IPSJ SIG Technical Report

Vol.2012-CSEC-58 No.21
Vol.2012-SPT-4 No.21
2012/7/19

Improvement of Network coding-based System for Ensuring Data Integrity in Cloud Computing

Kazumasa Omote[1,a)] Tran Thao Phuong[1,b)]

Abstract: Since the amount of information is increasing quickly, the database owners have tendency to outsource their data to external service provider called Cloud Storage. In Cloud Storage, because the service provider may be not fully trustworthy, there are many challenges in securing the data stored on Cloud. We focus mainly on two challenges: data integrity and data availability. To ensure integrity and availability of such outsourced data, researchers proposed POR (Proof of Retrievability) scheme that enables servers on Cloud to demonstrate whether the client's data is retrievable or not. Although some schemes have been proposed based on POR, most of them neither minimize computation cost in repairing the corrupted data nor prevent against small data corruption attack. In this paper, we propose an efficient and secure POR scheme based on network coding approach that can reduce computation cost in repair phase when corruption is detected and Error-correcting code (ECC) approach that can protect against small data corruption.

Keywords: cloud storage, data integrity, data availability, proofs of retrievability.

1. Introduction

Nowadays, many individuals and organizations outsource their data to remote cloud service providers. Such outsourcing of data enables clients to store more data on the cloud storage than on private computer systems. Also, it permits clients to manage their data easily due to the ability of sharing and access from everywhere.

In Cloud Computing, the researchers have considered many scenarios. In our scheme, we consider the scenario that there are two kinds of entity:

- *Client*: This entity is the data owner who publishes his data to Cloud and is fully trusted.
- *Server*: There are multiple servers used to store the data in Cloud. Unlike the client, those servers are untrusted.

As described in Figure 1, the client communicates with the servers through a web server which helps to deliver the data that can be accessed through the Internet.

In our scheme, the data stored in the servers is treated as static data. That means the client only archives and backups the data without updating operations, i.e., insertion, deletion, modification, appending... like the case of dynamic data. In this work, we focus on static data, then we will improve our scheme to dynamic data in future work.

Although outsourcing data reduces storage burden for the client, it has a problem that the servers are untrusted. Thus, this model introduces numerous interesting research challenges: data privacy, data availability and data integrity.

- *Data privacy*: The data needs to be prevented from the disclosure of the outsourced data to the server.
- *Data availability*: Since system aims to remain available at all times, the data needs to be prevented service discorruptions.
- *Data integrity*: The data cannot be modified by the server.

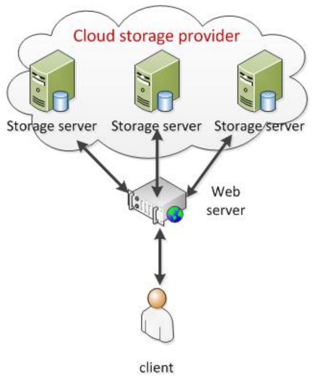


Fig. 1 *Our scenario*

In this work, we concentrate to ensure data availability and data integrity.

Related work There are two approaches of checking data availability and data integrity. In naive approach, the entire data is only stored in single server. The client can periodically check data possession at the server and can thus detect data corruption [3][4][5]. However, when corruption is detected, the single server does not allow data recovery. Therefore, the data has tendency to be stored redundantly at multiple servers [1][2][6][7]. By this way, once data corruption is detected at any of those servers, the client can use the remaining healthy servers to restore the corrupted data.

R. Curtmola et al. [6] proposed *replication* technique in which the client stores one file replica on each server. When corruption

1 Japan Advanced Institute of Science and Technology (JAIST), 1-1 Asahidai, Nomi, Ishikawa 923-1292 Japan
a) omote@jaist.ac.jp
b) tpthao37@jaist.ac.jp

is detected, the client uses one of healthy replicas to recover the corrupted data. However, replication method has a drawback that it has high storage cost due to multiple replicas.

Since replication technique is not efficient in storage cost, the data is applied *erasure coding* technique [7][2]. *Erasure coding* can reduce storage cost because it stores file block instead of file replica like *replication*. However, the drawback of *erasure coding* is that it has high computation cost because the client has to reconstruct the entire file to create a new coded block.

To overcome the weak point of erasure coding, Chen et al. then proposed RDC-NC [1] that applied a technique called *network coding* to distributed storage system. Unlike the erasure coding, the client does not need to reconstruct the entire file in order to generate new coded blocks for a new server. The drawback of this method is that the new coded blocks are different from the original coded blocks so that we need to preserve ability to check the integrity of new coded blocks.

By using network coding approach, RDC-NC not only has low cost in repair phase, but also can prevent *replay attack and pollution attack*. However, RDC-NC cannot prevent a kind of attack called *small data corruption*.

In contrast, HAIL [2] can prevent *small data corruption* but has high computation cost in repair phase because it uses erasure coding instead of network coding like RDC-NC

Contribution Based on these above knowledges, we proposed an efficient and secure scheme that satisfies the following:

- Our scheme can resist replay attack, pollution attack, large data corruption and small data corruption. Our scheme is more secure than RDC-NC [1] which cannot prevent small data corruption.
- Compared with HAIL [2], our computation cost is more efficient than HAIL since we use network coding approach instead of erasure coding.

Organization In Section 2, we give an overview of background of Proof of Retrievability (POR) scheme, network coding and adversarial model. We review previous works in Section 3. Our proposed scheme is described in Section 4, and its security analysis in Section 6. Then, we give conclusion in Section 7 and future works in Section 8. The appendices are in the last of the paper.

2. Preliminary

2.1 Proofs of Retrievability (PORs)

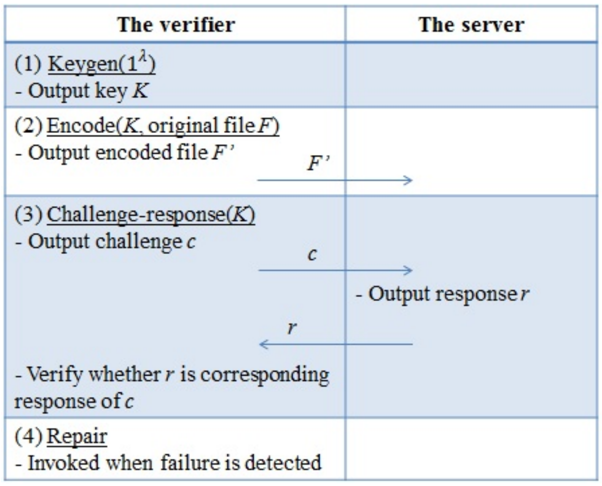
To address data integrity and data availability, researchers have proposed a tools called Proof of Retrievability (POR) [3]. POR is a challenge-response protocol that enables the server to demonstrate whether the file is retrievable or not.

Figure 2 is a holistic POR scheme. Before explaining each phase of POR, we give notations used in that Figure: λ is the security parameter, K is the secret key of the client, F is the original file, F' is the encoded file, c is the challenge generated by the client and r is the response from the server.

A POR scheme has 4 phases as follows:

Keygen The client generates the secret key to use in encoding phase.

Encode The client transforms the raw file to an encoded file, then sends this encoded file to the server.



The verifier	The server
(1) Keygen(1^λ) - Output key K	
(2) Encode(K, original file F) - Output encoded file F' $\xrightarrow{F'}$	
(3) Challenge-response(K) - Output challenge c $\xrightarrow{c}$ $\xleftarrow{r}$ - Verify whether r is corresponding response of c	- Output response r
(4) Repair - Invoked when failure is detected	

Fig. 2 *The holistic POR scheme*

Challenge-response The client generates a challenge and sends it to the server. After receiving the challenge, the server computes the corresponding response and send it to the client. The client then checks whether file F is intact or not.

Repair This algorithm is invoked by the client when failure is detected in challenge-response phase.

2.2 Network coding

Network coding [8] is a method that allows intermediate nodes to transform the data in transit. The sender breaks the message into m vectors $\hat{v}_1, ..., \hat{v}_m$ in an n-dimensional linear space F_q^n. When these message vectors are transmitted to its sibling nodes in the network, the nodes randomly combine the vectors with each other. Recipients can recover the original message from any set of m random linear combinations.

Each vector $\hat{v}$ must carry the coefficients $\alpha_1, ..., \alpha_m \in F_q$ that produce $\hat{v}$ as a linear combination of the original message vectors. Before transmitting, the source node augments each vector $\hat{v}_i$ with m additional components. The resulting vectors $v_1, ..., v_m$ called *augmented vectors* as follows:

$$v_i = (-\hat{v}_i-, \overbrace{\underbrace{0, ..., 0, 1}_{i}, 0, ..., 0}^{m}) \in F_q^{n+m}$$

Each original vector $\hat{v}_i$ is appended with the vector of length m containing a single 1 in the ith position. These augmented vectors are then sent by the source as packets in the network. Observe that if $y \in F_q^{n+m}$ is a linear combination of $v_1, ..., v_m \in F_q^{n+m}$ then the linear combination coefficients are contained in the last m coordinates of y.

Since network coding is an efficient approach in transmitting data in network, researchers applied network coding to distributed storage system. The principle of this method is described as follows: Given an original file F which has m blocks $b_1, \ldots, b_m$, the client chooses coding coefficient vector randomly: $(x_1, \ldots, x_m)$, then, linearly combines coded blocks using the formula $\bar{c} = \sum_{i=0}^{m} x_i.b_i$ and stores those coded blocks on servers. Once corruption is detected, the client retrieves coded blocks from the healthy servers and linearly combines them to regenerate new coded block as Figure 3.

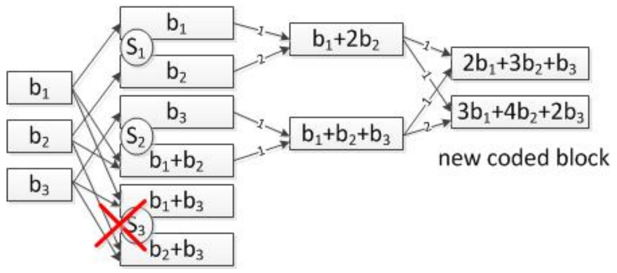


Fig. 3 *Example of network coding.* The file F has three blocks b_1, b_2, b_3. Two coded blocks are stored on each of three servers. When S_3 is corrupted, the client uses two remaining servers to create two new blocks by retrieving one block from each healthy server and mixing them (linear combinations) to obtain two new coded blocks

2.3 Adversarial model

2.3.1 Small data corruption and large data corruption

We consider two common attacks in POR scheme: *small data corruption* and *large data corruption*.

Small data corruption happens when the adversary corrupts the data with small data unit. For example, a small block of whole data is damaged, or even one bit in whole data is flipped. To prevent small corruption, researchers proposed *ECC (Error-correcting code)* technique which expresses the original data and parity information so that any errors can be detected and corrected within certain limitation. An ECC has two parameters n and k where k is the number of original blocks, n is the total number of blocks after adding $n-k$ redundant blocks. An (n,k)-ECC is able to correct $\frac{n-k}{2}$ corruptions. This bound value is ECC's resilience capability.

In contrast, *large data corruption* happens when the adversary corrupts the data with large data unit, e.g, a large block of entire file is corrupted. To detect large data corruption, there is *spot-checking* technique. In this technique, the client firstly partitions the original file into multiple blocks to store on the server. When the client verifies whether the file is intact or not, the client samples a subset of file blocks stored in the server. Then, the server returns a computation over these blocks to the client. The results are checked using some additional information embedded into the file at encoding phase such as MAC (Message Authentication Code), sentinels.

2.3.2 Replay attack and pollution attack

We also consider another typical attacks in network coding: *replay attack* and *pollution attack*.

Replay attack is that the adversary reuses the old coded block to response the client so that the adversary can pass the verification. By this way, the adversary can reduce the redundancy on the servers. For example, there are three coded blocks CB_1, CB_2, CB_3 which are stored on three servers S_1, S_2, S_3 (each coded block per server). In challenge-response phase, after the client challenges three servers, S_1, S_2 will return CB_1, CB_2, respectively. However, the adversary lets S_3 return CB_2 instead of CB_3. Thus, the adversary can reduce redundancy on the servers.

In the case of *pollution attack*, the adversary uses the correct data to avoid detection in challenge-response phase, but cheats the verifier by using the corrupted data in repair phase. For example, there are three servers S_1, S_2, S_3 in which S_1 is corrupted. The client has to recover S_1 by contacting with remaining healthy servers to request their coded blocks. Assume that adversary corrupts S_2 and uses the correct data to avoid detection in challenge-response phase. Thus, the client still thinks that S_2 is a healthy server and requests S_2 its coded block in repair phase. After that, S_2 just returns wrong coded block to the client without letting the client know whether returned coded block is true or not.

3. Previous works

3.1 RDC-NC

B.Chen et at. [1] proposed RDC-NC in which the authors applied *network coding* to remote data checking, instead of *erasure coding*. This work has three main achievements:

- RDC-NC has low computation cost in repair phase because by using network coding, the client does not need to reconstruct the entire file to recover the corrupted block when corruption is detected.
- RDC-NC can protect against replay attack by using challenge tags which act as a counter and is increase when the blocks are recreated due to failure.
- RDC-NC can prevent pollution attack by using repair tags which allow the client to check whether the server combines the blocks correctly or not during the repair phase.

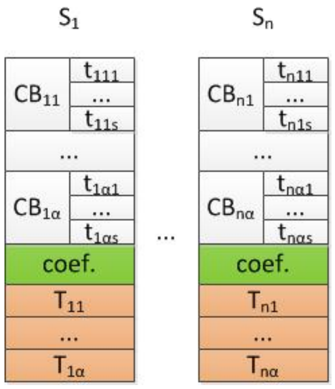


Fig. 4 *RDC-NC approach*

As described in Figure 4, RDC-NC has n servers. Each server stores:

- $\{CB_{ij}\}_{i=[1,n],j=[1,\alpha]}$: coded blocks where i is server index, j is coded block index, α is the total number of coded blocks in each server.
- $\{t_{ijk}\}_{i=[1,n],j=[1,\alpha],k=[1,s]}$: challenge tags for each coded block where i is server index, j is coded block index, k is challenge tag index, s is the total number of segments of a coded block.
- $coef.$: coefficients which are used to constructed coded blocks.
- $\{T_{ij}\}_{i=[1,n],j=[1,\alpha]}$: repair tags for each coded block where i is server index, j is repair tag index, α is the number of repair tags.

We now briefly describe RDC-NC through 4 phases of a POR scheme.

Keygen This algorithm generates the secret key for the client to use in encoding phase.

Encode For each server, the client performs:

(1) *Step 1*: Choose coefficients to compute coded blocks
(2) *Step 2*: Compute coded blocks from original data
(3) *Step 3*: Generate challenge tags and repair tags for each coded block and then embed them to that coded block.
(4) *Step 4*: Send to the server: coded blocks, encrypted coefficients, challenge tags and repair tags.

Challenge-response For each server, the client checks possession of each coded block using *spot-checking* technique.
(1) *Step 1*: The client generates a set of queries and send them to all servers
(2) *Step 2*: Each server computes a proof of possession. After that, the server sends to the client: challenge tag, proof of possession, encrypted coefficients.
(3) *Step 3*: The client checks the validity of the proof of possession.

Repair This phase is invoked when a corrupted server is detected.
(1) *Step 1*: The client contacts with remaining healthy servers to ask them to generate a new coded block.
(2) *Step 2*: The client combines these coded blocks to generate α new coded blocks and metadata (embed challenge tags and repair tag for the block).
(3) *Step 3*: The client sends to the new server: new coded block, encrypted coefficients, challenge tags and repair tags.

Although RDC-NC obtains important achievements as mentioned above, this scheme has a drawback that it cannot protect against small data corruption because it uses *spot-checking* in the challenge-response phase which is only efficient for detecting large data corruption.

3.2 HAIL

The second scheme we consider is HAIL [2]. In HAIL, small data corruption can be prevented by using Error-correcting code (ECC). However, HAIL has high computation cost in repair phase because the authors applied erasure coding approach instead of network coding approach like RDC-NC. We can see that HAIL has opposite achievement and drawback with RDC-NC.

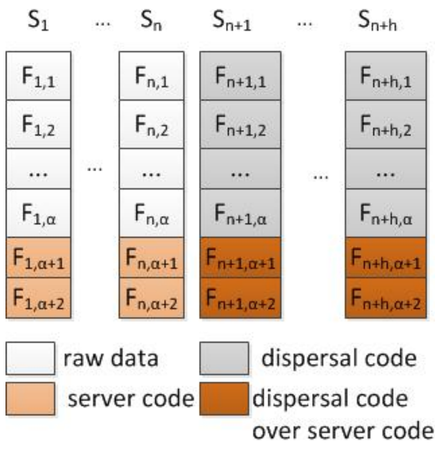

S1 ... Sn Sn+1 ... Sn+h
F1,1 F1,2 ... F1,α F1,α+1 F1,α+2
Fn,1 Fn,2 ... Fn,α Fn,α+1 Fn,α+2
Fn+1,1 Fn+1,2 ... Fn+1,α Fn+1,α+1 Fn+1,α+2
Fn+h,1 Fn+h,2 ... Fn+h,α Fn+h,α+1 Fn+h,α+2
raw data
dispersal code
server code
dispersal code over server code

Fig. 5 *HAIL scheme*

As defined in Figure 5, HAIL has two kinds of servers: primary servers ($S_1, ..., S_n$) and secondary servers ($S_{n+1}, ..., S_{n+h}$)
(1) *Primary server*: Each primary server stores:
- The raw file blocks F_{ij} where i is server index, j is block index. Note that HAIL stores raw file blocks instead of coded blocks like RDC-NC.
- *Server code* parity blocks over the raw file blocks. Server code is an advanced ECC which can achieve higher error resilience than a classic ECC [5].

(2) *Secondary server*: Each secondary server stores:
- *Dispersal code* parity blocks over the raw file blocks. Dispersal code is combination of ECC and MAC (Message Authentication Code).
- Dispersal code parity blocks over server code parity blocks.

If corruption is detected in challenge-response phase, HAIL firstly uses ECC in dispersal code to correct corrupted data. As we mentioned in Section 2.3.1, an $(n, k)-ECC$ is only able to correct $\frac{n-k}{2}$ errors. If the number of detected corruptions is more than $\frac{n-k}{2}$, HAIL then uses server code to correct them. Thus, HAIL can recover the corrupted data with high probability.

We now briefly describe HAIL in 4 phases of POR scheme:

Keygen The client generates the keys for dispersal code, server code and challenging.

Encode The client performs:
(1) *Step 1*: Partition the raw file into multiple file blocks.
(2) *Step 2*: Add server code [5] for each column.
(3) *Step 3*: Adding dispersal code: By applying the dispersal code IP-ECC given in Appendix A.4 to the column $1, ..., n$, we can obtain the column $n + 1, ..., n + h$
(4) *Step 4*: Compute MAC for whole file

Challenge-response
(1) *Step 1*: The client sends a challenge to all servers.
(2) *Step 2*: The servers compute responses to send to the client.
- Upon receiving the challenge, server S_i derives a random subset of row indices.
- Each server computes the response.

(3) *Step 3*: The client verifies the responses from all servers
The client calls MVerECC algorithm of dispersal code as described in Appendix A.4 on those responses and verifies them.

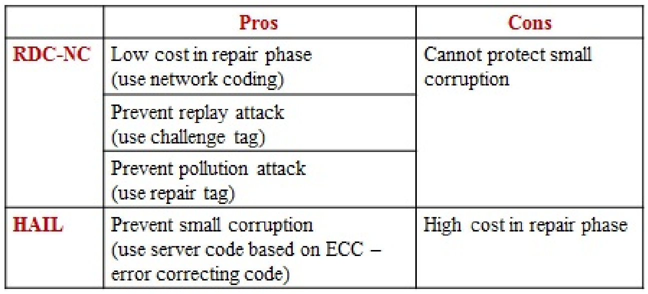
Repair When corruptions are detected, the client downloads the file shares of all servers, and decodes them. Once the client decodes the original file, he can reconstruct the shares of the corrupted servers as in the original encoding algorithm.

3.3 Problem statements

Both RDC-NC and HAIL obtain important advantages, but they still exist drawbacks. We summarize good points and weak points of both schemes as Table 6:

We have comment that:
- The good point of RDC-NC is the weak point of HAIL: RDC-NC has low computation code in repair phase while HAIL has high computation cost in repair phase.
- The weak point of RDC-NC is the good point of HAIL: RDC-NC cannot prevent small data corruption while HAIL can do that by using ECC.

Based on that comment, we give our problem statements that how to design an efficient and secure protocol which satisfies the following:



	Pros	Cons
RDC-NC	Low cost in repair phase (use network coding)	Cannot protect small corruption
	Prevent replay attack (use challenge tag)	
	Prevent pollution attack (use repair tag)	
HAIL	Prevent small corruption (use server code based on ECC – error correcting code)	High cost in repair phase

Fig. 6 *Summary of the good points and weak points of RDC-NC and HAIL*

- The scheme can minimize the cost of repair phase by using network coding.
- The scheme can solve small data corruption.
- The scheme can prevent replay attack and pollution attack.

4. Proposed scheme

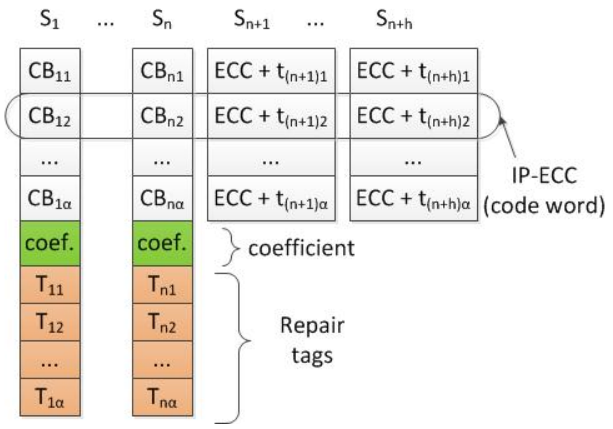

S1 ... Sn Sn+1 ... Sn+h
CB11 CBn1 ECC + t(n+1)1 ECC + t(n+h)1
CB12 CBn2 ECC + t(n+1)2 ECC + t(n+h)2
CB1α CBnα ECC + t(n+1)α ECC + t(n+h)α
IP-ECC (code word)
coef. coef.
coefficient
T11 T12 T1α Tn1 Tn2 Tnα
Repair tags

Fig. 7 *Our approach*

4.1 The structure of our scheme

In our scheme, we employ $n + h$ servers. The first n servers ($S_1, ..., S_n$) are quite similar to the structure RDC-NC excepting that RDC-NC has the tags t while ours has not. The structure of remaining h servers ($S_{n+1}, ..., S_{n+h}$) is the same with HAIL's dispersal code.

(1) Each of servers $\{S_i\}_{i=[1,n]}$ stores:

- $\{CB_{ij}\}_{i=[1,n],j=[1,\alpha]}$: coded blocks where i is server index, j is coded block index, α is the number of coded blocks stored in each of servers.
- $coef.$: coefficients which are used to constructed coded blocks.
- $\{T_{ij}\}_{i=[1,n],j=[1,\alpha]}$: repair tags for each block where i is server index, j is repair tag index, α is the number of repair tags in each server.

(2) Each of servers $\{S_i\}_{i=[n+1,n+h]}$ stores:

- ECC parity block of each row.
- $\{t_{ij}\}_{i=[1,n],j=[1,\alpha]}$: the *MAC* of each row.

4.2 Our improvements

There are three improvements in our scheme:

Improvement 1 Since RDC-NC has low cost in repair phase unlike HAIL and since HAIL can prevent small data corruption by using ECC unlike RDC-NC, thus we try to apply ECC to RDC-NC so that we still keep the advantages of RDC-NC and can prevent small data corruption by ECC.

Improvement 2 We integrate coded blocks, ECC and MAC to one code called IP-ECC (Integrity-protected error-correcting code) [2] as described in Appendix A.4. Although HAIL also has IP-ECC, it only consider the raw file block instead of coded block because HAIL does not use network coding that conduces high cost in repair phase.

Although our scheme has more storage cost than RDC-NC, we can prevent replay attack since the coded blocks is integrated into a concrete IP-ECC code and the adversary cannot reuse the old coded block.

Improvement 3 We challenge the servers based on row (codeword IP-ECC) [see Figure 7] like HAIL. While in RDC-NC, the client can only challenge one server per challenge, in our scheme one challenge can consider $n + h$ servers.

4.3 Tag construction

In this section, we describe the tag construction in our scheme and also compare with tag construction in RDC-NC.

Consider each row in RDC-NC scheme and our scheme, one coded block in RDC-NC has s tags t where s is the number of segments of each coded block as described in Figure 8. Thus, the total number of tags t in each row of RDC-NC is $s \times n$.

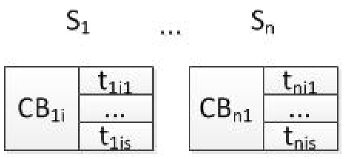


Fig. 8 *Row i in RDC-NC scheme*

Meanwhile, each row in our scheme only has h tags which are stored in the servers $S_{n+1}, ..., S_{n+h}$ as described in Figure 9. So, we can see that the total number of tags in our scheme is less than in RDC-NC. This is because we use *aggregated tag* combined with ECC.

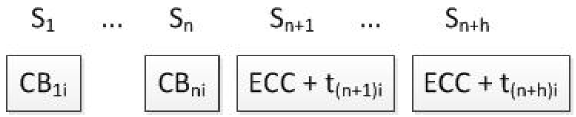


Fig. 9 *Row i in our scheme*

Aggregated tag is a kind of MAC. To compute aggregated tag combined with ECC, our technique is based on RS-UH given in [2] and also described in Appendix A.2. In row i, on input n coded blocks $CB_{1i}, ..., CB_{ni}$, to encode these coded block with aggregated tag combined with an ECC (use $(n + h, n)$-Reed Solomon code), we apply this formula:

$$\begin{aligned} CB'_{ij} &= RS-UHF_{K_i}(CB_{i1}, ..., CB_{in}) + f_{K'_i}(r) \\ &= (CB_{in}K_i^{n-1} + CB_{i(n-1)}K_i^{n-2} + ... + CB_{i1}) + f_{K'_i}(r) \end{aligned}$$

where f is a pseudo-random function and r is a random value.

4.4 Our approaches in details

We now describe our approach in detail through each phase of POR scheme. Firstly, we introduce some conventions used in our scheme:

- The original file F has m blocks: $F = (b_1, ..., b_m)$
- α is the number of coded blocks stored in each server
- $GF(p)$ is Galois field of integers modulo p where p is a large prime.
- f is a pseudo-random function (PRF): $f : \{0,1\}^* \times \{0,1\}^\kappa \rightarrow GF(p)$
- $RS - UHF$ is the universal hash function (UHF) based on Reed-Solomon code (a kind of ECC) given in [2] and presented in Appendix A.2.
- $RS-UHF_K(message)+g_{K'}(random\ value)$ is a form of MAC obtained from UHF. This form is used to compute dispersal code parity blocks in encoding phase.

Keygen The client generates the secret keys to use in encoding phase: $\{K_{rtag}, K'_{rtag}, \{K_i, K'_i\}_{i=[n+1,n+h]}\}$ where each key is randomly chosen in $\{0,1\}^\kappa$

Encode Like RDC-NC, each server in our scheme also stores α coded blocks, coefficients, one repair tag for each coded block. However, the difference from RDC-NC is that:

- We add ECC and MAC for each row as described in Figure 7
- We do not use *challenge tag* like RDC-NC for preventing *replay attack* since our scheme can prevent this attack by concrete IP-ECC code without needing challenge tags.

The following part is the detail of encoding algorithm:

(1) *Step 1: Transform the raw file to coded blocks*
$\forall i = [1, n]$:
- Generate u values $\lambda_1, ..., \lambda_u : \lambda_k = f_{K_{rtag}}(i||k)$ where $k = [1, u]$
- $\forall j = [1, \alpha]$:
  - $\forall k = [1, m]$: generate m coefficients $z_{ijk} \overset{R}{\leftarrow} GF(p)$
  - Compute coded blocks: $CB_{ij} = \sum_{k=1}^{m} z_{ijk} b_k$

Therefore, we have matrix $\{CB_{ij}\}_{i=[1,n], j=[1,\alpha]}$

(2) *Step 2: Compute ECC and MAC of coded blocks to store in server $S_{n+1}, ..., S_{n+h}$*
$\forall j = [1, \alpha]$:
$\forall i = [n+1, n+h]$:
$$\begin{aligned} CB'_{ij} &= RS - UHF_{K_i}(CB_{i1}, ..., CB_{in}) + f_{K'_i}(\tau_{ij}) \\ &= (CB_{in}K_i^{n-1} + CB_{i(n-1)}K_i^{n-2} + ... + CB_{i1}) + f_{K'_i}(\tau_{ij}) \end{aligned}$$
where τ is position index that depends on the file handle, server index i, block index j, i.e, hash of file name, i and j. The details of RS-UHF is presented in Appendix A.2.

(3) *Step 3: Add repair tags*
$\forall i = [1, n]$
- Generate u values $\lambda_1, ..., \lambda_u : \lambda_k = f_{K_{rtag}}(i||k)$ where $k = [1, u]$
- $\forall j = [1, \alpha]$
  - View CB_{ij} as a column vector of u symbols $CB_{ij} = (CB_{ij1}, ..., CB_{iju})$ with $CB_{ijk} \in GF(p)$ where $k = [1, u]$
  - Compute a repair tag for CB_{ij}:
    $T_{ij} = f_{K'_{rtag}}(i||j||z_{ij1}||...||z_{ijm}) + \sum_{k=1}^{u} \lambda_k CB_{ijk}\ mod\ p$

(4) *Step 4: Client sends following data to the server $S_i (i = [1, n])$:*
- $\{CB_{ij}\}_{j=[1,\alpha]}$
- $\{z_{ijk}\}_{j=[1,\alpha],k=[1,m]}$
- $\{T_{ij}\}_{j=[1,\alpha]}$

Challenge-response Unlike RDC-NC which uses *spot-checking* that is only efficient for detecting large corruption and cannot detect and correct small corruption, our scheme uses ECC to overcome the drawback of RDC-NC. Concretely, our approach is based on HAIL: In each challenge, the client chooses the number of row indices to challenge the servers. The servers then return the corresponding codewords to the client. After receiving these codewords, the client checks whether these codewords are valid or not by using the MVerECC algorithm of IP-ECC as presented in Appendix A.4.

(1) *Step 1: Client challenges the servers*
The clients sends to each server a set of row indices $D = \{j_1, ..., j_v\}$ and a key $k \in I$ where I is a field with operation $(+, \times)$, i.e, $GF[2^{128}]$

(2) *Step 2: The servers respond the client*
After receiving the challenge from the client, each server S_i responds: $R_i = RS - UHF_k(CB_{ij_1}, ..., CB_{ij_v})$

(3) *Step 3: The client verifies the responses from the servers*
The client calls MVerECC algorithm of IP-ECC [see Appendix A.4] to verify the responses from the servers: $MVerECC(R_1, ..., R_{n+1})$
- Return $fail$ if the codewords is not valid
- Otherwise, return *true*

Repair Once the failure is detected, the repair phase consists two sub phases:

- *Sub phase 1:* The corrupted data is firstly corrected by ECC.
- *Sub phase 2:* Since an *(n,k)*-ECC only incurs $\frac{n-k}{2}$ corruptions, if the number of corruption is over $\frac{n-k}{2}$, we cannot use ECC to repair the corrupted data. Thus we use the second sub phase based on RDC-NC: the client will contact with healthy servers and asks them to generate a new coded blocks, then combines these coded blocks to generate α coded blocks using linear combination and stores them on a new server.

Assume S_y is the corrupted server which stores blocks $(CB_{y1}, ..., CB_{y\alpha})$

(1) *Step 1: The client contacts with l healthy servers $S_{i_1}, ..., S_{i_l}$ to ask them to compute a new coded block and the proof of correct encoding.*
$\forall i = [i_1, i_l]$
- Generate coefficients $(x_{i1}, ..., x_{i\alpha})$ where $x_{ik} \overset{R}{\leftarrow} GF(p)$ with $k = [1, \alpha]$
- The client sends the request to S_i to compute a new coded block and the proof of correct encoding.
- The server S_i does:
  - Compute $\overline{a_i} = \sum_{j=1}^{\alpha} x_{ij} CB_{ij}$
  - Compute a proof of correct encoding: $\theta = \sum_{j=1}^{\alpha} x_{ij} T_{ij}\ mod\ p$
  - Send $\overline{a_i}, \{z_{ij1}, ..., z_{ijm}\}_{j=[1,\alpha]}$ to the client
- The client re-generates u values $\lambda_1, ..., \lambda_u : \lambda_k = f_{K_{rtag}}(i||k)$ where $k = [1, u]$
- Check if $\theta \neq \sum_{j=1}^{\alpha} x_{ij} f_{K'_{rtag}}(i||j||z_{ij1}||...||z_{ijm}) + \sum_{k=1}^{u} \lambda_k a_{ik}$

where $a_{i1}, ..., a_{iu}$ are symbols of block $\overline{a_i}$

(2) *Step 2: Recover the corrupted data in the server S_y*

- Generate u values $\lambda_1, ..., \lambda_u$: $\lambda_k = f_{K_{rtag}}(y||k)$ where $k = [1, u]$
- $\forall j = [1, \alpha]$:
  - $\forall k = [1, l]$: generate coefficients $z_{yjk} \overset{R}{\leftarrow} GF(p)$
  - Compute coded block: $CB_{yj} = \sum_{k=1}^{l} z_{yjk}\overline{a_k}$
  - View CB_{yj} as a column vector of u symbols $CB_{yj} = (CB_{yj1}, ..., CB_{yju})$ with $CB_{yjk} \in GF(p)$
  - Compute repair tag for the block CB_{yj}: $T_{yj} = f_{K'_{rtag}}(i||j||z_{yj1}||...||z_{yjl}) + \sum_{k=1}^{u} \lambda_k CB_{yjk}$ *mod p*

(3) *Step 3: The client sends the following data to the new server S'_y*:

- $\{CB_{yj}\}_{j=[1,\alpha]}$
- $\{z_{yjk}\}_{j=[1,\alpha],k=[1,l]}$
- $\{T_{yj}\}_{j=[1,\alpha]}$

5. Security analysis

5.1 Data recover condition

Data recover condition of network coding: *The data original can be recovered as long as at least k out of the n servers collectively store at least m coded blocks which are linearly independent combinations of the original m file blocks* [1]. Thus, as long as we choose the parameters to satisfy this condition, our scheme can recover data corruptions.

5.2 Repairing data in the servers $S_{n+1}, ..., S_{n+h}$

We consider two cases when corruption happens:

- *Case 1*: Corruption is detected in $S_1, ..., S_n$
  The corrupted blocks in $S_1, ..., S_n$ can be recovered by ECC or from another healthy coded blocks using linear combination of network coding technique.
- *Case 2*: Corruption is detected in $S_{n+1}, ..., S_{n+h}$
  Unlike $S_1, ..., S_n$, the servers $S_{n+1}, ..., S_{n+h}$ do not store coded blocks. Thus, the corrupted block cannot be recovered using network coding technique, but they can be recovered by ECC. To tolerate more corruptions, we can encode the data in $S_{n+1}, ..., S_{n+h}$ by *server code* [5] as described in Figure 10 because server code is an advanced ECC which can achieve higher error resilience than a classic ECC [5].

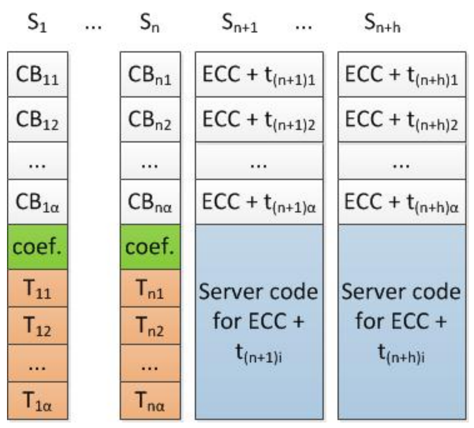


Fig. 10 *The approach for efficient repairing of the servers $S_{n+1}, ..., S_{n+h}$*

6. Conclusion

We have proposed a new POR protocol in which the client can ensure the data integrity based on RDC-NC scheme. By improving the previous work, our approach can solve the problem of small data corruption by applying ECC to coded blocks. However, there are still some remaining problems that need to be solved in future works.

7. Future works

We leave our following three remaining works in the future:

(1) Compared with RDC-NC, our construction increases storage cost due to ECC parity blocks and MAC. Although in this paper, we can reduce this storage cost by combining them into one codeword, we may minimize more storage cost by combining repair tags into that codeword.

(2) Since the server code can achieve higher error resilience than classical ECC, we can apply server code to coded blocks instead of classical ECC.

(3) We improve our scheme to dynamic data in which the client can perform update operations, i.e, modification, deletion, insertion.

References

[1] Bo Chen, Reza Curtmola, Giuseppe Ateniese, Randal Burns: Remote Data Checking for Network Coding-based Distributed Storage Systems, *Trans. CCSW* (2010).
[2] Kevin D. Bowers, Ari Juels, Alina Oprea: HAIL: A High-Availability and Integrity Layer for Cloud Storage, *Trans. CCS* (2009).
[3] A. Juels, B.Kaliski: PORs: Proofs of retrievability for large files, *Trans. CCS* (2007).
[4] Hovav Shacham, Brent Waters: Compact Proofs of Retrievability, *Trans. ASIACRYPT* (2008).
[5] Kevin D. Bowers, Ari Juels, Alina Oprea: Proofs of retrievability: theory and implementation, *Trans. CCSW* (2009).
[6] R. Curtmola, O. Khan, R. Burns, G. Ateniese: MR-PDP: Multiple-Replica Provable Data Possession, *Trans. ICDCS* (2008)
[7] M. K. Aguilera, R. Janakiraman, and L. Xu: Efcient fault-tolerant distributed storage using erasure codes, *Tech. Rep., Washington University in St. Louis* (2004)
[8] R. Ahlswede, N. Cai, S. Li, and R. Yeung: Network information flow, *IEEE Transactions on Information Theory, 46(4):1204-1216* (2000)

Appendix

A.1 Error-correcting code (ECC)

ECC is an algorithm for expressing a sequence of numbers that includes the original data and additional information (redundancy/parity data) such that any errors can be detected and corrected within certain limitations based on the remaining numbers. An ECC usually has two parameter (n, k) where k is the number of original blocks, n is the total number of blocks after adding $n - k$ redundant blocks.

We now give an example of ECC: *(3,7)-Reed-Solomon code* defined in $GF(929)$. The message $\vec{m} = (3, 2, 1)$ is used in this example.

Encode We have $k = 3, n = 7$ as the parameters of Reed-Solomon code. Let $t = n - k = 4$ and choose $\alpha = 3$.

Firstly, we compute a generator polynomial $g(x)$:

$$\begin{aligned} g(x) &= (x-\alpha)(x-\alpha^2)...(x-\alpha^t) \\ &= (x-3)(x-3^2)(x-3^3)(x-3^4) \\ &= x^4+809x^3+723x^2+568x+522 \end{aligned}$$

Then, we compute the remainder $s_r(x)$:

$$\begin{aligned} s_r(x) &= p(x)x^t \bmod g(x) \\ &= 547x^3+738x^2+442x+455 \end{aligned}$$

Using the remainder $s_r(x)$ to compute the codeword $s(x)$:

$$\begin{aligned} s(x) &= p(x)x^t - s_r(x) \\ &= 3x^6+2x^5+1x^4+382x^3+191x^2+487x+474 \end{aligned}$$

Finally, we send this codeword $s(x)$ to the receiver.

Verify Assume that the codeword $s(x)$ is modified as $r(x)$ with the errors $e(x)$:

$$\begin{aligned} r(x) &= s(x)+e(x) \\ &= 3x^6+2x^5+123x^4+456x^3+191x^2+487x+474 \end{aligned}$$

Thus, our purpose is to check where the errors are in codeword and what error values are.

Firstly, we compute syndromes by evaluating $r(x)$ at powers of α:

- $S_1 = r(3^1) = 3.3^6+2.3^5+123.3^4+456.3^3+191.3^2+487.3+474 = 732$
- $S_2 = r(3^2) = 637$
- $S_3 = r(3^3) = 762$
- $S_4 = r(3^4) = 925$

Then, we calculate the error locator polynomial $\Lambda(x)$ using Berlekamp - Massey algorithm:

n	S_n	C
1	732	$197x+1$
2	637	$173x+1$
3	762	$634x^2+173x+1$
4	925	$329x^2+821x+1$

The finally value of C is the error locator polynomial:

$$\Lambda(x) = 329x^2+821x+1.$$

After obtaining the error locator polynomial, we find the error values by solving the equation in $GF(929)$:

$$\Lambda(x) = (329x^2+821x+1)\ mod\ 929 = 0$$

We have 2 solutions:

- $x_1 = 757 = \frac{1}{3^3}$ because $757.3^3 = 1\ in\ GF(929)$
- $x_2 = 562 = \frac{1}{3^4}$ because $562.3^4 = 1\ in\ GF(929)$

Thus we know the errors happened at the position x^3 and x^4.

Finally, we find error values by applying the Forney algorithm:

$$\begin{aligned} \Omega(x) &= s(x).\Lambda(x) \bmod x^t \\ &= s(x).\Lambda(x) \bmod x^4 \\ &= 546x+843 \\ &= 74 \\ \Lambda'(x) &= 658x+821 \end{aligned}$$

Then, the error values are:

- $e_1 = \frac{-\Omega(x_1)}{\Lambda'(x_1)} = \frac{-649}{54} = 280 \times 843 = 74$ happened in the position x^3
- $e_2 = \frac{-\Omega(x_2)}{\Lambda'(x_2)} = 122$ happened in the position x^4

Therefore, the error is $e(r) = 122x^4 + 74x^3$

Recover We subtract $e(r) = 122x^4 + 74x^3$ from $r(x)$ to reproduce the original codeword $s(x)$:

$$\begin{aligned} s(x) &= r(x) - e(x) \\ &= 3x^6+2x^5+1x^4+382x^3+191x^2+487x+474 \end{aligned}$$

A.2 RS-UHF (Universal hash function based on Reed-Selomon code)

Assume that a message m is a vector $\overrightarrow{m} = (m_1, ..., m_l)$ and we use (n,l)-Reed Solomon code. $\overrightarrow{m}$ can be viewed in terms of a polynomial representation of the form $p_{\overrightarrow{m}} = m_l x^{l-1} + m_{l-1}x^{l-2} + ... + m_1$.

A Reed-Solomon code can be defined in terms of a vector $\overrightarrow{k} = (k_1, ..., k_n)$. The codeword of a message $\overrightarrow{m}$ is the evaluation of polynomial $p_{\overrightarrow{m}}$ at point $(k_1, ..., k_n)$: $(p_{\overrightarrow{m}}(k_1), ..., p_{\overrightarrow{m}}(k_n))$.

A UHF is $h_\kappa(m) = p_{\overrightarrow{m}}(\kappa)$ where κ is the key.

A.3 UMAC (MAC obtained from Universal hash function UHFs)

Let I denote a field with operation $(+,\times)$, i.e, $GF(p)$ where p is large prime Given a UHF family $h : K_{UHF} \times I^l \to I$ and PRF $g : K_{PRF} \times L \to I$, UMAC is a tuple of (UGen, UTag, UVer):

- $UGen(1^\lambda)$: generate key (κ, κ') uniformly at ransom from $K_{UHF} \times K_{PRF}$
- $UTag$: $K_{UHF} \times K_{PRF} \times I^l \to L \times I$ is defined as $UTag_{\kappa,\kappa'}(m) = (r, h_\kappa(m) + g_{\kappa'}(r))$ where r is randomly chosen in L
- $UVer$: $K_{UHF} \times K_{PRF} \times I^l \times L \times I$ is defined as $UVer_{\kappa,\kappa'}(m, (c_1, c_2)) = 1$ if and only if $h_\kappa(m) + g_{\kappa'}(c_1) = c_2$.

A.4 IP-ECC (Integrity-protected error-correcting code)

Assume that we use (n,l)-Reed Solomon code. IP-ECC has 3 algorithms:

- $KGenECC(1^\lambda)$: taking the security parameter λ as the input, this algorithm generates key κ to compute Reed-Solomon code and key κ' to compute MAC.
- $MTagECC_{\kappa,\kappa'}(m_1, ..., m_l)$: This algorithm takes the message as the input. The output is $(c_1, ..., c_n)$ where $c_i = RS - UHF_\kappa(\overrightarrow{m}) + g_{\kappa'}(r)$ in which g is a pseudo-random function and r is a random value.
- $MVerECC_{\kappa,\kappa'}(c_1, ..., c_n)$: This algorithm strips off the PRF as $c'_i = c_i - g_{\kappa'}(r)$ and then uses the decoding algorithm of Reed-Solomon code to obtain the original message $\overrightarrow{m} = (m_1, ..., m_l)$.