A New Efficient and Secure POR Scheme Based on Network Coding

Kazumasa Omote
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa, Japan 923-1292
Email: omote@jaist.ac.jp

Tran Phuong Thao
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa, Japan 923-1292
Email: tpthao@jaist.ac.jp

***Abstract*—Information is increasing quickly, database owners have tendency to outsource their data to an external service provider called Cloud Computing. Using Cloud, clients can remotely store their data without burden of local data storage and maintenance. However, such service provider is untrusted, therefore there are some challenges in data security: integrity, availability and confidentiality. Since integrity and availability are prerequisite conditions of the existence of a system, we mainly focus on them rather than confidentiality. To ensure integrity and availability, researchers have proposed network coding-based POR (Proof of Retrievability) schemes that enable the servers to demonstrate whether the data is retrievable or not. However, most of network coding-based POR schemes are inefficient in data checking and also cannot prevent a common attack in POR: *small corruption attack*. In this paper, we propose a new network coding-based POR scheme using *dispersal code* in order to reduce cost in checking phase and also to prevent small corruption attack.**

***Keywords*-data integrity, data availability, proof of retrievability, network coding**

I. Introduction

Nowadays, many individuals and organizations outsource their data to remote cloud service providers. Such outsourcing of data enables customers to store more data on cloud storage than on private computer systems. Also, it permits clients to manage their data easily due to the ability to share and access the data from anywhere through a web browser. Although data outsourcing reduces storage burden for the client, one problem of this method is that such provider is untrusted. Therefore, this model introduces interesting research challenges: data confidentiality, data availability and data integrity. Since integrity and availability are pre-conditions of the existence of a system, they are more important than confidentiality. In this work, we focus on ensuring integrity and availability.

There are two approaches of checking integrity and availability. The first approach is that the entire data is only stored in a single server [3], [4], [6]. The client can periodically check data possession at the server and can thus detect data corruption. However, this approach does not allow data recovery when a corruption is detected. The second approach is that the client stores data redundantly in multiple servers [1], [2], [5], [7]–[9], [13]–[17]. When a corruption is detected in any of the servers, the client can use remaining healthy servers to restore the corrupted data. In multi-server approach, there are three techniques: *replication*, *erasure coding* and *network coding*.

Curtmola et al. [7] proposed *replication* in which the client stores file replica in each server. When a corruption is detected, the client uses one of healthy replicas to recover. The drawback of this technique is: storage cost is increased because the client needs to store whole file F in each server. Since *replication* is inefficient in storage cost, *erasure coding* can be applied on outsourced data [8] to provide space-optimal data redundancy. *Erasure coding* can reduce storage cost because each server stores file block instead of file replica like *replication*. However, to recover a corrupted data, the client has to reconstruct the entire file before generating a new coded block. Therefore, this method increases computation cost and communication cost in repairing data. To enable efficiency in repairing phase, researchers apply *network coding* to such outsourced data. The fundamental concept of *network coding* was first introduced for satellite communication networks in [10] and then fully developed in [9]. Unlike *erasure coding*, the client does not need to reconstruct the entire file before generating coded blocks, instead coded blocks are retrieved from healthy servers to generate new coded blocks. Therefore, we focus on *network coding* in this work.

To restore security assurances eroded by cloud environments, researchers proposed a basic tool for the client to verify file availability and integrity, called POR (Proof Of Retrievability) [3], [4]. A POR has four phases: keygen, encode, check (challenge-response) and repair. These phases are performed between the client and the untrusted server. Based on POR scheme, many schemes have been proposed, e.g., [13] using replica, [2], [5], [14], [15] using *erasure code*. However, due to the advantage of *network coding*, we are aware of some network coding-based POR schemes. Dimakis et al. [16] applied *network coding* to achieve a remarkable reduction in the communication overhead of the repair component. Li et al. [17] proposed tree-structure data regeneration with *network coding* in distributed storage systems. After that, Chen et al. adapted these previous works to propose RDC-NC [1] which provides a decent solution for efficient data repair by recoding encoded blocks in the healthy servers during the repair procedure. Also, RDC-NC can prevent three attacks in POR: *replay attack*, *pollution attack* and *large corruption attack*. However, RDC-NC has some weaknesses: firstly, it is inefficient in checking corruptions because it only checks one server per challenge; secondly, it cannot prevent a common attack in POR: **small corruption attack** in which the adversary corrupts data with small data unit, i.e., a small block or even one bit of whole data. Protecting against small corruptions protects the data itself, not just the storage resource. Modifying a single bit may destroy an encrypted file or invalidate authentication information.

To address *small corruption*, one solution is to use Error-Correcting Code (ECC) [19] which allows data to be checked for errors and corrected even one bit on the fly. There are many types of ECC, i.e., Hamming code, Golay code, Reed-Muller code, Reed-Solomon code, etc. We pay attention to Reed-

Solomon code because it can be constructed under Universal Hash Function. Bowers et al. [2] then proposed *dispersal code* using Reed-Solomon code in order to prevent *small corruption* and also to ensure the file integrity with high probability. Unfortunately, [2] uses *erasure code* instead of *network coding*.

Contribution In this paper, we propose a new efficient and secure scheme based on *network code* and *dispersal code*. In the best of our knowledge, our scheme is the first in which both *dispersal code* and *network code* are applied in POR. We describe our advantages in Table I.
Security: While RDC-NC cannot prevent *small corruption attack*, we can deal with this attack by using ECC in *dispersal code*.
Efficiency:

- In checking phase: In RDC-NC, the client can only check one server for each challenge (This is because the client challenges and verifies the servers separately). In our scheme, the client can check all n servers for each challenge (This is because the client challenges and checks the servers based on *dispersal code* which consists of coded blocks from all n servers).
- To check data possession, the common solution is to use MAC (Message Authentication Code). In RDC-NC, the number of MACs is $n \cdot \alpha \cdot s$ where n is the number of servers, α is the number of coded blocks in a server, s is the number of segments in a coded block (This is because MAC is embedded in each segment of a coded block called *challenge tag*). In our scheme, the number of MACs is only $t \cdot \alpha$ where t is some servers out of n servers ($t < n$) (This is because we only need MAC for *dispersal code* which is not in all n servers but only t servers). Note that t is independent on the dominant parameters s.
- In repairing phase: RDC-NC only uses *network coding* to recover data. In our scheme, we divide two cases: if the number of corruptions is smaller than ECC's boundary, we use ECC decoder to recover; otherwise we use *network coding* to recover. Therefore, we can recover data with overwhelming probability (This is because the corrupted data is recovered by two repairing layers: ECC decoder and *network coding*). By using ECC, the client does not need to contact the healthy servers to request coded blocks for recovering like *network coding*, instead the client uses parity information in the server itself to recover.

Note: We use the fact that dispersal code is constructed based on UMAC *(a MAC obtained from Universal Hash Function) which is deeply related with network code-based schemes [11], [12]. As a result, we can* ***suitably combine*** *network code and dispersal code in our scheme.*

Organization In Section II, we give preliminaries. We describe adversarial model in Section III. Our proposed scheme are described in Section IV. We analyse our security in Section V. We discuss efficiency analysis and parameter choice in Section VI. Conclusion and future work are given in Section VII.

II. Preliminaries

A. Proof Of Retrievability (POR)

To address integrity and availability, researchers have proposed Proof Of Retrievability (POR) scheme [3]–[5] which is a challenge-response protocol between client (verifier) and server (prover). *We use* POR *as our system model*. A POR scheme has four phases:

1) $\mathsf{keygen}(1^\lambda)$: On input security parameter λ, the client runs this algorithm to generate (sk, pk) where sk is private key and pk is public key. For symmetric key, $pk = \mathsf{null}$.
2) $\mathsf{encode}(sk, F)$: The client runs this algorithm to transform a raw file F to an encoded file F', then sends F' to the server to store.
3) $\mathsf{check}(sk)$: The client uses his private key sk to generate a challenge c and sends c to the server. The server then computes a response r and sends r back to the client. Finally, the client verifies whether file F is intact or not.
4) $\mathsf{repair}()$: The client runs this algorithm when a failure is detected in checking phase. The technique of repairing phase depends on each concrete scheme.

B. Network coding

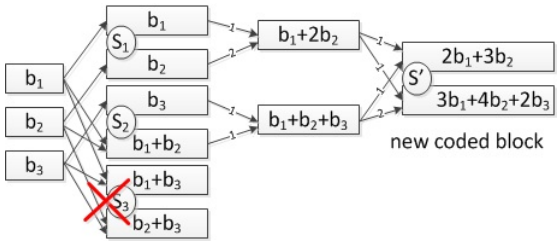

Figure 1. Network coding

Network coding [9]–[11] offers a good trade-off in terms of redundancy, reliability, and repair bandwidth. Assume that the file F has m blocks: $F = \{b_1, \cdots, b_m\}$. The client firstly chooses coding coefficient vector randomly: $x_1, \cdots, x_m$; then, linearly combines n coded blocks using the formula $c = \sum_{i=0}^{m} x_i.b_i$ and stores them in the servers. Note that $x_1, \cdots, x_m$ are chosen so that the corresponding matrix has full rank. Once a corruption is detected, the client retrieves coded blocks from k healthy servers and linearly combines them to regenerate new coded blocks. For example, in Figure 1, let $F = \{b_1, b_2, b_3\}$ be the original file, the client chooses coding coefficients to compute six coded blocks and stores two coded blocks in each server S_1, S_2, S_3. Assume that S_3 is corrupted, the client uses S_1 and S_2 to create new blocks as follows: firstly, the client requests S_1 and S_2 to compute a new coded block by themselves using linear combination, and then the client mixes them (linear combinations) to obtain two new coded blocks.

C. Dispersal code

To prevent *small corruption attack* and enable the client to recover data with a high probability, *dispersal code* has been proposed [2] with a minimal additional storage overhead (just one codeword symbol). *Dispersal code* is constructed from Pseudo-Random Function (PRF), Error-Correcting Code (ECC) and Universal Hash Function (UHF) into a single primitive called *codeword*.

Table I
OUR CONTRIBUTION

		RDC-NC	Our scheme
Security	Small corruption attack	no	yes
Efficiency	Checked servers per challenge	1	n
	MAC in checking phase	$O(n\alpha s)$	$O(t\alpha)$
	Repairing phase	network coding	ECC or network coding

1) Structure: Let (n, l) be parameters of *dispersal code*. Under *dispersal code*, each file block is distributed across n servers. The first l servers store file blocks and $n-l$ servers store dispersal code parity blocks which are used to recover file blocks from failure.

2) Construction: We explain how *dispersal code* works by building blocks as follows:

Universal Hash Function (UHF) [18] is an algebraic function h: $\mathcal{K} \times \mathcal{I}^l \rightarrow \mathcal{I}$ where $\mathcal{I}$ denotes a field with operations $(+, \times)$. This UHF compresses a message $m \in \mathcal{I}^l$ into a compact digest based on a key $\kappa \in \mathcal{K}$ such that the hash of two different messages is different with overwhelming probability over keys. An common UHF is almost exclusive-or universal (AXU) which satisfies:

- h is an ϵ-universal hash function family if $\forall x \neq y \in \mathcal{I}^l$: $\mathsf{Pr}_{\kappa \leftarrow \mathcal{K}}[h_\kappa(x) = h_\kappa(y)] \leq \epsilon$.
- h is an ϵ-AXU family if $\forall x \neq y \in \mathcal{I}^l$, and $\forall z \in \mathcal{I}$: $\mathsf{Pr}_{\kappa \leftarrow K}[h_\kappa(x) \oplus h_\kappa(y) = z] \leq \epsilon$.
- If an UHF is linear, for any message pair (m_1, m_2): $h_\kappa(m_1) + h_\kappa(m_2) = h_\kappa(m_1 + m_2)$.

Error-Correcting Code (ECC) [19] is an algorithm for expressing a sequence of the original data and parity data such that any errors can be detected and corrected. An ECC has two parameters (n, l) where l is the number of original blocks, n is the number of blocks after adding $n-l$ redundant blocks. An (n, l)-ECC can correct even one and up to $t = \frac{n-l+1}{2}$ symbols. Reed-Solomon (RS) code [20] is a kind of ECC which uses a special polynomial. The general form of the generator polynomial is $g(x) = (x - a^i)(x - a^{i+1}) \cdots (x - a^{i+2t})$ and the codeword is $c(x) = g(x) \cdot i(x)$ where $g(x)$ is the generator polynomial, $i(x)$ is the information block, $c(x)$ is a codeword and a is a primitive element of the field.

Encoder: the $2t$ parity symbols are given by: $p(x) = i(x) \cdot x^{n-l} \bmod g(x)$.

Decoder: on input a codeword $r(x)$ which is the original codeword $c(x)$ plus errors: $r(x) = c(x) + e(x)$, RS decoder identifies the position and magnitude of up to t and to correct the errors.

- Syndrome calculation: RS codeword has $2t$ syndromes that depend on errors (not on the transmitted codeword). The syndromes can be calculated by substituting the $2t$ roots of the generator polynomial $g(x)$ into $r(x)$.
- Finding symbol error locations: This involves solving simultaneous equations with t unknowns. Two steps are involved: (i) Finding an error locator polynomial using Berlekamp-Massey algorithm or Euclid's algorithm, (ii) Finding the roots of this polynomial using Chien search algorithm.
- Finding symbol error values: Again, this involves solving simultaneous equations with t unknowns. A widely-used algorithm is Forney algorithm.

Reed-Solomon code based on Universal Hash Function (RS-UHF) [21]: Assume that a message m is a vector $\overrightarrow{m} = (m_1, \cdots, m_l)$ where $m_i \in \mathcal{I}$ and we use (n, l)-RS code over $\mathcal{I}$. The vector $\overrightarrow{m}$ can be viewed in terms of a polynomial representation of the form $p_{\overrightarrow{m}} = m_l x^{l-1} + m_{l-1} x^{l-2} + \cdots + m_1$. A RS code can be defined in terms of a vector $\overrightarrow{k} = (k_1, \cdots, k_n)$. The codeword of a message $\overrightarrow{m}$ is the evaluation of polynomial $p_{\overrightarrow{m}}$ at point $(k_1, \cdots, k_n)$: $(p_{\overrightarrow{m}}(k_1), \cdots, p_{\overrightarrow{m}}(k_n))$. Finally, a UHF is $h_\kappa(m) = p_{\overrightarrow{m}}(\kappa)$ where κ is the key.

Message Authentication Code (MAC) [22] is used to authenticate a message and to detect message tampering and forgery. A MAC is a tuple of $(\mathsf{MGen}, \mathsf{MTag}, \mathsf{MVer})$:

- $\mathsf{MGen}(1^\lambda)$: generates a secret key κ given a security parameter λ.
- $\mathsf{MTag}_\kappa(m)$: computes a tag τ on message m with the key κ.
- $\mathsf{MVer}_\kappa(m, \tau)$: outputs 1 if τ is a valid tag on m, and 0 otherwise.

MAC based on Universal Hash Function (UMAC) [21] can be constructed as the composition of a UHF with a Pseudo-Random Function (PRF). A PRF is a keyed family of functions $g : \mathcal{K}_{\mathsf{PRF}} \times \mathcal{D} \rightarrow \mathcal{R}$ that is, intuitively, indistinguishable from a random family of functions from $\mathcal{D}$ to $\mathcal{R}$. Given a UHF family $h : \mathcal{K}_{\mathsf{UHF}} \times \mathcal{I}^l \rightarrow \mathcal{I}$ and a PRF family $g : \mathcal{K}_{\mathsf{PRF}} \times \mathcal{L} \rightarrow \mathcal{I}$, the construction of UMAC is a tuple of $\mathsf{UMAC} = (\mathsf{UGen}, \mathsf{UTag}, \mathsf{UVer})$:

- $\mathsf{UGen}(1^\lambda)$: generates key (κ, κ') uniformly at random from $\mathcal{K}_{\mathsf{UHF}} \times \mathcal{K}_{\mathsf{PRF}}$.
- $\mathsf{UTag}_{\kappa,\kappa'}(m)$: works in space $\mathcal{K}_{\mathsf{UHF}} \times \mathcal{K}_{\mathsf{PRF}} \times \mathcal{I}^l \rightarrow \mathcal{L} \times \mathcal{I}$, outputs $(r, h_\kappa(m) + g_{\kappa'}(r))$ in which a unique counter $r \in \mathcal{L}$ is increased in each execution.
- $\mathsf{UVer}_{\kappa,\kappa'}(m, (c_1, c_2))$: works in space $\mathcal{K}_{\mathsf{UHF}} \times \mathcal{K}_{\mathsf{PRF}} \times \mathcal{I}^l \times \mathcal{L} \times \mathcal{I}$, outputs 1 if and only if $h_\kappa(m) + g_{\kappa'}(c_1) = c_2$.

Dispersal code [1] allows checking of server responses. To tag a message, the message is encoded under an (n, l)-RS code, and then is applied a PRF to the last s code symbols (where $s = [1, n]$ is a parameter), obtaining a MAC on each of those s code symbols using UMAC. A codeword is valid if at least one of its last s symbols are valid MAC under UMAC on its decoding m.

- $\mathsf{KGenECC}(1^\lambda)$: selects key $\overrightarrow{\kappa} = \{\{\kappa_i\}_{i=1}^n, \{\kappa'_i\}_{i=n-s+1}^n\}$ randomly from space $\mathcal{K} = \mathcal{I}^n \times (\mathcal{K}_{\mathsf{PRF}})^s$. The keys $\{\kappa_i\}_{i=1}^n$ define a RS code. The keys $\{\kappa'_i\}_{i=n-s+1}^n$ are used as PRF keys in UMAC.
- $\mathsf{MTagECC}_{\overrightarrow{\kappa}}(m_1, \cdots, m_l)$: outputs $(c_1, \cdots, c_n)$ where $c_i = \mathsf{RS\text{-}UHF}_{\kappa_i}(\overrightarrow{m}), i = [1, n-s]$ and $c_i = \mathsf{UTag}_{\kappa_i,\kappa'_i}(m_1, \cdots, m_l) = (r_i, \mathsf{RS\text{-}UHF}_{\kappa_i}(\overrightarrow{m}) + g_{\kappa'_i}(r_i)), i = [n-s+1, n]$.
- $\mathsf{MVerECC}_{\overrightarrow{\kappa}}(c_1, \cdots, c_n)$: first strips off the PRF from $c_{n-s+1}, \cdots, c_n$ as: $c'_i = c_i - g_{\kappa'_i}(r_i), i = [n-s+$

$1, n]$, and then decodes $(c_1, \cdots, c_{n-s}, c'_{n-s+1}, \cdots, c'_n)$ using the decoding algorithm of RS to obtain message $\overrightarrow{m} = (m_1, \cdots, m_l)$. If the decoding algorithm of RS code defined by point $\{\kappa_i\}_{i=1}^n$ fails (when the number of corruptions in a codeword is more than $\frac{n-l+1}{2}$), then MVerECC outputs $(\perp, 0)$. If one of the last s symbols of $(c_1, \cdots, c_n)$ is a valid MAC on $\overrightarrow{m}$ under UMAC, MVerECC outputs $(\overrightarrow{m}, 1)$, otherwise it outputs $(\overrightarrow{m}, 0)$.

Since *dispersal code* uses RS code in MTagECC to tag the message and uses the decoder of RS code in MVerECC to verify, *dispersal code* can detect and recover *small corruption*.

III. Adversarial model

We consider an adversary $\mathcal{A}$ whose goals are: (1) to access to encode, check to output codeword c such that $\mathcal{A}$ can pass verification without being tagged in checking phase and (2) to prevent recovering the original file in repairing phase. We define the advantage of $\mathcal{A}$ is: $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{scheme}} = \mathsf{Pr}[\kappa \leftarrow \mathsf{KGenECC}_\kappa(1^\lambda); c \leftarrow \mathcal{A}^{\mathsf{MTagECC}_\kappa(\cdot),\mathsf{MVerECC}_\kappa(\cdot)} : \mathsf{MVerECC}_\kappa(c) = (m, 1) \wedge m$ *is not queried to* $\mathsf{MTagECC}_\kappa(\cdot) \wedge F = \{b_1, \cdots, b_m\}$ *is not recovered*]. **There is one important restriction on $\mathcal{A}$: It can control only $n - k$ out of the n servers within any given time step (epoch)**.

We also consider $\mathcal{A}$ who can perform four following threats:

Small corruption attack: $\mathcal{A}$ corrupts data with small data unit, e.g, a small block of whole data is damaged, or even one bit is flipped. Concretely, $\mathcal{A}$ *corrupts at most a t-fraction of F* (where t is a parameter) to hide data loss incidents. This applies to the servers that wish to preserve their reputation. Data loss incidents may be accidental or malicious. To prevent small corruption, researchers use ECC (Error-Correcting Code) to detect and correct any errors within certain limitation [2], [5]. *t is ECC's boundary defined in Section II.C.2: $t = \frac{n-l+1}{2}$.*

Large corruption attack: $\mathcal{A}$ corrupts the data with large data unit, e.g, a large block of entire file is corrupted. Concretely, $\mathcal{A}$ *corrupts more than a t-fraction of F* (where t is a parameter as in small corruption attack) to discard a significant fraction of the data. This applies to the servers that are financially motivated to sell the same storage resource to multiple clients. To detect large data corruption, *spot checking* was proposed [4], [6] in which the client randomly samples small portions of the data and checks whether they are intact or not. Then the server returns a computation over these portions of the data to the client. The results are checked using some additional information embedded into the file in encoding phase such as MAC, sentinels. *Spot checking* can minimize the I/O at the server and allows the client to detect if a fraction of the data stored in the server has been corrupted. *Spot checking* is only effective in detecting *large data corruption*, cannot prevent small corruption [3], [6].

Replay attack: $\mathcal{A}$ reuses the old coded block to respond the client so that $\mathcal{A}$ can pass checking phase and can reduce the redundancy in the servers. For example, in epoch 1, the client encodes the original file blocks b_1, b_2, b_3 to generate six coded blocks: $c_{11} = b_1, c_{12} = b_2 + b_3$, $c_{21} = b_3, c_{22} = b_1 + b_2$, $c_{31} = b_1 + b_3, c_{32} = b_2 + b_3$. c_{11} and c_{12} are stored in the server S_1. c_{21} and c_{22} are stored in the server S_2. c_{31} and c_{32} are stored in the server S_3. In the end of epoch 1, we assume that S_3 is corrupted. In epoch 2, the client recovers S_3 by generating two new coded blocks: $c'_{31} = b_1 + b_2 + 2b_3$ and $c'_{32} = 2b_1 + b_2$. In the end of epoch 2, we assume that S_1 is corrupted. In epoch 3, S_1 is recovered by two new coded blocks: $c'_{11} = 3b_1 + 3b_2$ and $c'_{12} = 3b_2 + 3b_3$. After that, $\mathcal{A}$ re-uses the old coded blocks c_{31}, c_{32} of S_3 instead of the new coded blocks in order to reduce redundant blocks without being detected. Thus, if S_2 is corrupted in epoch 4, the linear combination between coded blocks of S_1 and S_2 is unable to recover the corrupted data in S_2 any more.

Pollution attack: $\mathcal{A}$ uses valid data to avoid detection in checking phase, but provides invalid data in repairing phase. For example, in epoch 1, the client encodes the original file blocks b_1, b_2, b_3 to six coded blocks: $c_{11} = b_1, c_{12} = b_2 + b_3$, $c_{21} = b_3, c_{22} = b_1 + b_2$, $c_{31} = b_1 + b_3, c_{32} = b_2 + b_3$. c_{11} and c_{12} are stored in the server S_1. c_{21} and c_{22} are stored in the server S_2. c_{31} and c_{32} are stored in the server S_3. In checking phase, we assume that S_3 is corrupted. In repairing phase, S_3 is recovered by two new coded blocks: $c'_{31} = b_1 + b_2 + 2b_3$ and $c'_{32} = 2b_1 + b_2$. At this time, $\mathcal{A}$ suddenly corrupts S_1 without being detected. Because the client does not know that S_1 is corrupted, he still thinks that S_1 is a healthy server. To recover S_3, the client requests coded blocks from the healthy servers S_1 and S_2. S_1 then provides invalid coded blocks to the client.

*Note: in these four threats, we pay attention to protect the scheme against **small corruption attack** since this is one of our contributions. The other attacks: large corruption, replay attack and pollution attack are addressed in RDC-NC, we keep RDC-NC solutions for these three attacks.*

IV. Proposed scheme

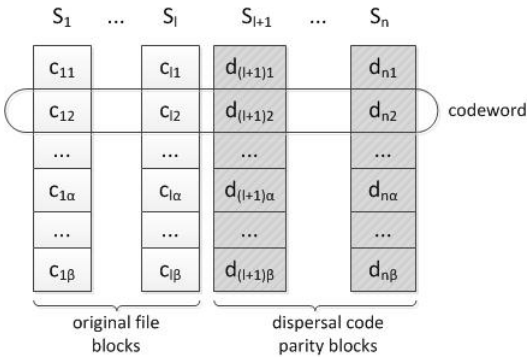

Figure 2. Structure of our scheme

As in Figure 2, we employ n servers: the first l servers $(S_1, \cdots, S_l)$ store coded blocks $\{c_{ij}\}_{i=[1,l],j=[1,\beta]}$; and $n - l$ servers $(S_{l+1}, \cdots, S_n)$ store dispersal code parity blocks $\{d_{ij}\}_{i=[l+1,n],j=[1,\beta]}$ where i is the server index, j is the coded block index, β is the number of blocks in each server. Let $F = (b_1, \cdots, b_m)$ be the original file which has m blocks, let $\mathbb{Z}_p$ be the finite field of integers modulo p where p is a large prime, let f be Pseudo-Random Function (PRF) defined as $f : \{0,1\}^* \times \{0,1\}^\kappa \rightarrow \mathbb{Z}_p$.

A. Keygen

The client generates the secret key: $\mathcal{K} = \{\mathcal{K}_{\mathsf{rtag}}, \mathcal{K}'_{\mathsf{rtag}}, \{\mathcal{K}_i, \mathcal{K}'_i\}_{i=[l+1,n]}, \mathcal{K}_{\mathsf{enc}}\}$ where each key is randomly chosen in $\{0,1\}^\kappa$.

B. Encoding

The client partitions F into m blocks, then computes $l\beta$ coded blocks using linear combination in other to store in servers $S_1, \cdots, S_l$. Each server stores β coded blocks. We denote by c_{ij} these coded blocks (where i is the server index and j is the coded block index in each server). Then, the client encodes the coded blocks using *dispersal code* for each row and stores dispersal code parity blocks in the servers $S_{l+1}, \cdots, S_n$. We denote by d_{ij} these dispersal code parity blocks.

1) *The client computes coded blocks from file blocks:* $\forall i \in [1, l], \forall j = [1, \beta], \forall k \in [1, m]$, the client generates m coding coefficients $z_{ijk} \overset{rand}{\leftarrow} \mathbb{Z}_p$ to compute coded blocks: $c_{ij} = \sum_{k=1}^{m} z_{ijk} b_k$. We have matrix $\{c_{ij}\}_{i=[1,l], j=[1,\beta]}$.
2) *The client computes dispersal code parity blocks in each row and store them in $S_{l+1}, \cdots, S_n$:* $\forall i \in [l+1, n]$, $\forall j \in [1, \beta]$, the client computes: $d_{ij} = \mathsf{MTagECC}_{\mathcal{K}_i, \mathcal{K'}_i}(c_{i1}, \cdots, c_{il})$.
3) *The client computes metadata for coded blocks:* $\forall i \in [1, l]$:
   - Generate u values $\xi_1, \cdots, \xi_u : \xi_k = f_{\mathcal{K}_{\mathsf{rtag}}}(i||k)$ where $k \in [1, u]$.
   - $\forall j \in [1, \beta]$, we view c_{ij} as a column vector of u symbols $c_{ij} = (c_{ij1}, \cdots, c_{iju})$ with $c_{ijk} \in \mathbb{Z}_p$ where $k \in [1, u]$. Then, we compute a repair tag for c_{ij}: $T_{ij} = f_{\mathcal{K'}_{\mathsf{rtag}}}(i||j||z_{ij1}|| \cdots ||z_{ijm})$ + $\sum_{k=1}^{u} \xi_k c_{ijk}$ mod p, and encrypt coefficients: $\forall k \in [1, m], \epsilon_{ijk} = \mathsf{Enc}_{\mathcal{K}_{\mathsf{enc}}}(z_{ijk})$. This encryption is to make *replay attack* become negligible.
4) *The client distributes data to the servers:* The client sends coded blocks $\{c_{ij}\}_{i=[1,l], j=[1,\beta]}$, coefficients $\{\epsilon_{ijk}\}_{i=[1,l], j=[1,\beta], k=[1,m]}$ and repair tags $\{T_{ij}\}_{i=[1,l], j=[1,\beta]}$ to the server S_i where $i = [1, l]$. Then, the clients sends dispersal codes $\{d_{ij}\}_{i=[l+1,n], j=[1,\beta]}$ to the server S_i where $i = [l+1, n]$.

C. Checking

In each challenge, the client chooses a number of row indices to challenge the servers (spot checking). The servers then return the responses to the client. The client checks the responses using MVerECC. Because all servers operate over the same subset of rows, the responses of all servers are checked in a codeword, thus we can consider n servers for each challenge.

1) *The client challenges the servers:* The client sends to each server a set of row indices $D = \{j_1, \cdots, j_v\}$ and a key $k \in \mathcal{I}$ where $\mathcal{I}$ is a field with operation $(+, \times)$.
2) *The servers respond the client:* After receiving query from the client, the server S_i responds: $R_i = \mathsf{RS\text{-}UHF}_k(c_{ij_1}, \cdots, c_{ij_v})$.
3) *The client verifies the servers:* Because all servers operate over the same subset of rows D, the combined response $R = (R_1, \cdots, R_n)$ is a codeword in the dispersal code. The client firstly check the validity of R by calling $\mathsf{MVerECC}(R_1, \cdots, R_n)$ algorithm of *dispersal code* to verify. It returns false if the responses are invalid; otherwise, return true. After checking the validity of R, the client can then check the validity of each individual response R_i to detect which server is corrupted and which server is healthy: R_i is a valid response for the first l servers $(S_1, \cdots, S_l)$ if it matches the i-th symbol in $\overrightarrow{m}$; for $(n - l)$ serves $(S_{l+1}, \cdots, S_n)$, R_i is a valid response if it is a valid MAC on $\overrightarrow{m}$.

D. Repairing

If a failure is detected in checking phase, repairing phase is executed with two sub-phases:

Sub-phase 1: The corrupted data is firstly corrected by RS decoder. Because an *(n,l)*-ECC can incur up to $t = \frac{n-l+1}{2}$ corruptions, thus if the number of corruption is over t, we next use sub-phase 2.

Sub phase 2: The corrupted data is corrected by network coding. The client contacts with the healthy servers and asks each of them to generate a new coded block. Then, the client combines these coded blocks to generate β coded blocks and stores them in a new server. Assume that S_y is corrupted. The blocks in S_y are $(c_{y1}, \cdots, c_{y\beta})$.

1) *The client requests l healthy servers $S_{i_1}, \cdots, S_{i_l}$ to compute a new coded block and the proof of correct encoding:* $\forall i = [i_1, i_l]$:
   - Generating coding coefficients $x_{i1}, \cdots, x_{i\beta}$ where $x_{ik} \overset{rand}{\leftarrow} \mathbb{Z}_p$ with $k = [1, \beta]$.
   - Requesting S_i to compute a new coded block and proof of correct encoding.
   - S_i computes $\overline{a_i} = \sum_{j=1}^{\beta} x_{ij} c_{ij}$, then computes a proof of correct encoding: $\theta = \sum_{j=1}^{\beta} x_{ij} T_{ij}$ mod p and sends $\overline{a_i}, \theta, \{\epsilon_{ij1}, \cdots, \epsilon_{ijm}\}_{j=[1,\beta]}$ to the client.
   - The client decrypts the encrypted coefficients from S_i to get coefficients: $\{\epsilon_{ij1}, \cdots, \epsilon_{ijm}\}_{j=[1,\beta]}$.
   - The client re-generates u values $\xi_1, \cdots, \xi_u : \xi_k = f_{\mathcal{K}_{\mathsf{rtag}}}(i||k)$ where $k = [1, u]$.
   - The client checks if $\theta \neq \sum_{j=1}^{\beta} x_{ij} f_{\mathcal{K'}_{\mathsf{rtag}}}(i||j||z_{ij1}|| \cdots ||z_{ijm}) + \sum_{k=1}^{u} \xi_k a_{ik}$ where $a_{i1}, \cdots, a_{iu}$ are symbols of block $\overline{a_i}$. This verification is to ensure S_i from *pollution attack*.
2) *The client recovers the corrupted data in the server S_y:*
   - The client generates u values $\xi_1, \cdots, \xi_u : \xi_k = f_{\mathcal{K}_{\mathsf{rtag}}}(y||k)$ where $k = [1, u]$.
   - $\forall j = [1, \beta]$, $\forall k = [1, l]$, the client generates coding coefficients $z_{yjk} \overset{rand}{\leftarrow} \mathbb{Z}_p$, then computes coded block: $c_{yj} = \sum_{k=1}^{l} z_{yjk} \overline{a_k}$. By viewing c_{yj} as a column vector of u symbols $c_{yj} = (c_{yj1}, \cdots, c_{yju})$ with $c_{yjk} \in \mathbb{Z}_p$, the client computes repair tag for block c_{yj}: $T_{yj} = f_{\mathcal{K'}_{\mathsf{rtag}}}(i||j||z_{yj1}|| \cdots ||z_{yjl}) + \sum_{k=1}^{u} \xi_k c_{yjk}$ mod p, and encrypts coefficient: $\forall k = [1, l], \epsilon_{ijk} = \mathsf{Enc}_{\mathcal{K}_{\mathsf{enc}}}(z_{ijk})$.
3) *The client sends to the new server $S'y$:* $\{c_{yj}\}_{j=[1,\beta]}$, $\{\epsilon_{yjk}\}_{j=[1,\beta], k=[1,l]}$, $\{T_{yj}\}_{j=[1,\beta]}$.

V. Security Analysis

We give a bound to against the defined adversary and show how we prevent : *small corruption*, *large corruption*, *replay attack* and *pollution attack*.

A. Adversarial checking and repairing

Given a MAC which consists of three algorithms: $\mathsf{MGen}, \mathsf{MTag}, \mathsf{MVer}$ and q_1 is the number of queries to MTag, q_2 is the number of queries to MVer and t is the running time, the advantage of adversary on UMAC [21] is as follows:

Fact 1: *Assume a* UHF h *that* h *is an* ϵ^{UHF}-AXU *family of hash function and* g *is a* PRF *family. Then* UMAC *is a stateful* MAC *with advantage:* $\mathsf{Adv}^{\mathsf{uf-mac}}_{\mathsf{UMAC}}(q_1, q_2, t) \leq \mathsf{Adv}^{\mathsf{prf}}{}_g(q_1, q_2, t) + \epsilon^{\mathsf{UHF}} q_2$ *in which* $\mathsf{Adv}^{\mathsf{prf}}{}_g(q_1, q_2, t)$ *is the maximum advantage of all adversaries to make* $q_1 + q_2$ *queries to its oracle and running in time* t.

Assume that: (1) the PRF is secure and (2) the MAC is unforgeable, we have: $\mathsf{Adv}^{\mathsf{prf}}{}_g(q_1, q_2, t) + \epsilon^{\mathsf{UHF}} q_2 \leq \varepsilon$ (negligible). Therefore, the advantage of UMAC is also negligible: $\mathsf{Adv}^{\mathsf{uf-mac}}_{\mathsf{UMAC}}(q_1, q_2, t) \leq \varepsilon$.

Consider the advantage of all adversaries on a codeword which consists of three algorithms: $\mathsf{KGenECC}, \mathsf{MTagECC}, \mathsf{MVerECC}$. Given q_1 is the number of queries to MTagECC, q_2 is the number of queries to MVerECC and t is the running time, the advantage of all adversaries on codeword is as follows:

Theorem 1: *If* RS-UHF *is constructed from an* (n, l)-RS *code and* g *is a* PRF *family, then the codeword has the advantage:* $\mathsf{Adv}^{\mathsf{uf-ecc}}_{\mathsf{codeword}}(q_1, q_2, t) \leq 2[\mathsf{Adv}^{\mathsf{uf-mac}}_{\mathsf{UMAC}}(q_1, q_2, t)]$.

Proof: Let $\mathcal{A}$ be a successful adversary algorithm for codeword that makes q_1 queries to the tagging oracle MTagECC, q_2 queries to the verification oracle MVerECC and runs in time t. It outputs a codeword $(c_1, \cdots, c_n)$ that decodes to message $\overrightarrow{m} = (m_1, \cdots, m_l)$ such that at least one of the last s symbols in the codeword is a valid MAC on $\overrightarrow{m}$ computed with algorithm UMAC. We consider the adversary $\mathcal{A}'$ for the UMAC construction. $\mathcal{A}'$ is given access to a tagging oracle $\mathsf{UTag}_{\kappa,\kappa'}(\cdot)$ and a verification oracle $\mathsf{UVer}_{\kappa,\kappa'}(\cdot,\cdot)$ and needs to output a new message and tag pair. $\mathcal{A}'$ chooses a position $j \in [n-s+1, n]$ at random, and generates keys $\{\kappa_i\}_{i=1}^{n}$ and $\{\kappa'_i\}_{i=n-s+1}^{n}$ for $i \neq j$. $\mathcal{A}'$ runs $\mathcal{A}$. When $\mathcal{A}$ makes a query to tagging $\overrightarrow{m} = (m_1, \cdots, m_l)$, $\mathcal{A}'$ computes $c_i \leftarrow \mathsf{RS\text{-}UHF}_{\kappa_i}(\overrightarrow{m})$ for $i = [1, n-s]$, and $c_i = \mathsf{UTag}_{\kappa_i,\kappa'_i}(\overrightarrow{m})$ for $i = [n-s+1, n], i \neq j$. $\mathcal{A}'$ calls the UTag oracle to compute $c_j = \mathsf{UTag}_{\kappa,\kappa'}(\overrightarrow{m})$. $\mathcal{A}'$ then responds to $\mathcal{A}$ with $\overrightarrow{c} \leftarrow (c_1, \cdots, c_n)$. When $\mathcal{A}$ makes a query $\overrightarrow{c} = (c_1, \cdots, c_n)$ to the verification oracle, $\mathcal{A}'$ tries to decode $(c_1, \cdots, c_{j-1}, c_{j+1}, \cdots, c_n)$ into message $\overrightarrow{m}$. If decoding fails (there are more than the boundary of RS code: $\frac{n-l+1}{2}$ errors in the codeword), then $\mathcal{A}'$ responds to $\mathcal{A}$ with $(\perp, 0)$. Otherwise, let $\overrightarrow{m}$ be the decoded message. $\mathcal{A}'$ makes a query to the verification oracle $\alpha \leftarrow \mathsf{UVer}_{\kappa,\kappa'}(\overrightarrow{m}, c_j)$ and returns $(\overrightarrow{m}, \alpha)$ to $\mathcal{A}$. Assume that $\mathcal{A}$ outputs an encoding $\overrightarrow{c} = (c_1, \cdots, c_n)$ under the codeword that can be decoded to $\overrightarrow{m}$, such that $\overrightarrow{m}$ was not an input to the tagging oracle and at least one of the last s symbols in $\overrightarrow{c}$ is a valid MAC for $\overrightarrow{m}$. Then $\mathcal{A}'$ outputs $(\overrightarrow{m}, c_j)$. Since the codeword $\overrightarrow{c}$ can be decoded, at least a majority of its parity blocks are correct. Then, with probability at least $\frac{1}{2}$, c_j is a correct MAC on $\overrightarrow{m}$. It follows that $\mathcal{A}'$ succeeds in outputting a correct message and MAC pair $(\overrightarrow{m}, c_j)$ with probability at least half the success probability of $\mathcal{A}$. □

Since $\mathsf{Adv}^{\mathsf{uf-mac}}_{\mathsf{UMAC}}(q_1, q_2, t) \leq \varepsilon$, the advantage on *dispersal code* is: $\mathsf{Adv}^{\mathsf{uf-ecc}}_{\mathsf{codeword}}(q_1, q_2, t) \leq \varepsilon$. In other words, the probability for $\mathcal{A}$ to output codeword c such that $\mathcal{A}$ can pass verification without being tagged in checking phase is negligible: $\mathsf{Pr}[\kappa \leftarrow \mathsf{KGenECC}_\kappa(1^\lambda); c \leftarrow \mathcal{A}^{\mathsf{MTagECC}_\kappa(\cdot),\mathsf{MVerECC}_\kappa(\cdot)} : \mathsf{MVerECC}_\kappa(c) = (m, 1) \wedge m$ *is not queried to* $\mathsf{MTagECC}_\kappa(\cdot)] \leq \varepsilon$.

We now consider the probability for $\mathcal{A}$ to prevent recovering the original file as follows:

Theorem 2 (Data recover condition): *Original blocks can be recovered as long as in any epoch, at least* k *out of* n *servers collectively store at least* m *coded blocks which are linearly independent combinations of the original* m *file blocks and the corresponding matrix has full rank, i.e., rank equals to* m.

Proof: Since F has m blocks: $F = (b_1, \cdots, b_m)$, the number of servers is n and each server stores x coded blocks, thus the total number of coded blocks is $n \times x$. To compute each coded block, the client chooses m coefficients α corresponding with each file block, then uses the linear independent combination: $c_j = \sum_{i=1}^{m} \alpha_{ij} b_i \quad \text{for } j \in [1, nx]$. To recover the original file blocks, we view m file blocks as the variables that need to be solved. To solve m variables, we need at least m coded blocks which make the matrix have full rank because the number of variables in a equation system has to be less than the number of equations.

$$\begin{cases} c_{j_1} = \sum_{i=1}^{m} \alpha_{ij_1} b_i \\ c_{j_2} = \sum_{i=1}^{m} \alpha_{ij_2} b_i \\ \cdots \\ c_{j_m} = \sum_{i=1}^{m} \alpha_{ij_m} b_i \end{cases} \qquad \square$$

Therefore, the number of healthy server has to be at least $k = \frac{m}{x}$ in any epoch. In RDC-NC, there are n servers, α coded blocks in each server, thus the minimum number of healthy servers in an epoch is $\frac{m}{\alpha}$. In our scheme, also n servers are employed but we divide n servers into two types: l servers store coded blocks and $n - l$ servers store dispersal code parity blocks. Since l servers store coded blocks ($l < n$), each server has $\beta = \frac{n \times \alpha}{l}$ coded blocks. Therefore, our minimum number of healthy servers in each epoch is $\frac{m}{\alpha} \times \frac{l}{n}$. If the data recover condition is assumed, the probability for $\mathcal{A}$ to prevent recovering F is negligible: $\mathsf{Pr}[F = \{b_1, \cdots, b_m\}$ *is not recovered*$] \leq \varepsilon$.

B. Small corruption attack

Preventing this attack is one of our contribution, therefore we pay attention to this attack rather than 3 other attacks: *large corruption*, *replay attack*, *pollution attack*.

Theorem 3: *The RS code in dispersal code is sufficient to ensure data detecting and recovering from small corruption attack.*

Proof: Let t_{error} is the number of corruptions in an epoch caused by an adversary $\mathcal{A}$. $\mathcal{A}$ can corrupts at most $\frac{n-l+1}{2}$. Thus, $t_{error} \leq \frac{n-l+1}{2}$. In encoding phase, the client creates $n\alpha$ coded blocks from m original file blocks (step 1), then adds the dispersal code parity blocks into each row using MTagECC algorithm (step 2). MTagECC is constructed from RS-UHF which is based on RS code (Section II.C). RS code is a kind of ECC which can recover a *small corruption* (Section II.C.b). In checking phase, after being challenged by the client, each of n servers responds the client an output of RS-UHF on input some chosen coded blocks (step 2) and sends the response to the client. The client verifies by calling MVerECC algorithm on input the responses of all n servers. After that, repairing phase is performed in case that the number of corruptions is $\leq \frac{n-l+1}{2}$ using sub-phase 1 of repairing phase which uses RS code in dispersal code (MVerECC firstly strips off the PRF then uses RS decoder to obtain the coded blocks). We now prove that how RS code is enough to stand t_{error} errors. Firstly,

since RS is constructed using parameter (n, l), the message is interpreted as the description of a polynomial p of degree less than l which is evaluated at n distinct points $a_1, \cdots, a_n$ of the field $\mathbb{F}$ and the sequence of values is the corresponding codeword $\mathcal{C}$: $\mathcal{C} = \{p(a_1), \cdots, p(a_n)\}$ (Section II.C.2). Since any two different polynomials of degree less than l agree in at most $l-1$ points, this means that any two codewords of the RS code disagree in at least $n-(l-1) = n-l+1$ positions. Also, there are two polynomials that do agree in $l-1$ points but are not equal, thus, the distance of the RS code is $d = n-l+1$. Secondly, since any two strings in $\mathcal{C}$ differ in at least places d, we have: $2t' \leq d$ where t' be the number of errors. Therefore, $t' \leq \frac{n-l+1}{2}$ (This terminology reflects the fact that, given any string s, there is at most one string $c \in \mathcal{C}$ which is within the distance d of t from s). The largest such value t' (called t'_{max}) is referred to as error resilience of the code. Since $t_{error} \leq t'_{max}$, the advantage of $\mathcal{A}$ is always bounded by error resilience of the RS code. □

C. Large corruption attack, replay attack and pollution attack

These three attacks are addressed in RDC-NC. We keep their solutions which are briefly described as follows:

Large corruption attack In checking phase, since the client checks the servers by periodically sampling a set of row indices as *spot checking* method, large corruption attack is prevented.

Replay attack To prevent *replay attack*, the common solution is to use *counter*. RDC-NC uses a different solution: coding coefficients are stored *encrypted* together with coded blocks to prevent $\mathcal{A}$ from knowing how the original blocks were combined to obtain the coded block, thus $\mathcal{A}$'s ability is negligible since it does not know which old coded blocks to replay.

Pollution attack For each coded block c_{ij}, RDC-NC adds a repair tag to each coded block. This tag is formed as MAC which allows the client to check that the server combines the blocks correctly during repairing phase.

VI. Efficiency analysis and Parameter choice

A. Efficiency analysis

(*) *Note: In Table II, we can see that our encoding cost are greater than RDC-NC. However, as in Section I, the client only encodes the data only one time in the beginning but has to performs checking phase and repairing phase many times during system lifetime. Therefore, the costs of these two phases should be aware rather than encoding cost. Our checking and repair costs are more efficient than RDC-NC.*

(**) *Note: We compute storage cost in this part and analyse how to choose parameters in order to make our storage efficient in the subsection of parameter choice.*

Assume that each coded block is in $\mathbb{F}_q^w$ where q is a prime and w is the length. The size of a coded blocks is $w\log_2 q$. Let m be the number of file blocks, n be is the number of servers.

Encoding cost: RDC-NC computes $n\alpha$ coded blocks. In our scheme, we compute $l\beta$ coded blocks, then we compute $(n-l)\beta$ dispersal code parity blocks, thus there are $n\beta = n\alpha \times \frac{n}{l}$ computations. Compared to RDC-NC, we have more $\frac{n}{l}$ times, however encoding phase is performed one time in the beginning.

Checking cost: In one challenge, RDC-NC checks a subset of segment indices in each coded block in one server, thus, in order to check all coded blocks in all n servers, the client needs $n\alpha$ challenge times. In our scheme, in one challenge, we check a subset of row indices based on codeword of *dispersal code* of all multiple servers, thus, in order to check all coded blocks in all n servers, the client needs $\beta = \frac{n\alpha}{l}$ challenge times. Compared to our cost, RDC-NC computation is greater than l times. That is the advantage when we can check multiple servers per challenge instead of one severs like RDC-NC.

Repairing cost: In RDC-NC, the author analyses the cost to repair one corrupted server: $O(k\frac{2|F|}{k+1} + \frac{|F|}{1+\frac{1}{k}})$ (the cost in each healthy server: $\frac{2|F|}{k+1}$, thus the cost in all k healthy servers: $k\frac{2|F|}{k+1}$, the cost in client-side: $\frac{|F|}{1+\frac{1}{k}}$). Since $k = \frac{m}{\alpha}$, their repairing cost is $O(|F|\frac{2\alpha+m}{\alpha+m})$. In our scheme, in case of sub-phase 1, we recover the corrupted server by RS decoder which has computation cost: $O(n\log_2^2 n \log_2\log_2 n)$ [25]. Since n is independent on the dominant parameter $|F|$ which is the size of entire data, the cost of RS decoder is much smaller than RDC-NC. If we let $n = 12$ as in experimental evaluation of RDC-NC, RS can be decoded in only 284 field operations. In case of sub-phase 2, we recover the corrupted server by network coding like RDC-NC: $O(k'\frac{2|F|}{k'+1} + \frac{|F|}{1+\frac{1}{k'}})$. Since $k' = \frac{l}{n}\frac{m}{\alpha}$, our cost is: $O(\frac{3lm|F|}{n\alpha+lm})$. Since $n > l$, the cost in RDC-NC is greater than our cost: $\frac{3m|F|}{m+\alpha} > \frac{3lm|F|}{n\alpha+lm}$.

Required healthy servers for repairing: As in Theorem 2, RDC-NC needs at least $\frac{m}{\alpha}$ healthy servers to recover while our scheme only need $\frac{m}{\alpha} \times \frac{l}{n}$ where $l < n$. Compared to our scheme, the number of healthy servers in RDC-NC is greater than $\frac{n}{l}$ times.

Storage cost: In RDC-NC: The size of coded blocks is $n\alpha\log_2 q$ (because the number of coded blocks is $n\alpha$). The size of MACs for challenge tags is $n\alpha s\log_2 q$ (because there are $n\alpha s$ challenge tags in $\mathbb{F}_q$). The size of repair tags is $n\alpha\log_2 q$ (because there are $n\alpha$ repair tags which are in $\mathbb{F}_q$). Therefore, the storage cost is $O(n\alpha\log_2 q + (s+1)n\alpha\log_2 q)$. In our scheme: The size of coded blocks and dispersal code parity blocks is $w\log_2 q(n\alpha + (n-l)\frac{n\alpha}{l}) = n\frac{n\alpha}{l}w\log_2 q$ (because there are $n\alpha$ coded blocks and $(n-l)\beta$ dispersal code parity blocks where $\beta = \frac{n\alpha}{l}$). The size of repair tags is $n\alpha$ (because there are $n\alpha$ repair tags which are in $\mathbb{F}_q$). Therefore, the storage cost is $O(n\frac{n\alpha}{l}w\log_2 q + n\alpha\log_2 q)$.

B. Parameter choice

For storage cost: RDC-NC storage cost is: $n\alpha w\log_2 q + (s+1)n\alpha\log_2 q$, our storage cost is $n\frac{n\alpha}{l}w\log_2 q + n\alpha\log_2 q$. To make our cost better than RDC-NC, $w(1-\frac{n}{l})+1$ should be greater than 0. This means that we should choose $w > \frac{sl}{n-l}$.

For balancing recovery probability between ECC and network code: Since an (n, l)-ECC can recover up to $t = \frac{n-l+1}{2}$ errors in each row, the adversary can win ECC's boundary if the number of corruptions is greater than t. Furthermore, based on analysis about *data recovery condition* (Theorem 2), our number of required healthy servers has to be at least $\frac{m}{\alpha} \times \frac{l}{n}$, thus an adversary can win if he corrupts more than $\frac{m}{\alpha} \times \frac{l}{n}$. Let $f_1(l) = \frac{n-l+1}{2}, f_2(l) = \frac{m}{\alpha} \times \frac{l}{n}$. Our purpose is to choose l so that the advantage of the adversary is reduced. In other words, we want to increase $f_1(l)$ and $f_2(l)$. If $f_1(l)$ and $f_2(l)$ are considered separately, we have: $f_1(l) = \frac{n-l+1}{2}$ increases

Table II
COMPARISON BETWEEN RDC-NC AND OUR SCHEME

	RDC-NC	**Our scheme**
Encoding computation (*)	$O(n\alpha)$	$O(n\alpha \times \frac{n}{l})$
Checking computation (*)	$n\alpha$	$\frac{n\alpha}{l}$
Repairing computation (*)	$O(\frac{3m\|F\|}{m+\alpha})$	$O(n\log_2^2 n \log_2\log_2 n)$ (RS decode)
		$O(\frac{3lm\|F\|}{n\alpha+lm})$ (Network code)
Required healthy servers (*)	$\frac{m}{\alpha}$	$\frac{l}{n} \cdot \frac{m}{\alpha}(l < n)$
Storage server cost (**)	$O(n\alpha w \log_2 q + (s+1)n\alpha \log_2 q)$	$O(n\frac{n\alpha}{l} w \log_2 q + n\alpha \log_2 q)$

only if l increases, $f_2(l) = \frac{m}{\alpha} \times \frac{l}{n}$ increases only if l decreases. They are not synchronous. Hence, we should balance l between f_1 and f_2. Let $f_1(l) = f_2(l)$ then we determine $l = \frac{n\alpha(n+1)}{2m+n\alpha}$.

VII. CONCLUSION AND FUTURE WORK

We propose a new POR protocol in which *network code* and *dispersal code* are combined to reduce costs in important phases: checking phase and repairing phase and to prevent *small corruption*, *replay attack*, *pollution attack* and *large corruption*.

There is one thing we leave in future work. The repairing phase may spend high communication cost. This is because when the client repairs corrupted blocks by *network coding*, k healthy servers need to provide their coded blocks to the client then the client computes new coded blocks and stores new coded blocks in the new server. We plan to find a new mechanism in which k healthy servers send their coded blocks directly to the new server without sending back to the client. This mechanism can reduce the burden for the client and also reduce communication cost. To do that, we employ a signature scheme such as [23], [24] in which the new server can authenticate coded blocks provided from k healthy servers instead of the client, and can construct new coded blocks for itself.

REFERENCES

[1] B. Chen, R. Curtmola, G. Ateniese and R. Burns, *Remote Data Checking for Network Coding-based Distributed Storage Systems*, ACM Cloud Computing Security Workshop CCSW (2010).

[2] K. Bowers, A. Juels and A. Oprea, *HAIL, A High-Availability and Integrity Layer for Cloud Storage*, the 16th ACM Conference on Computer and Communications Security CCS (2009).

[3] A. Juels and B.Kaliski, *PORs: Proofs of retrievability for large files*, ACM CCS (2007).

[4] H. Shacham and B. Waters, *Compact Proofs of Retrievability*, ASIACRYPT (2008).

[5] K. Bowers, A. Juels and A. Oprea, *Proofs of retrievability: theory and implementation*, CCSW (2009).

[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, *Provable data possession at untrusted stores*, ACM CCS (2007).

[7] R. Curtmola, O. Khan, R. Burns and G. Ateniese, *MR-PDP: Multiple-Replica Provable Data Possession*, IEEE ICDCS (2008).

[8] M. K. Aguilera, R. Janakiraman and L. Xu, *Efficient fault-tolerant distributed storage using erasure codes*, Tech. Rep., Washington University in St. Louis (2004).

[9] R. Ahlswede, N. Cai, S. Li and R. Yeung, *Network information flow*, IEEE Transactions on Information Theory, 46(4):1204-1216 (2000).

[10] R. W. Yeung and Z. Zhang, *Distributed source coding for satellite communications*, IEEE Trans. Inform. Theory (1999).

[11] S. Agrawal and D. Boneh, *Homomorphic MACs: MAC-Based Integrity for Network Coding*, ACNS (2009).

[12] H. Handschuh and B. Preneel, *Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms*, CRYPTO (2008).

[13] Z.Zhang, Q. Lian, S. Lin, W.Chen, Y.Chen and C. Jin, *Bitvault: A highly reliable distributed data retention platform*, SIGOPS (2007).

[14] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows and M. Isard, *A cooperative Internet backup scheme*, USENIX (2003).

[15] J. Hendricks, G. R. Ganger and M. Reiter, *Verifying distributed erasure-coded data*, ACM PODC (2007).

[16] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright and K. Ramchandran, *Network coding for distributed storage systems*, IEEE Transactions on Information Theory (2010).

[17] J. Li, S. Yang, X. Wang, X. Xue and B. Li, *Tree-structured Data Regeneration in Distributed Storage Systems with Network Coding*, IEEE INFOCOM (2010).

[18] L. Carter and M. Wegman, *Universal Hash Functions*, Journal of Computer and System Sciences (1979).

[19] J. Baylis, *Error-Correcting Codes: A Mathematical Introduction*, Boca Raton, FL: CRC Press (1998).

[20] M. Riley and I. Richardson, *An introduction to Reed Solomon codes: principles, architecture and implementation*, prentice-hall (2001).

[21] V. Shoup, *On fast and provably secure message authentication based on universal hashing*, CRYPTO (1996).

[22] D. R. Stinson, *Cryptography - Theory and Practice*, CRC Press, Boca Raton (1995).

[23] D. Catalano, D. Fiore and B. Warinschi, *Efficient network coding signature in the standard model*, PKC (2012).

[24] W. Yana, M. Yanga, L. Lia and H. Fang, *Short signature scheme for multi-source network coding*, Journal Computer Communications (2012).

[25] F. Didier, *Efficient erasure decoding of Reed-Solomon codes*, arXiv:0901.1886v1 [cs.IT] (2009).