregated tag.
(2) S_i ($i \in \{1, \cdots, n\}$) combines its coded blocks and tags as follows:
– S_i generates d coefficients: $\beta_{ij} \xleftarrow{rand} \mathbb{F}_q$ where $\forall j \in \{1, \cdots, d\}$.
– S_i combines coded blocks: $c_{S_i} = \sum_{j=1}^{d} \beta_{ij} \cdot c_{ij} \in \mathbb{F}_q^{z+m}$.
– S_i combines tags: $t_{S_i} = \sum_{j=1}^{d} \beta_{ij} \cdot t_{c_{ij}} \in \mathbb{F}_q$.
– S_i sends $\{c_{S_i}, t_{S_i}\}$ to $\mathcal{C}$.
(3) $\mathcal{C}$ verifies S_i as follows:
For $\forall i \in \{1, \cdots, n\}$:
– $\mathcal{C}$ computes $t'_{S_i} = c_{S_i} \cdot k_C \in \mathbb{F}_q$.
– $\mathcal{C}$ checks the following equation. If it holds, $\mathcal{S}_i$ is healthy. Otherwise, $\mathcal{S}_i$ is corrupted.

$$t_{S_i} = t'_{S_i} \tag{2}$$

Repair. Suppose S_r is corrupted. A set of l healthy servers $\{S_{i_1}, \cdots, S_{i_l}\}$ are required to provide data to a new server S'_r, which is used to replace S_r.
(1)S_i ($i \in \{i_1, \cdots, i_l\}$) provides its data to S'_r as follows:
– S_i generates d coefficients: $\beta_{ij} \xleftarrow{rand} \mathbb{F}_q$ where $j \in \{1, \cdots, d\}$.
– S_i combines coded blocks: $c_{S_i} = \sum_{j=1}^{d} \beta_{ij} \cdot c_{ij} \in \mathbb{F}_q^{z+m}$.
– S_i combines tags: $t_{S_i} = \sum_{j=1}^{d} t_{ij} \cdot \beta_{ij} \in \mathbb{F}_q$.
– S_i sends $\{c_{S_i}, t_{S_i}\}$ to S'_r.
(2) S'_r checks $S_i (i \in \{i_1, \cdots, i_l\})$ as follows:
– S'_r computes $t'_{S_i} = c_{S_i} \cdot k_r \in \mathbb{F}_q$.
– S'_r checks if the following equation holds:

$$t'_{S_i} = t_{S_i} \tag{3}$$

(3) S'_r computes d new coded blocks and d corresponding tags:
For $\forall j \in \{1, \cdots, d\}$:

- S'_r generates l coefficients: $\gamma_{ri} \overset{rand}{\leftarrow} \mathbb{F}_q$ where $i \in \{i_1, \cdots, i_l\}$.
- S'_r computes new coded block: $c_{rj} = \sum_{i=i_1}^{i_l} \gamma_{ri} \cdot c_{S_i} \in \mathbb{F}_q^{z+m}$.
- S'_r computes new tag: $t_{rj} = \sum_{i=i_1}^{i_l} \gamma_{ri} \cdot t_{S_i} \in \mathbb{F}_q$.

4.3 Correctness

(1) We firstly prove the correctness of Eq. 2:
$t_{S_i} = \sum_{j=1}^{d} \beta_{ij} \cdot t_{c_{ij}} = \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot t_{w_k} = \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot w_k \cdot k_C$.
$t'_{S_i} = c_{S_i} \cdot k_C = \sum_{j=1}^{d} \beta_{ij} \cdot c_{ij} \cdot k_C = \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot w_k \cdot k_C = t_{S_i}$.
(2) We prove the correctness of Eq. 3
$t_{S_i} = \sum_{j=1}^{d} \beta_{ij} \cdot t_{c_{ij}} = \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot t_{w_k} = \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot w_k \cdot k_C$.
$t'_{S_i} = c_{S_i} \cdot k_r = \sum_{j=1}^{d} \beta_{ij} \cdot c_{ij} \cdot k_r = \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot w_k \cdot (k_C + k_p)$
$= \sum_{j=1}^{d} \sum_{k=1}^{m} \beta_{ij} \cdot \alpha_{ijk} \cdot w_k \cdot k_C \ // \ k_p \cdot w_k = 0$
$= t_{S_i}$

4.4 Dynamic Operations

When $\mathcal{C}$ performs a dynamic operation on a file block, herein introduces a difficulty of the task: how the servers deal with the coded blocks which are related to the modified/inserted/deleted file block. The trivial solution is to encode data again. This solution incurs very high costs. In our solution, the old coded blocks and tags stored in the servers can be re-used, and only a small additional computation is needed for the dynamic operations.

Firstly, we give the following theorem, which will form the basis of the dynamic operations.

Theorem 2. *The basis vector of the matrix consisting of m augmented blocks is unique.*

Proof. Let M be the matrix in which each of m augmented blocks is a row in M:

$$M = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} = \begin{pmatrix} v_1, 1, 0, 0, \cdots, 0 \\ v_2, 0, 1, 0, \cdots, 0 \\ \vdots \\ v_m, 0, \cdots, 0, 1 \end{pmatrix}$$

Because the dimension of M is $m \times (m+1)$, the number of pivot variables is m and the number free variables is $(m+1) - m = 1$. Thus, the number of basis vectors of M is 1. □

Modification. Suppose that $\mathcal{C}$ modifies a file block v_X to a new file block v'_X. Let w_X and w'_X be the augmented block of v_X and v'_X, respectively.
(1) $\mathcal{C}$ *re-computes k_r for the next repair time*:

Let M be the matrix consisting of m augmented blocks. After the modification, only v_X is changed and other elements in M are not changed. Namely, M is changed to M' as follows:

$$M = \begin{pmatrix} v_1, 1, 0, 0, \cdots, 0 \\ v_2, 0, 1, 0, \cdots, 0 \\ \vdots \\ \boxed{v_X}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \vdots \\ v_m, 0, \cdots, 0, 1 \end{pmatrix} \overset{modification}{\rightarrow} M' = \begin{pmatrix} v_1, 1, 0, 0, \cdots, 0 \\ v_2, 0, 1, 0, \cdots, 0 \\ \vdots \\ \boxed{v'_X}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \vdots \\ v_m, 0, \cdots, 0, 1 \end{pmatrix}$$

Observer that the modification does not affect k_C, but affect $k_r (= k_C + k_p)$ because k_p is constructed from M. This is why we need to update k_r.

– Because the number of columns in M is $(m+1)$, and because the number of basis vectors of M is 1 (Theorem 2), we have that the unique basis vector of M consists of $(m+1)$ elements (says, $B = [b_1, \cdots, b_{m+1}]$). Similarly, the unique basis vector of M' also consists of $(m+1)$ elements (says, $B' = [b'_1, \cdots, b'_{m+1}]$). We need to find B' from B.
– Because only v_X in M is changed and other elements are not changed, $\mathcal{C}$ only needs to re-compute the $(X+1)$-th element of B by $(-v'_X \cdot b_1) \mod q$. Namely,

$$B' = [b_1, \cdots, b_X, \boxed{(-v'_X \cdot b_1) \mod q}, b_{X+2}, \cdots, b_{m+1}] \tag{4}$$

– $\mathcal{C}$ then computes $k'_p \leftarrow \mathsf{Kg}(B')$ (described in Section 2.3). $\mathcal{C}$ finally sends $k'_r = k_C + k'_p$ to the new server when the next repair phase is executed.

(2) *$\mathcal{C}$ computes the tag for w'_k. Each server updates its coded blocks and tags*:
– $\mathcal{C}$ computes the tag: $t'_X = w'_X \cdot k_C \in \mathbb{F}_q$, and sends $\{w'_X, t'_X\}$ to each S_i.
– S_i updates its coded blocks and tags as follows:
For $\forall j \in \{1, \cdots, d\}$: $\begin{cases} c'_{ij} = c_{ij} - \alpha_{ijX} \cdot w_X + \alpha_{ijX} \cdot w'_X = c_{ij} + \alpha_{ijX} \cdot \Delta w \\ t'_{ij} = t_{ij} - \alpha_{ijX} \cdot t_X + \alpha_{ijX} \cdot t'_X = t_{ij} + \alpha_{ijX} \cdot \Delta t \end{cases}$
where c_{ij} and t_{ij} are the old coded block and tag. $\Delta w = w'_X - w_X$, $\Delta t = t'_X - t_X$. The coefficient α_{ijX} can be found in the $(X+1)$-th element of c_{ij}.

Insertion. Suppose that $\mathcal{C}$ inserts a file block v_I after the existing file block v_X. Let w_I be the augmented block of v_I.
(1) *$\mathcal{C}$ modifies k_C*:

Before the insertion, because an augmented block has $(m+1)$ elements $(w_k = (v_k, \underbrace{0, \cdots, 0, 1}_{k}, \overbrace{0, \cdots, 0}^{m}))$, thus k_C also has $(m+1)$ elements (says, $k_C = [k_1, \cdots, k_{m+1}]$). After the insertion, because an augmented block has $(m+2)$ elements $(w'_k = (v_k, \underbrace{0, \cdots, 0, 1}_{k}, \overbrace{0, \cdots, 0}^{m+1}))$, and thus the new k'_C also has $(m+2)$ elements (says, $k'_C = [k'_1, \cdots, k'_{m+2}]$). Given k_C, we find k'_C as follows:

$$k'_C = [k_1, \cdots, k_{X+1}, \boxed{k_I}, k_{X+2}, \cdots, k_{m+1}] \tag{5}$$

– The first $(X+1)$ elements of k'_C are the same as these of k_C.
– The $(X+2)$-th element of k'_C (denoted by k_I) is computed as: $k_I \xleftarrow{rand} \mathbb{F}_q$.
– The last $(m-X)$ elements of k'_C are the same as the last $(m-X)$ elements of k_C.

The reason that we construct such k'_C will be explained in Step 3 (tag update).
(2) *$\mathcal{C}$ re-computes k_r for the next repair time*:
After the insertion, the matrix M is changed as follows:
– In each of the first X rows, a single '0' is padded in the final position.
– In the inserted row (w_I), v_I is the first element, a '1' bit is the $(X+1)$-th element counted from the second element and '0' elsewhere.
– In each of the last $(m-X)$ rows, a single '0' is padded in the final position and the single '1' is shipped to the next right position.

$$M = \underbrace{\begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ \boxed{v_X}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \boxed{v_{X+1}}, \underbrace{0, \cdots, 0, 1}_{X+1}, 0, \cdots, 0 \\ \vdots \\ v_m, 0, \cdots, 0, 1 \end{pmatrix}}_{m \times (m+1)} \overset{insertion}{\rightarrow} M' = \underbrace{\begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ v_X, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \boxed{v_I}, \underbrace{0, \cdots, 0, 1}_{X+1}, 0, \cdots, 0 \\ v_{X+1}, \underbrace{0, \cdots, 0, 1}_{X+2}, 0, \cdots, 0 \\ \vdots \\ v_m, \underbrace{0, \cdots, 0, 1}_{m+1} \end{pmatrix}}_{(m+1) \times (m+2)}$$

We now update k'_r as follows:

– Let $B = [b_1, \cdots, b_{m+1}]$ and $B' = [b'_1, \cdots, b'_{m+2}]$ be the basis vector of M and M', respectively. Given B, we firstly find B':
  - The first $(X+1)$ elements of B' are the same as these of B:
$$(b'_1 = b_1), \cdots, (b'_{X+1} = b_{X+1})$$
  - The last $(m-X+1)$ elements of B' are simply computed as:
$$\begin{cases} b'_{X+2} = (-v_I \cdot b_1) \mod q \\ b'_t = b_{t-1} \quad where \quad t \in \{X+3, \cdots, m+2\} \end{cases}$$
  In other words:
$$B' = [b_1, \cdots, b_{X+1}, \boxed{(-v_I \cdot b_1) \mod q}, b_{X+2}, \cdots, b_{m+1}] \tag{6}$$
– $\mathcal{C}$ then computes $k'_p \leftarrow \mathsf{Kg}(B')$ (described in Section 2.3). $\mathcal{C}$ finally sends $k'_r = k'_C + k'_p$ to the new server when the next repair phase is executed.

(3) *$\mathcal{C}$ computes a tag for w_I. Each server S_i updates its coded blocks and tags*:

– $\mathcal{C}$ computes a tag for w_I as: $t_I = w_I \cdot k'_C$, then sends $\{w_I, t_I\}$ to S_i.
– S_i updates its coded blocks as follows:
An old coded block has $(m+1)$ elements: $c_{ij} = (\sum_{k=1}^{m} \alpha_{ijk} \cdot w_k, \alpha_{ij1}, \cdots, \alpha_{ijm})$. Let $c_{ij}[x]$ denote the x-th element of c_{ij} ($x \in \{1, \cdots, m+1\}$). We compute the new coded block as:
$$\begin{aligned} c'_{ij} &= (\textstyle\sum_{k=1}^{m} \alpha_{ijk} w_k + \alpha_{ijI} w_I, \alpha_{ij1}, \cdots, \alpha_{ijX}, \boxed{\alpha_{ijI}}, \alpha_{ij(X+1)}, \cdots, \alpha_{ijm}) \\ &= (c_{ij}[1] + \alpha_{ijI} w_I, c_{ij}[2], \cdots, c_{ij}[X+1], \boxed{\alpha_{ijI}}, c_{ij}[X+2], \cdots, c_{ij}[m+1]) \end{aligned} \tag{7}$$
where $\alpha_{ijI} \leftarrow \mathbb{F}_q$.
– S_i updates its tags as follows:
By constructing k'_C as Step 1, the tags of the augmented blocks before the insertion are:
$$\begin{pmatrix} t_{w_1} \\ \vdots \\ t_{w_X} \\ t_{w_{(X+1)}} \\ \vdots \\ t_{w_m} \end{pmatrix} = M \cdot k_C = \begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ \boxed{v_X}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \boxed{v_{X+1}}, \underbrace{0, \cdots, 0, 1}_{X+1}, 0, \cdots, 0 \\ \vdots \\ v_m, 0, \cdots, 0, 1 \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_{(X+1)} \\ k_{(X+2)} \\ \vdots \\ k_{m+1} \end{pmatrix} = \begin{pmatrix} v_1 k_1 + k_2 \\ \vdots \\ v_X k_1 + k_{(X+1)} \\ v_{(X+1)} k_1 + k_{(X+2)} \\ \vdots \\ v_m k_1 + k_{m+1} \end{pmatrix}$$
The tags of the augmented blocks after the insertion are:
$$\begin{pmatrix} t'_{w_1} \\ \vdots \\ t'_{w_X} \\ \boxed{t_I} \\ t'_{w_{(X+1)}} \\ \vdots \\ t'_{w_{m+1}} \end{pmatrix} = M' \cdot k'_C = \begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ v_X, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \boxed{v_I}, \underbrace{0, \cdots, 0, 1}_{X+1}, 0, \cdots, 0 \\ v_{X+1}, \underbrace{0, \cdots, 0, 1}_{X+2}, 0, \cdots, 0 \\ \vdots \\ v_m, \underbrace{0, \cdots, 0, 1}_{m+1} \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_{(X+1)} \\ \boxed{k_I} \\ k_{(X+2)} \\ \vdots \\ k_{m+1} \end{pmatrix} = \begin{pmatrix} v_1 k_1 + k_2 \\ \vdots \\ v_X k_1 + k_{(X+1)} \\ \boxed{v_I k_1 + k_I} \\ v_{(X+1)} k_1 + k_{(X+2)} \\ \vdots \\ v_m k_1 + k_{m+1} \end{pmatrix}$$
Observe that before and after the insertion, the first $(X+1)$ tags and the last $(m-X)$ tags are not changed; only a new tag t_I, which is the tag of w_I, is inserted. Furthermore, the old tag of c_{ij} is computed as $t_{ij} = \sum_{k=1}^{m} \alpha_{ijk} \cdot t_{w_k}$. We are now ready to compute the tag for c'_{ij} as follows:
$$t_{c'_{ij}} = \sum_{k=1}^{X} \alpha_{ijk} t_{w_k} + \boxed{\alpha_{ijI}} t_I + \sum_{k=X+1}^{m} \alpha_{ijk} t_{w_k} = t_{c_{ij}} + \boxed{\alpha_{ijI} t_I} \tag{8}$$
where α_{ijI} is the same as in Eq. 7.

Deletion. Suppose that $\mathcal{C}$ deletes the X-th file block (v_X). Let w_X be the augmented block of v_X.

(1) *$\mathcal{C}$ modifies k_C*:

Similar to the insertion operation, before the deletion, the key of $\mathcal{C}$ has the form: $k_C = [k_1, \cdots, k_{m+1}]$. After the deletion, $\mathcal{C}$ simply removes the $(X+1)$-th element in k_C. Namely,

$$k'_C = [k_1, \cdots, k_X, k_{X+2}, \cdots, k_{m+1}] \tag{9}$$

The reason to construct such k'_C will be explained in Step 3 (tag update).

(2) *$\mathcal{C}$ re-computes k_r for the next repair time*:

After the deletion, the matrix M is now changed as follows:

- In each of the first $(X-1)$ rows, the single '0' in the final position is removed.
- The X-th row is removed.
- In each of the last $(m-X)$ rows, the single '1' is shipped to the previous left position and the single '0' in the final position is removed.

$$M = \underbrace{\begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ v_{X-1}, \underbrace{0, \cdots, 0, 1}_{X-1}, 0, \cdots, 0 \\ \boxed{v_X}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ v_{X+1}, \underbrace{0, \cdots, 0, 1}_{X+1}, 0, \cdots, 0 \\ \vdots \\ v_m, \underbrace{0, \cdots, 0, 1}_{m} \end{pmatrix}}_{m \times (m+1)} \overset{deletion}{\rightarrow} M' = \underbrace{\begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ v_{X-1}, \underbrace{0, \cdots, 0, 1}_{X-1}, 0, \cdots, 0 \\ v_{X+1}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \vdots \\ v_m, \underbrace{0, \cdots, 0, 1}_{m-1} \end{pmatrix}}_{(m-1) \times m}$$

We now update k'_r as follows:

- Let $B = [b_1, \cdots, b_{m+1}]$ be the basis vector of M.
- To compute the basis vector B' of M', $\mathcal{C}$ simply removes the $(X+1)$-th element of B. Namely,

$$B' = [b_1, \cdots, b_X, b_{X+2}, \cdots, b_{m+1}] \tag{10}$$

- $\mathcal{C}$ then computes $k'_p \leftarrow \mathsf{Kg}(B')$ (Section 2.3). $\mathcal{C}$ finally sends $k'_r = k'_C + k'_p$ to the new server when the next repair phase is executed.

(3) *S_i updates its coded blocks and tags*:

- S_i updates its coded blocks:

  Because an old coded block has $(m+1)$ elements: $c_{ij} = (\sum_{k=1}^{m} \alpha_{ijk} \cdot w_k, \alpha_{ij1}, \cdots, \alpha_{ijm})$ and let $c_{ij}[x]$ denote the x-th element of c_{ij} where $x \in \{1, \cdots, m+1\}$, we compute the new coded block as follows:

$$\begin{aligned} c'_{ij} &= (\textstyle\sum_{k=1}^{X-1} \alpha_{ijk} w_k + \sum_{k=X+1}^{m} \alpha_{ijk} w_k, \alpha_{ij1}, \cdots, \alpha_{ij(X-1)}, \alpha_{ij(X+1)}, \cdots, \alpha_{ijm}) \\ &= (c_{ij}[1] - \alpha_{ijX} w_X, c_{ij}[2], \cdots, c_{ij}[X], c_{ij}[X+2], \cdots, c_{ij}[m+1]) \end{aligned} \tag{11}$$

  where $\alpha_{ijX} = c_{ij}[X+1]$.
- S_i updates its tags as follows:

  By constructing k'_C as Step 1, the tags of the augmented blocks before the deletion are:

$$\begin{pmatrix} t_{w_1} \\ \vdots \\ t_{w_{X-1}} \\ t_{w_X} \\ t_{w_{(X+1)}} \\ \vdots \\ t_{w_m} \end{pmatrix} = M \cdot k_C = \begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ v_{X-1}, \underbrace{0, \cdots, 0, 1}_{X-1}, 0, \cdots, 0 \\ \boxed{v_X}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ v_{X+1}, \underbrace{0, \cdots, 0, 1}_{X+1}, 0, \cdots, 0 \\ \vdots \\ v_m, 0, \cdots, 0, 1 \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_{(X+1)} \\ \vdots \\ k_{m+1} \end{pmatrix} = \begin{pmatrix} v_1 k_1 + k_2 \\ \vdots \\ v_{X-1} k_1 + k_X \\ v_X k_1 + k_{(X+1)} \\ v_{X+1} k_1 + k_{(X+2)} \\ \vdots \\ v_m k_1 + k_{m+1} \end{pmatrix}$$

  The tags of all augmented blocks after the insertion are:

$$\begin{pmatrix} t'_{w_1} \\ \vdots \\ t'_{w_{X-1}} \\ t'_{w_{(X+1)}} \\ \vdots \\ t'_{w_{m+1}} \end{pmatrix} = M' \cdot k'_C = \begin{pmatrix} v_1, 1, 0, \cdots, 0 \\ \vdots \\ v_{X-1}, \underbrace{0, \cdots, 0, 1}_{X-1}, 0, \cdots, 0 \\ v_{X+1}, \underbrace{0, \cdots, 0, 1}_{X}, 0, \cdots, 0 \\ \vdots \\ v_m, \underbrace{0, \cdots, 0, 1}_{m-1} \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_X \\ k_{(X+2)} \\ \vdots \\ k_{m+1} \end{pmatrix} = \begin{pmatrix} v_1 k_1 + k_2 \\ \vdots \\ v_{X-1} k_1 + k_X \\ v_{(X+1)} k_1 + k_{(X+2)} \\ \vdots \\ v_m k_1 + k_{m+1} \end{pmatrix}$$

Observe that before and after the deletion, the X-th tag is removed and the other tags are still the same. Furthermore, the old tag of c_{ij} is computed as: $t_{ij} = \sum_{k=1}^{m} \alpha_{ijk} \cdot t_{w_k}$. We are now ready to compute the tag for c'_{ij} as follows:

$$t_{c'_{ij}} = \sum_{k=1}^{X-1} \alpha_{ijk} t_{w_k} + \sum_{k=X+1}^{m} \alpha_{ijk} t_{w_k} = t_{c_{ij}} - \boxed{\alpha_{ijX} t_X} \tag{12}$$

where α_{ijX} is the same as in Eq. 11.

5 Security Analysis

We firstly show our scheme is secure from the pollution attack and curious attack as follows.

Theorem 3. *The DD-POR is secured from the pollution and curious attacks.*

Proof. (1) Pollution attack: suppose that $\mathcal{A}$ is a malicious server in a set of l servers which are used to repair the corrupted server. $\mathcal{A}$ injects an invalid pair of $(c_{\mathcal{A}}, t_{\mathcal{A}})$ to the new server S'_r. Then S'_r will check $(c_{\mathcal{A}}, t_{\mathcal{A}})$ using the key $k_r \in \mathbb{F}_q^{z+m}$. Because S'_r is assumed to not collude with the other servers, $k_r \in \mathbb{F}_q^{z+m}$ is only known by S'_r. Thus, $\mathcal{A}$ can only pass the verification of S'_r with a probability $1/q^{z+m}$ via the brute-force search. If q is chosen large enough (e.g, 160 bits), the probability is $1/(2^{160})^{z+m}$, which is negligible.

Consider that the S'_r itself is a malicious server who will perform the pollution attack in the next epoch. Even though S'_r holds k_r, S'_r cannot pass the verification in the repair phase because k_r is a one-time repair key. Another new server will be given a key $k'_r \neq k_r$.
(2) Curious attack: the new server is given the key $k_r = k_C + k_p \in \mathbb{F}_q^{z+m}$. Similar to the pollution attack, the probability of the new server to learn k_C is $1/q^{z+m}$ via the brute-force search. This probability is from learning k_C directly or learning k_p and then obtaining k_C by $k_C = k_r - k_p$. If q is chosen large enough (e.g, 160 bits), the probability is $1/(2^{160})^{z+m}$, which is negligible. □

We also describe the condition to reconstruct F via the following theorem.

Theorem 4. *F can be reconstructed if in any epoch, at least l out of n servers collectively store m coded blocks which are linearly independent combinations of m augmented blocks, and the matrix consisting of the accumulated coefficients has full rank (i.e, the rank is m).*

Proof. S_i contains d coded blocks: $\{c_{ij}\}(j \in \{1, \cdots, d\})$. c_{ij} is computed from m augmented blocks $w_1, \cdots, w_m$ by $c_{ij} = \sum_{k=1}^{m} \alpha_{ijk} \cdot w_k \in \mathbb{F}_q^{z+m}$. Therefore, to reconstruct F, m augmented blocks are viewed as the unknowns that need to be solved. To solve these unknowns, at least m coded blocks are required s.t the coefficient matrix has full rank.

$$\begin{cases} c_{(ij)_1} = \sum_{k=1}^{m} \alpha_{(ijk)_1} \cdot w_k \\ c_{(ij)_2} = \sum_{k=1}^{m} \alpha_{(ijk)_2} \cdot w_k \\ \cdots \\ c_{(ij)_m} = \sum_{k=1}^{m} \alpha_{(ijk)_m} \cdot w_k \end{cases}$$

Let l be the number of servers ($l < n$) which collectively stores these m coded blocks. Because each server stores d coded blocks, $l = \lceil \frac{m}{d} \rceil$. □

6 Efficiency Analysis

The efficiency comparison is given in Table 1. Because the MD-POR and NC-Audit schemes focus on the public authentication, the system models have one more entity called TPA (Third Party Auditor) who is delegated the task of checking the servers by $\mathcal{C}$. For the fair comparison, we assume that the check task in these schemes is performed by $\mathcal{C}$.

Table 1: The comparison

		RDC-NC [16]	**MD-POR** [22]	**NC-Audit** [23]	**DD-POR**
Feature	Direct repair	No	Yes	Not completed (*)	Yes
	Dynamic operations	No	No	No	Yes
	Symmetric key	Yes	Yes	Yes	Yes
Encode Computation	Client-side	$O(mnd)$	$O(mnd)$	$O(mnd)$	$O(mnd)$
	Server-side	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Check Computation	Client-side	$O(n)$	$O(n)$	$O(n)$	$O(n)$
	Server-side	$O(dn)$	$O(dn)$	$O(dn)$	$O(dn)$
Repair Computation	Client-side	$O(dl)$	$O(1)$	$O(1)$	$O(1)$
	Server-side	$O(dl)$	$O(dl)$	$O(dl)$	$O(dl)$
Modification Computation	Client-side	N/A	N/A	N/A	$O(1)$
	Server-side	N/A	N/A	N/A	$O(dn)$
Insertion Computation	Client-side	N/A	N/A	N/A	$O(1)$
	Server-side	N/A	N/A	N/A	$O(dn)$
Deletion Computation	Client-side	N/A	N/A	N/A	$O(1)$
	Server-side	N/A	N/A	N/A	$O(dn)$

N/A means not applicable due to the lack of support. (*) In the NC-Audit, the direct repair can lead to the pollution attack because the new server cannot check the provided coded blocks.

Encode Computation. In all the schemes, $\mathcal{C}$ needs $O(m)$ to compute m tags for m augmented blocks, and $O(mnd)$ to compute nd coded blocks along with the tags. The complexity on the client-side is thus $O(mnd)$. Meanwhile, the servers only need to receive the coded blocks and tags from $\mathcal{C}$ without any computation. The complexity on the server-side is thus $O(1)$.

Check Computation. In all the schemes, $\mathcal{C}$ needs $O(1)$ to verify the aggregated coded block and tag of each server. Therefore, the complexity on the client-side is $O(n)$ to verify n servers. Meanwhile, each server needs to combine d coded blocks and d tags to compute the aggregated coded block and aggregated tag, respectively. Therefore, the complexity of n servers is $O(dn)$.

Repair Computation. In the RDC-NC scheme, $\mathcal{C}$ needs $O(l)$ to check l pairs of the provided coded block and tag from the healthy servers, and needs $O(dl)$ to compute d pairs of new coded blocks and tags using the linear combinations of l pairs of the provided coded blocks and tags. Therefore, the complexity on the client-side is $O(dl)$. In the MD-POR, NC-Audit and DD-POR schemes, the complexity on the client-side is $O(1)$ because $\mathcal{C}$ does not need to do anything due to the direct repair.

In the RDC-NC scheme, each of l servers combines its d coded blocks and d tags to compute the aggregated coded block and aggregated tag, respectively. Therefore, the complexity on the server-side is $O(dl)$. In the MD-POR, NC-Audit and DD-POR schemes, l healthy servers perform as in the RDC-NC ($O(dl)$), and the new server performs the task of $\mathcal{C}$ as in the RDC-NC ($O(dl)$). Therefore, the complexity on the server-side is $O(dl)$.

Modification Computation. In the DD-POR, $\mathcal{C}$ only needs $O(1)$ to re-compute k_r (Step 1), and $O(1)$ to compute the new tag of the modified augmented block (Step 2). Therefore, the complexity on the client-side is $O(1)$. Meanwhile, each server needs $O(d)$ to update the coded blocks and tags (Step 2). Therefore, the complexity of n servers is $O(dn)$.

Insertion Computation. In the DD-POR, $\mathcal{C}$ only needs $O(1)$ to modify k_C (Step 1), $O(1)$ to re-compute k_r (Step 2), and $O(1)$ to compute the tag of the inserted augmented block (Step 3).

Therefore, the complexity on the client-side is $O(1)$. Meanwhile each server needs $O(d)$ to update the coded blocks and tags (Step 3). Therefore, the complexity of n servers is $O(dn)$.

Deletion Computation. In the DD-POR, $\mathcal{C}$ only needs $O(1)$ to modify k_C (Step 1), and $O(1)$ to re-compute k_r (Step 2). Thus, the complexity on the client-side is $O(1)$. Meanwhile each server needs $O(d)$ to update the coded blocks and tags (Step 3). Thus, the complexity of n servers is $O(dn)$.

7 Conclusion

This paper proposes a network coding-based POR scheme, name DD-POR, to support the direct repair and the dynamic operations in a symmetric key setting. The idea is based on the InterMac technique which can generate a key such s.t the key is orthogonal to the augmented blocks. The security analysis is showed to prevent the pollution attack and curious attack. The efficiency analysis is given based on complexity theory to compare with the previous scheme.

References

1. Juels A, Kaliski B (2007) PORs: Proofs of retrievability for large files. 14th ACM Computer and communications security Conf. - CCS, pp.584-597.
2. Shacham H, Waters B (2008) Compact Proofs of Retrievability. 14th Int. Conf. on the Theory and Application of Cryptology and Information Security - ASIACRYPT, pp.90-107.
3. Bowers K, Juels A, Oprea A (2009) Proofs of retrievability: theory and implementation. ACM workshop on cloud computing security - CCSW, pp.43-54.
4. Bolosky WJ, Douceur JR, Ely D, Theimer M (2000) Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. SIGMETRICS 2000, pp.34-43.
5. Curtmola R, Khan O, Burns R, Ateniese G (2008) MR-PDP: Multiple-Replica Provable Data Possession. 28th Distributed Computing Systems Conf. - ICDCS, pp.411-420.
6. Zhang Z, Lian Q, Lin S, Chen W, Chen Y, Jin C (2007) Bitvault: A highly reliable distributed data retention platform. ACM SIGOPS Operating Systems Review, 41(2):27-36.
7. Aguilera MK, Janakiraman R, Xu L (2004) Efficient fault-tolerant distributed storage using erasure codes. Tech. Rep., Washington University in St. Louis.
8. Bowers K, Juels A, Oprea A (2009) HAIL: A high-availability and integrity layer for cloud Storage. 16th ACM Computer and communications security Conf. - CCS, pp. 187-198.
9. Dodis Y, Vadhan S, Wichs D (2009) Proofs of Retrievability via Hardness Amplification. 6th Theory of Cryptography Conference on Theory of Cryptography - TCC, pp.109-127.
10. Hendricks J, Ganger GR, Reiter M (2007) Verifying distributed erasure-coded data. 26th ACM Principles of Distributed Computing Symposium, pp.163-168.
11. Ahlswede R, Cai N, Li S, Yeung R (2000) Network information flow. IEEE Trans., 46(4):1204-1216.
12. Li S, Yeung R, Cai N (2003) Linear Network Coding. IEEE Trans., 49(2):371-381.
13. Koetter R, Muriel M (2003) An Algebraic Approach to Network Coding. IEEE/ACM Trans. on Networking (TON), 11(5):782-795.
14. Dimakis A, Godfrey P, Wu Y, Wainwright M, Ramchandran K (2010) Network coding for distributed storage systems. IEEE Trans. Information Theory, 56(9):4539-4551.
15. Li J, Yang S, Wang X, Xue X, Li B (2000) Tree-structured Data Regeneration in Distributed Storage Systems with Network Coding. 29th IEEE Information commun. Conf., pp.2892-2900.
16. Chen B, Curtmola R, Ateniese G, Burns R (2010) Remote Data Checking for Network Coding-based Distributed Storage Systems. ACM workshop on cloud computing security, pp.31-42.
17. Chen HCH, Hu Y, Lee PPC, Tang Y (2014) NCCloud: A Network-Coding-Based Storage System in a Cloud-of-Clouds. IEEE Trans. on Computers, 63(1):31-44.
18. Cash D, Kupcu A, Wichs D (2013) Dynamic proofs of retrievability via oblivious ram. Conf. on the Theory and Applications of Cryptographic Techniques - EUROCRYPT.
19. Elaine S, Emil S, Charalampos P (2013) Practical dynamic proofs of retrievability. CCS, pp.325-336.
20. Chen B, Curtmola R (2012) Robust dynamic remote data checking for public clouds. Proc of. ACM conf. on Computer and communications security - CCS, pp.1043-1045.
21. Wang Q, Wang C, Ren K, Lou W, Li J (2011) Enabling Public Auditability and Data Dynamics for Storage Security in cloud Computing. IEEE Trans. parallel and distributed system, 22(5):847-859.
22. Omote K, Thao T (2015) MD-POR: Multi-source and Direct Repair for Network Coding-based Proof of Retrievability", Int. Journal of Distributed Sensor Networks (IJDSN), ArticleID:586720, Jan 2015.
23. Le A, Markopoulou A (2012) NC-Audit: Auditing for network coding storage. NetCod12, pp.155-160.
24. Le A, Markopoulou A (2012) On detecting pollution attacks in inter-session network coding, 31st IEEE conf. on Computer Communications - INFOCOM, pp.343-351.