Regenerating-Codes-based Efficient Remote Data Checking and Repairing in Cloud Storage

Jing Chen*, Yuling Peng*, Ruiying Du†, Quan Yuan‡ and Minghui Zheng‡

*Computer School, Wuhan University, Wuhan, China
†Computer School, Department of Math and Computer Science, University of Texas-Permian Basin, TX, USA
‡Department of Computer Science, Hubei University for Nationalities, Enshi, China

Abstract—The dramatic development of cloud storage services has led growing companies and individuals to outsource their data to cloud. However, users still concern about the availability and integrity of the data stored in cloud. To relieve these concerns, data redundancy is introduced into cloud storage systems, and data integrity verification schemes are used to check whether data is corrupted. Once data corruption is detected, the repair operations should be executed. However, most of the existing schemes based on erasure codes or network coding techniques either introduce high computation cost or cannot efficiently support remote data repairing. In this paper, we propose an efficient Remote Data Checking and Repairing (RDCR) scheme based on the minimum bandwidth regenerating codes. Our scheme reduces data owners’ burden of checking data integrity by enabling a third party to perform the public integrity verification. In addition, unlike previous schemes, our scheme supports exact repair of corrupted data so that the computation cost is further reduced. We implement our scheme and the experiment results show that, compared with the existing schemes, RDCR has lower computational overhead and communication cost.

Index Terms—Cloud storage, regenerating codes, data availability, data integrity

I. INTRODUCTION

Cloud storage provides users an on-demand data outsourcing service model and its many advantages attract more and more enterprises and individuals to move their data from local to remote cloud servers [1]. Currently, many companies, such as Amazon, Google and Microsoft, provide their own storage services. Users can upload their files to cloud and access them any time and anywhere without the burden of data management, and can save the expenditure of data maintenance. Although cloud storage introduces the appealing benefits, it also induces various threats towards users’ outsourced files.

Since users’ files are stored on third-party servers, users lose the control of their data. On one hand, the malicious servers may manipulate or delete rarely accessed users’ data for their own benefits. On the other hand, in an open cloud environment, the storage servers are vulnerable to external threats. For example, recently, more and more major cloud service providers have suffered from service outage or data corruption, which are caused by malicious attacks [2], [3]. Therefore, users should be able to detect the data corruption events in cloud servers in time.

To check whether the files stored in cloud are the same as original, proof of retrievability (POR) [4] and proof of data possession (PDP) [5] are introduced which can check the integrity of remote file without downloading it. Then, a series of extension schemes [6], [7], [8] based on PDP and POR have been proposed. However, few of them focus on how to perform efficient repair after detecting data corruption.

Since the above described PORs and PDPs are proposed for a single server, all of them are vulnerable to the single point of failure. Even if users detect the data corruption events, the single server cannot recover the data or must pay the heavy storage cost. Consequently, a multi-server model is considered by researchers. Through this way, once one server occurs data corruption, we can repair the data with the help of the remaining healthy servers. MR-PDP [9] first extended data integrity checking to a multi-server environment by replication, which is a common and simple way to enhance the reliability of data on untrusted storage systems. However, the storage cost and the repair bandwidth are both high, it means that multiple copies of the data are stored at different storage servers, and repair of data on corrupted servers is simple by reading the entire data from a healthy server. The amount of data being read from healthy servers during repair is what we refer to as repair bandwidth. Unlike MR-PDP, HAIL [10] striped data across servers via erasure codes which has less storage overhead and higher reliability with the same redundancy than the simple replication [11], [12]. However, the repair bandwidth of erasure codes is as high as that of replication.

Regenerating codes [13], [14] have been recently proposed to minimize repair bandwidth in a distributed storage system, which can achieve the optimal tradeoff between storage and repair bandwidth. There are a few studies [15], [16], [17] focusing on data integrity check based on regenerating codes in cloud storage, which not only protect data integrity but also save repair bandwidth compared with traditional erasure codes. However, these schemes perform private verification, which means that only the owner of the stored files can check the integrity. As a result, it requires that the data owner must be online. Moreover, they adopt so-called functional repair, in other word, the new regenerated data of the corrupted server are not the same as original as long as the repaired system maintains the same functions as the old one. Another kind of repair model is exact repair [11], [18], in which the corrupted data is exactly regenerated as it is previous stored,
it brings less changes to the system than functional repair. Since under functional repair, the newly regenerated data is different in each repair, there is a need to update the repairing and decoding rules whenever a failure occurs [11], which introduces the extra computational overhead compared with exact repair.

Inspired by the above observations, the previous schemes, which simultaneously realize remote data checking and repairing, are not efficient sufficiently in aspects of computation cost or repair bandwidth, and they are less of practicality. Therefore, it remains a challenge to protect data integrity and availability with the minimum cost.

In this paper, we propose a novel scheme of remote data checking and repairing (RDCR) for regenerating-codes-based cloud storage. We focus on the situation that once a corrupted server is detected and how we repair the data on it efficiently in a multi-server setting. Specifically, our major contributions can be summarized as follows:

1) We propose a novel scheme called RDCR to minimize the combined costs of remote data checking and repairing. Our proposed scheme provides a better overall efficiency than existing schemes by enabling public verification and exact repair.

2) We design our coding scheme based on minimum bandwidth regenerating (MBR) codes, which can achieve the minimum repair bandwidth. Moreover, unlike existing works, our scheme performs exact repair, which can efficiently recover the exact original form of any corrupted data and further reduce the repair cost.

3) We implement our scheme and evaluate its performance from several aspects through theoretical analysis and experiments. And the results demonstrate that our scheme is more efficient compared with its counterparts, it takes less time to repair a corrupted server and has lower communication for both data checking and repairing.

The rest of this paper is organized as follows. We briefly review the related work in Section II. In section III we state the problem that our scheme will solve. The detailed design of our scheme could be found in section IV. Section V gives the security analysis of our scheme. Experiments and evaluations are conducted in section VI. Section VII concludes the paper.

II. RELATED WORK

Regenerating codes for distributed storage systems are first proposed by Dimakis et al. [13]. Regenerating codes are built on the concept of network coding [19] and have advantages in reducing repair bandwidth over traditional erasure codes. Dimakis et al. [13] have summarized two kinds of regenerating codes in theory: minimum storage regenerating (MSR) codes and minimum bandwidth regenerating (MBR) codes. Subsequent works [20], [21], [18], [22] design different coding schemes of regenerating codes to satisfy the requirements of MSR or MBR codes in theory, and focus on how to perform repair to reduce the repair bandwidth. Yuchong Hu et al. [20] and Henry et al. [21] implement a kind of MSR codes, they perform functional repair and adopt random network coding during repair, which demands a large computational field to guarantee successful repair. Anyu Wang et al. [18] and Jun Li et al. [22] focus on the repair of more than one corrupted servers cooperatively, while our work concentrates on repairing one server.

On the other hand, the concepts of proof of retrievability (POR) [4] and proof of data possession (PDP) [5] are introduced to remotely check data integrity without downloading it. The basic POR scheme [4] embeds a set of pseudorandom blocks into an encrypted file stored on the server, and clients check the pseudorandom blocks to judge whether the file is intact. However, the number of checks that the client can issue is limited. PDP [5] allows the client to keep a small amount of metadata to check data integrity later on, which increases the storage overhead at client side. Several follow-up studies [6], [23], [24], [8] on POR and PDP are proposed. They all consider how to further reduce the cost of computation and communication during verification. However, none of them consider the efficient repair after the data corruption is detected, and they are designed for a single server setting.

There are a few works focusing on data integrity checking and repair in a multi-server setting. Reza Curtmola et al. [9] propose MR-PDP and Kevin D. Bowers et al. [10] propose HAIL, they extend data integrity checking to a multi-server setting using data replication and erasure codes respectively, but their schemes both have high repair bandwidth compared to regenerating codes. In [5], Chen et al. first apply data integrity checking schemes to regenerating-codes-based distributed storage. And since then related works appear (e.g., [25], [16], [17]). Chen et al. [5] adopt the private verification scheme that is adapted from [6], while our scheme is based on BLS signature and supports public data verification, that is to say, we can allow a third party to check data integrity for cloud users and greatly reduce users’ burden of being online. In [25] and [16], once data corruption is detected, their schemes perform functional repair, they employ random network coding to repair a failed server, though this is good for a dynamic system (i.e., the number of storage servers is not fixed), functional repair also increases the computational complexity due to the dynamics of repairing-and-decoding rules. Different from functional repair, our scheme adopts exact repair, which exactly regenerates the corrupted data. The data integrity checking scheme in [25] is adapted from the method of preventing pollution attacks in network coding. The scheme FMSR-DIP proposed in [17] is built on the minimum storage regenerating (MSR) codes, while ours is based on MBR codes, FMSR-DIP reduces storage cost in cloud and also considers the repair of one failed server. However, it adopts functional repair, it proposes a two-phase checking method to guarantee that system still holds its functionality even after iterative repairs, which is complex and time consuming. In addition, when checking the data integrity, FMSR-DIP needs to download the coded blocks from cloud, and then verifies the data, which deviates from the standard of POR and PDP. Accordingly, most of the above work cannot support efficient data checking and repairing simultaneously in cloud storage.
III. PROBLEM STATEMENT

A. System Model

Fig. 1 illustrates the system model of our scheme. We consider a cloud storage service that involves three entities: a user, storage servers, which make up cloud storage, and a third party auditor (TPA). The user uploads the encoded data blocks of the file $M$ with their corresponding verification tags to $n$ cloud storage servers. The user can select any $k$ out of the $n$ storage servers to download the encoded blocks, and recover the file $M$. This is because the regenerating code that we adopt satisfies $(n,k)$ MDS (Maximum Distance Separable) property (i.e. any $k$ out of $n$ suffice to recover the original whole data). The user resorts to a TPA to check the integrity of the data stored on each server. The TPA audits a storage server via issuing challenges to it, and to make sure whether the server correctly stores its assigned coded blocks. Upon receiving the challenges, the storage server generates a proof of data integrity and sends it to the TPA. If corruption is detected, then the auditor reports to the user and triggers a repair phase to repair corrupted data.

B. Threat Model

The cloud storage server is regarded as “curious-and-vulnerable”. Specifically, the storage server can behave properly and does not depart from the prescribed protocol. However, in order to cover some accidental data corruptions, it will try its best to prove that it is still holding the intact data. For its own benefits and reputation, the cloud has strong incentives to cheat the user or auditor. In addition to the untrusted storage server, the server itself is susceptible to external adversaries’ attacks. The adversaries damage the storage server, while the corruption may not be detected in time.

In summary, the threat model includes malicious storage servers who wants to hide data corruption, and an external attacker who maliciously damages the storage servers. We assume that the TPA is fully trusted and will not collude with the cloud servers or external attackers. We also assume that an external adversary $A$ is mobile, i.e., can corrupt different servers in any given time interval, we call such a time interval as an epoch. If the adversary $A$ controls all the $n$ servers, it is impossible to recover the original files. Thus we assume that $A$ controls at most $n - k$ servers in any given epoch. The goals of RDCR are to timely check the misbehaviors of cloud servers and the corruptions due to external attackers before they render a file $M$ unrecoverable.

C. Notations

We define some notations in Table I.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>M</td>
</tr>
<tr>
<td>$n$</td>
<td>the number of storage servers</td>
</tr>
<tr>
<td>$k$</td>
<td>the data stored in $k$ out of $n$ storage servers suffice to recover the original data</td>
</tr>
<tr>
<td>$d$</td>
<td>the number of servers that participate in repairing a corrupted server</td>
</tr>
<tr>
<td>$S_l$</td>
<td>the $l$-th storage server</td>
</tr>
<tr>
<td>$C_{ij}$</td>
<td>the $j$-th coded block that stored on the $i$-th storage server</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>the number of coded blocks that each storage server stores</td>
</tr>
<tr>
<td>$\beta$</td>
<td>the number of downloaded coded blocks from each helping server during repair</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>the total repair bandwidth</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>a security parameter</td>
</tr>
<tr>
<td>$Z_p$</td>
<td>a group of the prime order $p$</td>
</tr>
<tr>
<td>$G$</td>
<td>a cyclic multiplicative group</td>
</tr>
</tbody>
</table>

D. Preliminaries

Our coding technique is based on a kind of regenerating codes [26], and our data integrity checking technique is adapted from the BLS signature [27]. We now introduce some necessary regenerating coding and cryptographic background for our proposed scheme.

**MBR Codes.** Dimakis et al. [13] theoretically characterize MBR codes using flow arguments on an appropriately constructed graph. Recall that MBR codes are a kind of regenerating codes, and also satisfy $(n,k)$ MDS property. The MBR codes can achieve the minimum repair bandwidth, which are characterized by the following equation:

$$ (\alpha_{MBR}, \gamma_{MBR}) = \left( \frac{2|M|d}{2kd - k^2 + k}, \frac{2|M|d}{2kd - k^2 + k} \right) $$

(1)

Note that the minimum bandwidth regenerating codes, the storage size $\alpha$ is equal to $\gamma$, the total repair bandwidth. It means that MBR codes incur no repair bandwidth expansion at all, just like a replication system does, downloading exactly the amount of information stored during a repair.

**Bilinear Map.** Assuming that $G$ and $G_T$ are both cyclic multiplicative groups of prime order $p$, $g$ is the generator of $G$. If the map $e : G \times G \rightarrow G_T$ is a bilinear map, then it satisfies the three properties: 1) Computability: there exists
an effective algorithm of polynomial time to calculate \(e\); 2) Bilinearity: for all \(a, b \in G\) and \(x, y \in \mathbb{Z}_p\) (\(\mathbb{Z}_p\) is a group with the prime order \(p\)), \(e(a^x, b^y) = e(a, b)^{xy}\); 3) Nondegeneracy : \(e(g, g) \neq 1\), where \(g\) is the generator of \(G\).

IV. OUR PROPOSED SCHEME: RDCR

A. Definitions and Framework of RDCR

Definition 1 (Remote Data Checking and Repair): A remote data checking and repair scheme RDCR = (KeyGen, Encode, TagGen, Challenge, Respond, Verify, Repair) is a collection of six algorithms (KeyGen, Encode, TagGen, Challenge, Respond, Verify) and an interactive protocol Repair as follows:

- \((pk, sk) \leftarrow \text{KeyGen}(1^\lambda)\). This probabilistic algorithm is run by a user. It takes the security parameter \(1^\lambda\) as input, and generates public key \(pk\) and private key \(sk\).
- \(\{C_{ij}\} \leftarrow \text{Encode}(n, k, M)\). This algorithm is run by a user, and encodes the original file \(M\) as a set of file blocks, where the \(C_{ij}\) is the \(j\)-th coded block designated for the server \(i\). The encoding is designed to provide \(k\)-out-of-\(n\) redundancy across servers and satisfies \((n, k)\) MDS property.
- \(\{\tau_{ij}\} \leftarrow \text{TagGen}(C_{ij}, sk)\). This algorithm takes the coded blocks \(C_{ij}\) and private key \(sk\) as input, and outputs the corresponding verification tag \(\tau_{ij}\) of the coded block \(C_{ij}\).
- \(\Phi \leftarrow \text{Challenge}(\cdot)\). This algorithm takes no input, and outputs the challenge set \(\Phi\).
- \(R \leftarrow \text{Respond}(\Phi, C_{ij}, \tau_{ij})\). This algorithm takes as input the challenge \(\Phi\), the coded blocks \(C_{ij}\), and their corresponding verification tags \(\tau_{ij}\). It then returns the response \(R\).
- \((1, 0) \leftarrow \text{Verify}(R, \Phi, pk)\). This algorithm is run by a verifier upon receipt of the response \(R\). It takes as input the public key \(pk\), the challenge \(\Phi\), and the response \(R\). It outputs 1 if the integrity of the data is verified as correct, and 0 otherwise.
- \((1, 0) \leftarrow \text{Repair}(U, S_l)\). This repair protocol is run by a user \(U\) and a server \(S_l\), where \(l \in [1, n]\). It returns 1 if the repair of corrupted server is successful, and 0 otherwise.

Our proposed scheme can be defined as Definition 1 and consists of three phases:

Preprocess phase: For an original data file, the user first generates the key pair \((pk, sk)\) by running KeyGen, and then runs Encode and TagGen respectively, to generate the encoded data blocks and their verification tags. Lastly, the encoded blocks and their verification tags are uploaded to the cloud and the user can delete them from the local storage.

Check phase: This is an interactive phase between a storage server and a TPA. The TPA checks the integrity of data stored at each server periodically. It first issues a challenge which generated by executing Challenge algorithm to a storage server. The storage server runs Respond to response to the challenge, and sends the response back to TPA. Finally, with the response it receiving, the TPA runs Verify algorithm to check the integrity of data it challenged before and reports the check result to user.

Repair phase: If a corrupted storage server is detected by TPA in the check phase, then repair phase starts. The user and other remaining healthy storage servers cooperatively to repair corrupted data by running Repair protocol.

B. Coding Construction of RDCR

In this section, we illustrate the idea of MBR codes based complete graph. By the property of complete graph, the repair of a failed server needn’t involve encoding and decoding operations. As the repair bandwidth \(\gamma = d\beta\), from Eq. 1, we get

\[
\beta = \frac{2|M|}{2kd - k^2 + k}
\]  \(\text{(2)}\)

Clearly for a feasible system we need \(\beta\) to be an integer. Assume \(\beta\) to be the smallest possible positive integer, i.e. \(\beta = 1\). From Eq. 1 and Eq. 2, then we have

\[
|M| = \frac{kd - k(k - 1)}{2}
\]  \(\text{(3)}\)

and

\[
\alpha = d
\]  \(\text{(4)}\)

For any larger file size, we can divide the source file into blocks of size \(|M|\), each of which can be separately solved using the construction of \(\beta = 1\). The repair bandwidth are \(\gamma\) data blocks, where \(\gamma = d\beta = d = \alpha\), that is to say, through our coding construction, our repair bandwidth is equal to storage size.

Fig. 2 illustrates the basic idea of the coding construction through the example of \((n, k, d) = (5, 3, 4)\). Thus, according to Eq.2, Eq.3 and Eq.4, we can get \(\alpha = 4\), \(\gamma = d\beta = 4\), \(|M| = 9\). We store the original file across 5 servers redundantly, and the coding process is as follows: we first divide the original file into 9 blocks, and encode them into 10 blocks with (10, 9)-MDS code [26]. The coded blocks is denoted by \(Ci\) where \(i = 1, 2, \ldots, 10\). These ten coded blocks correspond to the ten edges of a complete graph with five vertices. Each vertex stands for a storage server, which stores the coded blocks corresponding to the four edges it connects. Thus from Fig. 2, we can find that any two storage servers have exactly one overlapping coded block. Note that the number of coded
Let \( e : G \times G \rightarrow G_F \) be a bilinear map, \( g \) is the generator of \( G \), and \( H : \{0,1\}^* \rightarrow G \) is a BLS hash function.

- \((pk, sk) \leftarrow \text{KeyGen}(1^\lambda)\):
  - Generate a random number \( \eta \leftarrow Z_p \) and compute \( v \leftarrow g^\eta \). Then choose \( s \) random elements \( u_1, u_2, \ldots, u_s \leftarrow G \). Set \( pk = (v, u_1, u_2, \ldots, u_s) \), \( sk = (\eta) \).
- \( \{C_{ij}\} \leftarrow \text{Encode}(n, k, M) \):
  1. Divide the file \( M \) into \( k' \) equal-size native blocks, where \( k' = \lceil |M|/kd \rceil \) and \( d = n - 1 \).
  2. Encode these \( k' \) original blocks using \((n', k')\) MDS code into \( n' \) encoded blocks, where \( n' = \frac{n(n-1)}{2} \).
  3. Assign these \( n' \) coded blocks across the \( n \) servers according to the MBR code based complete graph. The code blocks are then evenly stored in the \( n \) storage servers, each having \( \alpha \) chunks, where \( \alpha = d = n-1 \).
- \( \{\tau_{ij}\} \leftarrow \text{TagGen}(C_{ij}, sk) \):
  1. For each coded block \( C_{ij} \) stored on every server, split it into \( s \) sectors, denoted by \( C_{ijt}, t \in [1, s] \).
  2. For each coded block \( C_{ij}, i \in [1, n] \) and \( j \in [1, \alpha] \), compute
    \[
    \tau_{ij} \leftarrow H(i|j), \prod_{t=1}^{s} u_{C_{ijt}}^{a_{ijt}})^\eta.
    \]
  3. Upload the processed data \( C_{ij} \) together with its verification tag \( \tau_{ij} \) to the corresponding servers.

![Fig. 3. Preprocess phase](image)

blocks is determined by the number of edges of a complete graph whose vertices represent the storage servers.

In this way, the system satisfies \((5, 3)\) MDS property, we can recover the original file through downloading coded blocks from any 3 out of 5 storage servers. If one server is corrupted, since each of the existing \( n - 1 \) servers contains one distinct coded block stored on the failed server, exact repair of the failed server is possible by downloading one block each from the remaining \( n - 1 \) healthy servers.

C. Three Phases of RDCR

1) Preprocess Phase: In this phase, a user preprocesses his file \( M \) before upload it to \( n \) cloud storage servers, which mainly involves three algorithms: KeyGen, Encode and TagGen. These algorithms are shown in Fig. 3.

In Fig. 3, we encode the original file with the MBR code based complete graph, the coding idea has been given in last subsection. In addition, note that the verification tags of coded blocks are generated by the user himself, and only the file owner can produce the tags with his private key \( sk \), which guarantees the uniqueness and security of verification tags. At the end of this phase, each server will store \( \alpha \) coded blocks together with their corresponding verification tags. Besides, the cloud storage system which is composed of the \( n \) servers satisfies \((n, k)\) MDS property.

2) Check Phase: After a user outsources the processed file and corresponding verification tags from local storage to cloud, the user delegates a third party auditor(TPA) to remotely check data integrity for him. TPA is responsible for the integrity of data stored on each storage server. The check phase involves two entities: the server and TPA, and three algorithms: Challenge, Respond and Verify. The check process is shown in Fig. 4.

In the check phase, the TPA randomly chooses \( \alpha \) numbers \( a_1, a_2, \ldots, a_{\alpha} \), of which \( a_i \) corresponds to the \( i \)-th encoded block in every storage server. Thus, we can check all data stored on every server and timely detect which server and its data are corrupted. Through Eq. 5, TPA can remotely verify data integrity, if the verification fails, the TPA reports it to the user and triggers the repair phase. Furthermore, in check phase, the public key \( v \) and \( u_1, \ldots, u_s \) are sufficient for the verifier to check data integrity, while the private key \( \eta \) is only required for generating the verification tags \{\( \tau_{ij} \)\}. That’s the reason why our scheme is publicly verifiable, and the verification correctness in Eq. 5 is proved as follows:

\[
e(\tau_{ij}, g) = e\left(\prod_{j=1}^{\alpha} a_{ij}^\eta g\right)
= e\left(\prod_{j=1}^{\alpha} H(i|j)^{a_{ij}}, \prod_{t=1}^{s} u_{C_{ijt}}^{a_{ijt}} g\right)^\eta
= e\left(\prod_{j=1}^{\alpha} H(i|j)^{a_{ij}}, \prod_{t=1}^{s} u_{C_{ijt}}^{a_{ijt}}, v\right)
= e\left(\prod_{j=1}^{\alpha} H(i|j)^{a_{ij}}, \sum_{j=1}^{\alpha} a_{ij} C_{ijt}^{t}, v\right)
= e\left(\prod_{j=1}^{\alpha} H(i|j)^{a_{ij}}, \prod_{t=1}^{s} u_{C_{ijt}}^{a_{ijt}}, v\right).
\]

![Fig. 4. Check phase](image)

2) Return 1 if the verification succeeds and 0 otherwise.

3) Repair Phase: The repair phase is completed by healthy servers and the user cooperatively. We consider the situation...
\{1, 0\} \leftarrow \text{Repair}(U, S_l);
1) TPA checks the integrity of \(\alpha\) data blocks stored on the remaining healthy servers but the corrupted server \(S_l\), where \(l \in [1, n]\). If the verification fails, then outputs 0 and exits.
2) Each of the \(n - 1\) servers transfers \(\beta\) coded blocks required to help repair corrupted data, where \(\beta = 1\).
3) The user \(U\) downloads the newly generated blocks from server \(S_l\) and computes their verification tags \(\{\tau_{lj}\}\), where \(j \in [1, \alpha]\).
4) The user \(U\) uploads the verification tags \(\{\tau_{lj}\}\) to the repaired server \(S_l\). If all the steps are successful, outputs 1 and 0 otherwise.

Fig. 5. Repair phase

that one server fails and we perform exact repair of its stored data. We also take the Fig. 2 as an example, suppose server 1 fails, then server 2 gives C1, server 3 gives C2, server 4 gives C3, and server 5 gives C4. All these four coded data blocks are transferred to the new server 1. This is the so-called “repair-by-transfer” property, which is very simple for exact repair. Thus, the repaired server 1 stores exactly the same data blocks as previously stored. However, we only exactly produce the required data blocks, their corresponding verification tags have not been generated. Then the user downloads the generated data blocks and generates the corresponding verification tags at last, the new generated tags are uploaded to the server 1. In this way, we can easily generalize the repair operation for \(n\) storage servers. If server \(S_l\) (1 \(\leq\) \(l\) \(\leq\) \(n\)) needs to be repaired, the detailed procedures are shown in Fig. 5. The step 1 is to make sure that the other servers are undamaged, such that the coded data block from each of them is correct and the corrupted data blocks are exactly repaired.

Note that our repair scheme doesn’t involve any encoding and decoding operations compared to other schemes [25], [28], [21], and the repair bandwidth is exactly the same as the storage size, which is more efficient and faster for a repair.

V. Security Analysis

In this section, we analyze the security of our scheme from two aspects: data integrity and data availability.

A. Data Integrity

Since the cloud server is untrusted and for its own reputation, it tends to hide data corruption and cheat users that it stores intact data. To carry out cheat, the server will send a forged response back to the verifier. If the forged response can pass the checking of verifier, then we say the check scheme is not secure for data integrity. Next, we will show that our scheme is secure in the random oracle model, and the malicious server can never pass the verification by responding with forged values.

Note that our scheme of checking data integrity is based on BLS signatures, from [29] and [30], we have the following definition and theorem:

**Definition 2:** The computational Diffie-Hellman problem is hard, which means that for any given \(g, g^x, g^y \in G\) and \(x, y \in \mathbb{Z}_p\), if \(x\) and \(y\) are both unknown, then computing \(g^{xy}\) is hard.

**Theorem 1:** If the signature scheme used for generating file tags is unforgeable and the computational Diffie-Hellman problem is hard in bilinear groups, then, in the random oracle model, except with negligible probability no adversary can pass the verification with forged values, except by responding with values \(\{x, y\}\) that are correctly computed.

We omit the proof of this theorem here, which is similar to [30]. Since only the file owners can generate the file tags with private keys, it is difficult for adversaries to forge the tags without private keys and pass the verification. Furthermore, in our TagGen algorithm, the verification tag \(\tau_{ij}\) binds the storage server id \(i\) and the coded block id \(j\), in this way, we can prevent storage servers from colluding with each other and use another tag of the same data block stored on the another server during verification.

Accordingly, any corruption of data will be detected in our scheme with overwhelming probability, and our scheme can protect data integrity.

B. Data Availability

Recall that in Section III-B, in addition to the malicious storage servers, there is an external attacker \(A\) who can corrupt data in at most \(n - k\) out of the \(n\) servers in any epoch, and after several epochs they could render the original data unrecoverable. However, our scheme can guarantee the availability of data stored in cloud. We state the reasons as follows:

Firstly, we have prevention mechanism. The TPA checks the integrity of data stored on each server in every epoch. Secondly, we have remedy mechanism. If a faulty server is detected in an epoch, our scheme employs the redundancy at the remaining healthy servers to repair data at faulty servers (refer to Section IV-C3). What’s more, even if the number of corrupted servers is up to \(n - k\) in an epoch, note that in our preprocess phase, we encode the original data with MBR code and store the coded data across \(n\) servers redundantly, and the storage system satisfies \((n, k)\) MDS property. Thus, at the end of this epoch, there are remaining at least \(k\) healthy servers, and the original file can be recovered from any \(k\) out of the \(n\) servers with high probability.

In a word, our scheme is secure for both data integrity and availability against malicious servers and external attackers.

VI. Experimental Evaluation

In this section, we evaluate the computation and communication cost of our RDCR scheme compared with its counterparts: RDC-NC [15] and FMSR-DIP [17].

We implement the construction of RDCR, RDC-NC and FMSR-DIP, and conduct simulated experiments respectively on the same condition. The implementation uses OpenSSL version 1.0.0 and C++ language. We set the same desired reliability level, where \(n = 12, k = 3\), and in our scheme we set \(\alpha = n - 1 = 11, \beta = 1\). All experiments are conducted
on a machine equipped with Intel 2.8 GHz CPU, 16GB RAM, and 32-bit Ubuntu 11.04. All results are averaged over 20 runs.

We measure the running time of the three phases: preprocess, check and repair, and the communication cost of repair phase. A user preprocesses his file before outsources it. Specifically, in the preprocess phase, the user mainly encodes original file and generates corresponding verification tags, we record the total time of these operations. Fig. 6 depicts the preprocess time for different size of file under three schemes. From it, we can clearly find that the computation cost increases with file size. For fixed file size, the preprocess time of RDCR is much less compared with that of RDC-NC and FMSR-DIP in the preprocess phase. This is because our scheme employs the MBR code based complete graph, it generates \( n(n-1)/2 \) coded blocks (see the section IV-C1), while RDC-NC and FMSR-DIP both use random network coding to generate \( n_\alpha = n(n-1) \) coded blocks, which are two times the coded blocks of our scheme. In the case \( n = 12 \), our scheme generates 66 coded blocks, while RDC-NC and FMSR-DIP both generate 132 coded blocks. For the same block size, the preprocess time increases with the number of coded blocks.

To evaluate the performance of check phase, we focus on the time cost for checking the all data stored on one server.

Fig. 7 shows how the overall check time changes with file size. We can see that our scheme is more efficient in checking data integrity than the other two schemes. In the check phase, the auditor sends \( \alpha \) random numbers to the server, which corresponds to the \( \alpha \) data blocks stored on the server, so the computation cost is in direct proportion to the file size and increases with file size.

While in the FMSR-DIP scheme, it checks data integrity by downloading all the data from cloud to the local and verifies it. Obviously, its computation and communication cost are much higher than our scheme.

We assume that the user needs to repair one failed server. Fig. 8 and Fig. 9 demonstrate the communication cost and time cost for repairing one failed server respectively. Since RDC-NC and RDCR are both based on MBR codes, the communication cost of RDC-NC is the same as that of RDCR, and they both achieve the minimum repair bandwidth. Note that the communication cost during repair is the same as the repair bandwidth. Consequently, there are two lines overlapping in Fig. 8, and the position is lower than the line that represents FMSR-DIP. Fig. 9 shows the required time to repair one failed server for the three schemes. We can find that our scheme repairs a corrupted server more quickly than the other two schemes. The reason lies in that our scheme possesses
the “repair-by-transfer” property as we described in Section IV-C3, it doesn’t involve any computation, while the other two schemes perform functional repair via random network coding, which introduces the extra computational cost and increases the repair complexity.

VII. CONCLUSION

In this paper, we are devoted to protect data integrity and availability in cloud storage. We propose a novel and efficient scheme for remote data checking and repairing in cloud storage. Our scheme provides a better performance than existing schemes in terms of computation and communication cost. By allowing a third party auditor to check data integrity, our scheme greatly reduces users’ burden of being online. Our coding scheme is based on minimum bandwidth regenerating codes, which achieves the minimum repair bandwidth during repair. Moreover, unlike existing works, our scheme performs exact repair whenever data corruption occurs. It can exactly recover the original form of the data and further reduce the repair cost, and the repair operation is straightforward by simply transferring data blocks. The performance analysis and experimental results show that our scheme has less communication and computation cost for both data checking and repairing compared with existing schemes. In conclusion, our scheme is more practical in remote data checking and repairing for cloud storage.

ACKNOWLEDGEMENT

This research was supported in part by the National Natural Science Foundation of China under Grant No. 61272451, 61173154, 61173175, 61472121.

REFERENCES


