Data Integrity Protection scheme for Minimum Storage Regenerating Codes in Multiple Cloud Environment

Saravana M K, Harish K

Abstract— **The demand for large scale data storage has increased extremely, with applications that require storage access and security for massive volume of data. When the deployed cloud storage is individually unstable, as in the case of modern data centers, redundancy must be introduced into the system to improve stability against failures. To provide fault tolerance for cloud storage, one approach is to stripe the data across multiple cloud storage providers. Permanent failure of Clouds are occasional but occurs, a common practice is to reconstruct the lost data by using the data on the residual clouds. Storing the data using an erasure code, in fragments spread across clouds, requires less redundancy than simple replication for the same level of reliability. The high repair cost of Maximum Distance Separable (MDS) erasure codes has recently motivated a new class of code, called Minimum Storage Regenerating codes (MSR), which can significantly reduce repair bandwidth over conventional MDS codes. We study the problem of remotely checking the integrity of regenerating-coded data against corruptions under a commercial cloud storage setting. We design and implement a practical data integrity protection (DIP) scheme for a specific regenerating code, while preserving the intrinsic properties of fault tolerance and repair traffic saving. Our DIP scheme enables a client to feasibly verify the integrity of random subsets of outsourced data against general or malicious corruptions. Further We demonstrate that remote integrity checking can be feasibly integrated into regenerating codes in practical cloud deployment.**

Index Terms— **Data Integrity, Fault tolerance, Regenerating codes, Remote Integrity Protection.**

I. INTRODUCTION

Cloud storage provides reliable and scalable solutions to the increasing demand of data storage. However, using a single-cloud storage provider raises concerns such as having a single point of failure [7]. As suggested in [1], [7], an optimal solution is to stripe data across multiple cloud storage providers, so that if one cloud storage provider is not available we can still able to retrieve the data from other available cloud storage provider. A traditional way to repair a failed node in the context of multiple cloud storage is to download data from the surviving clouds and reconstruct the whole file first, and then regenerate the lost part of the file (e.g., RAID-5, RAID-6) and write the reconstructed data on to the new cloud. Since the size of the original file may be large, a lot of data traffic should be migrated between the clouds, for reconstructing the lost part of the file. In order to reduce the total data traffic required for repairing, called *repair-traffic*, a new class of erasure codes, called *regenerating codes* [3], was proposed that has significantly reduced the traffic involved in regenerating a failed cloud.

In addition to repairing the lost data, it is also desirable to enable cloud clients to verify the integrity of their outsourced data in the cloud, in case their data has been accidentally corrupted or maliciously compromised by insider/outsider attacks. Regenerating codes [3] aims to minimize repair traffic (i.e., the amount of data being read from surviving servers). In essence, they achieve this by not reading and reconstructing the whole file during repair as in traditional erasure codes, but instead reading a set of chunks smaller than the original file from other surviving servers and reconstructing only the lost data chunks.

An open question is, *can we enable integrity checks atop regenerating codes, while preserving the repair traffic saving over traditional erasure codes?* A related approach is HAIL [10], which applies integrity protection for erasure codes. It constructs protection data on a per-file basis and distributes the protection data across different servers. To repair any lost protection data in the presence of a server failure, one needs to access the whole file, and this violates the design of regenerating codes. Thus, we need a different design of integrity checking tailored for regenerating codes. In this paper, we propose the design and implementation of a practical data integrity protection (DIP) scheme for regenerating-coding-based cloud storage. We augment the implementation of the functional minimum storage regenerating (FMSR) code [2] and construct F-DIP, a code that allows clients to remotely verify the integrity of random subsets of long-term archival data under a multi-server setting. FMSR-DIP aims to achieve several design features. First, it preserves fault tolerance and repair traffic saving as in FMSR [2]. We implement F-DIP, and evaluate its overhead over the existing FMSR implementation through extensive experiments in a cloud storage environment.

Manuscript received Dec, 2015.

Saravana M K*, Asst Professor, Department of Computer Science and Engineering, Jyothy Institute of Technology., Bangalore, India, 9964961674.,*

Harish K*, Asst Professor, Department of Computer Science and Engineering, Jyothy Institute of Technology Bangalore, India, +919980430008.*

We evaluate the running times of different basic operations, including upload, check, download, and repair, for different parameter choices of our DIP scheme. Our work demonstrates the feasibility of enabling integrity protection, fault tolerance, and efficient recovery for cloud storage.

The rest of the paper is organized as follows: In section II, we discuss literature survey on related topics, in section III, we discuss preliminaries on proposed methodology. In section IV we present, F-DIP and its implementation. In section V we discuss experiments and results, section VI concludes the paper.

.

II. Literature survey

"HAIL: A High-Availability and Integrity Layer for Cloud Storage,"[10], provides integrity and availability guarantees for stored data. HAIL is a remote-file integrity checking protocol that offers efficiency, security, and modeling improvements over straight forward multi-server application. HAIL, extends the basic principles of RAID into the adversarial setting of the Cloud. It ensures file availability against a strong, mobile adversary.

Kannan et al.'10 [5], showed that the common procedure to repair a single node failure in erasure coded system, is sub-optimal and introduces the notion of regenerating codes, which allow a new node to communicate functions of the stored data from the surviving nodes. Further, they showed that regenerating codes can significantly reduce the repair bandwidth, and there is a fundamental tradeoff between storage and repair bandwidth.

Dimakis et al.'10 [11], has proposed an excellent work on regenerating codes, the work is mainly based on providing overview of research results on regenerating codes and various repair models were described with their key constructive techniques, such as *functional repair, exact repair, exact repair of systematic parts* and also several open problems regarding these models are discussed. The work is done from theoretical perspective.

C. Suh et al.'11 [13], described *(n, k, d) Exact-Repair MDS* codes, which allow any failed node to be repaired exactly, with access to *d* survivor nodes, where $k \leq d \leq n\text{-}1$. The construction of Exact-Repair MDS codes that are optimal in repair bandwidth for the cases of: (a) $k/n \leq 1/2$ and $d \geq 2k\text{-}1^1$, (b) $k \leq 3$, the construction of code is based on interference alignment and they showed that cutset bounds are achievable for the case of $k = 3$.

Salim et al.'10 [6], introduced a new class of regenerating codes called Exact minimum bandwidth regenerating codes (MBR) codes for distributed storage system. The code consist of two components, the outer MDS code and inner repetition code, the inner code as a Fractional Repetition code, the model for repair is *table-based,* and thus differs from the random access model. Further, they present the construction of Fractional Repetition codes based on regular graphs and Steiner systems for a large set of system parameter.

III. Preliminaries

A. FMSR CODES

The existing work on *Minimum storage regenerating code,* called NCCloud[2] considers multiple cloud storage setting with N clouds (N = 4), where the file of size M byte is fragmented into four equal sized fragments called *native* chunks, each of size M/4. These *native* chunks are then encoded into eight *parity* chunks and equally distributed among the N nodes. On event of permanent failure of any one of the cloud, the residual clouds sends the data to help reconstructing the lost chunks of failed cloud. Since the fragments are encoded and stored, this approach relaxes encoding operations that has to be performed during the repair operation by the residual clouds. Hence this approach significantly reduces the repair traffic and repair cost. The details of the approach is here, on top of NCCloud, (proxy), a *Minimum storage regenerating code* called *functional Minimum storage regenerating code* (FMSR) was built. The FMSR code implementation is divided into three basic operations, each of them is explained below:

File upload

Step 1: Divide the file 'F' of size M byte into four equal sized *native* chunks, denoted by $F_1 \ldots\ldots\ldots F_4.$

Step 2: Encode these four native chunks into eight parity chunks, denoted by $P_1 \ldots\ldots\ldots\ldots P_8.$ Each P_i, where i=1……..8, is formed by a linear combination of four *native* chunks. Specifically, let EM = $[\alpha_{i,j}]$ be an 8×4, encoding matrices for some co-efficient αi,j in Galois Field (2^8). The encoding matrix looks like this,

$$EM = \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} \\ \alpha_{7,1} & \alpha_{7,2} & \alpha_{7,3} & \alpha_{7,4} \\ \alpha_{8,1} & \alpha_{8,2} & \alpha_{8,3} & \alpha_{8,4} \end{bmatrix}$$

Each row vector of EM is called Encoding co-efficient vector (ECV), which contains four elements. Let ECV_i denote the ith row vector of EM,

Ex: $ECV_1 = \alpha_{1,1}, \alpha_{1,2}, \alpha_{1,3}, \alpha_{1,4}.$

Step 3: Compute the *parity* chunks P_i, by the product of ECV_i and all the native chunks i. e, $F_1, F_2,$ F3, F4, as shown below:

$$P_i = \sum_{j=1}^{k(n-k)} \alpha_{i,j} F_j, \text{ where I = 1…..8. i. e,}$$

$$P_1 = \alpha_{1,1} F_1 + \alpha_{1,2} F_2 + \alpha_{1,3} F_3 + \alpha_{1,4} F_4.$$
$$P_2 = \alpha_{2,1} F_1 + \alpha_{2,2} F_2 + \alpha_{2,3} F_3 + \alpha_{2,4} F_4.$$

.
.
.

$$P_8 = \alpha_{8,1} F_1 + \alpha_{8,2} F_2 + \alpha_{8,3} F_3 + \alpha_{8,4} F_4.$$

Step 4: The *parity* chunks are then evenly stored in to the four clouds, each cloud will have two chunks.

Note: store the whole EM in a metadata object. After that replicate the metadata object to all clouds.

Definition 1: metadata object holds the file details and the coding information (ex: encoding co-efficient for FMSR code).

File Download

Fig. 3 shows file download operation.

Step 1: To download a file, first download the metadata object that contains the ECV's.

Step 2: Then, select any two out of four clouds.

Step 3: Download the four *parity* chunks from the two clouds.

Step 4: The ECV's of the four *parity* chunks can form 4×4 square matrix, if the MDS property is maintained, then by definition the inverse of the square matrix exists. Calculate the inverse.

Step 5: Multiply the inverse of the square matrix with each of the four parity chunks, and get four native chunks.

Iterative Repair

FMSR codes generate different chunks in each repair, one challenge is to ensure, that the MDS property still holds even after several rounds repairs. To do so, perform *Two-phase checking.*

Two-phase checking: consider the rth round of repair for permanent single cloud failure (where, $r \geq 1$). In the first-phase, check whether the new set of chunks in all clouds satisfies MDS property after the rth round of repair. In the second-phase, check if the MDS property is still satisfied after the (r + 1)th round of repair for any possible cloud failure, and this property is called *repairMDS property*.

Step 1: Download the EM from a surviving cloud.

EM specifies the ECV's for constructing all *parity* chunks via linear combinations of *native* chunks.

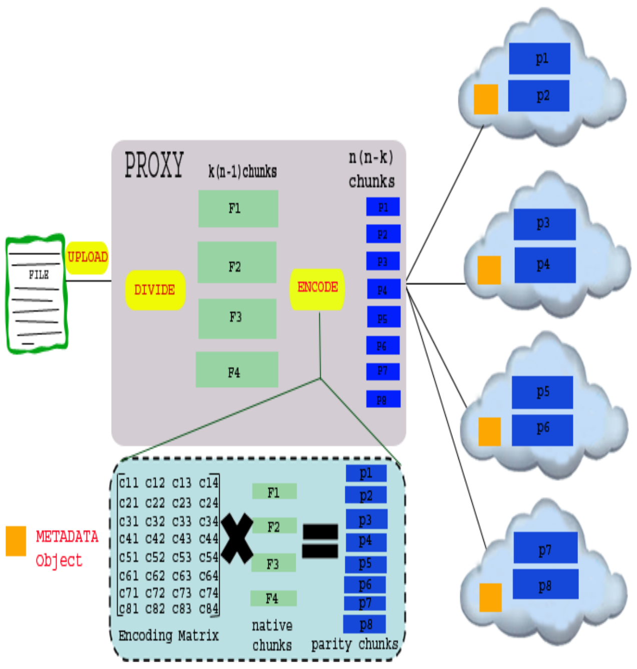

Fig. 1 File Upload Operation.

Step 3: Generate a Repair Matrix (R_m).

Construct an 2×3 *Repair matrix* $R_m = [\Upsilon_{i,j}]$, where each elements in $\Upsilon_{i,j}$ is randomly selected in GF(2^8).

$$R_m = \begin{bmatrix} \Upsilon_{1,1} & \Upsilon_{1,2} & \Upsilon_{1,3} \\ \Upsilon_{2,1} & \Upsilon_{2,2} & \Upsilon_{2,3} \end{bmatrix}$$

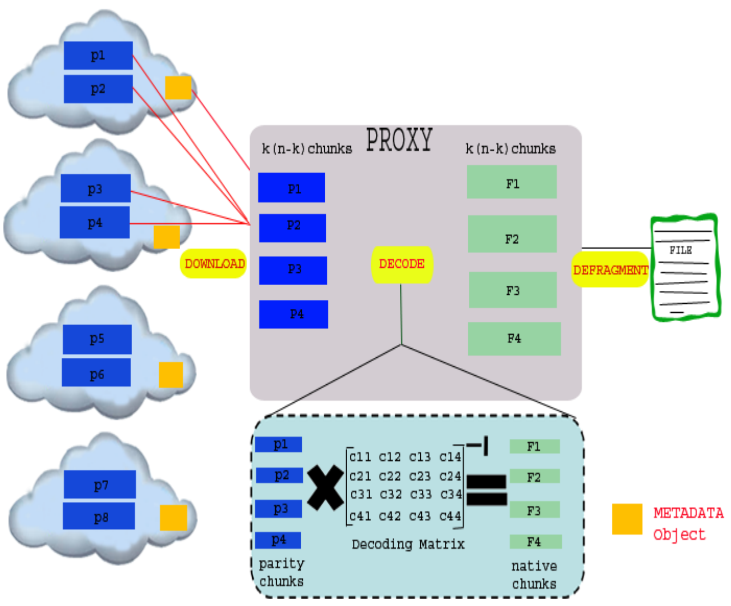

Fig. 2 File Download Operation.

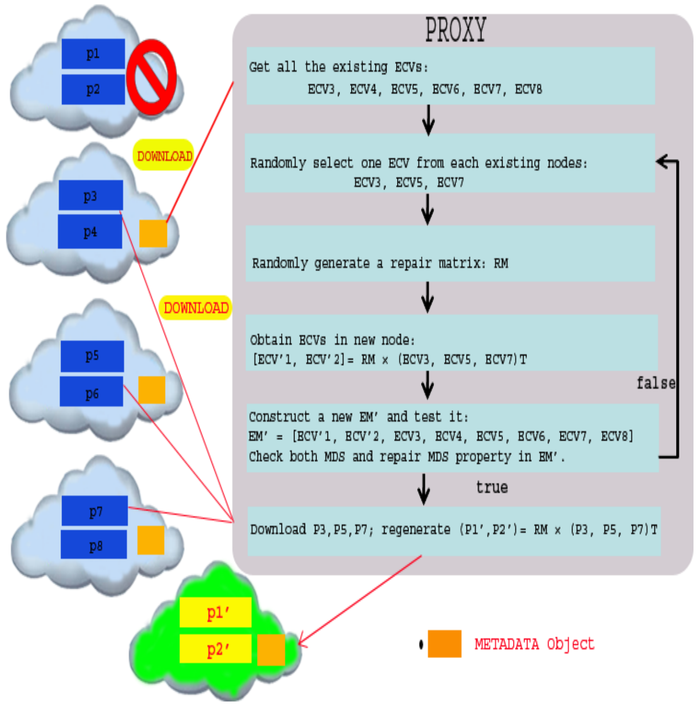
PROXY
p1
p2
p3
p4
p5
p6
p7
p8
DOWNLOAD
DOWNLOAD
Get all the existing ECVs:
ECV3, ECV4, ECV5, ECV6, ECV7, ECV8
Randomly select one ECV from each existing nodes:
ECV3, ECV5, ECV7
Randomly generate a repair matrix: RM
Obtain ECVs in new node:
[ECV'1, ECV'2]= RM × (ECV3, ECV5, ECV7)T
false
Construct a new EM' and test it:
EM' = [ECV'1, ECV'2, ECV3, ECV4, ECV5, ECV6, ECV7, ECV8]
Check both MDS and repair MDS property in EM'.
true
Download P3,P5,P7; regenerate (P1',P2')= RM × (P3, P5, P7)T
p1'
p2'
METADATA Object

Fig. 3 Repair Operation

Step 4: compute the ECV's for the new code chunks and reproduce a new encoding matrix ($\overline{EM}$).

Multiply RM with ECV's selected in step 2 to construct two new *parity* chunks.

ECV'$_i$=$\sum_{j=1}^{n-1} \Upsilon_{i,j} ECV_{i_j}$, for i=1,….n-k.

Ex: ECV'$_1$= $\Upsilon_{1,1} ECV_{i_1} + \Upsilon_{1,2} ECV_{i_2} + \Upsilon_{1,3} ECV_{i_3}$.

Now reproduce a encoding matrix i.e., change EM to EM' by substituting the ECV's of failed cloud in EM with the corresponding ECV'$_i$.

Step 5: Given EM', check if both MDS and rMDS properties are satisfied.

Intuitively, verify the MDS property by enumerating all $\binom{n}{k}$ subsets of k clouds to see if each of their corresponding encoding matrices forms a full rank. For rMDS property, check that for any possible cloud failure (1 out of n), it is possible to collect one out of two chunks from each of the other three surviving clouds such that MDS property is maintained.

The total number of checks performed for the rMDS property is at most $n(n-k)^{n-1}\binom{n}{k}$, i.e., $8^3\binom{4}{2}$.

B. Cryptographic Primitives

Our DIP scheme is built on several cryptographic primitives, the primitives include: (i) symmetric encryption, (ii) a family of pseudorandom functions (PRFs), (iii) a family of pseudorandom permutations (PRPs), and (iv) message authentication codes (MACs). Each of the primitives takes a secret key. Intuitively, it means that it is computationally infeasible for an adversary to break the security of a primitive without knowing its corresponding secret key. We also need a systematic adversarial error-correcting code (AECC) [12], [15] to protect against the corruption of a chunk. In conventional error-correcting codes (ECC), when a large file is encoded, it is first broken down into smaller stripes to which ECC is applied independently. AECC uses a family of PRPs as a building block to randomize the stripe structure so that it is computationally infeasible for an adversary to target and corrupt any particular stripe. Note that both FMSR and AECC provide fault tolerance. The difference is that FMSR applies to a file that is striped across servers, while AECC applies to a single chunk stored within a server.

IV. FMSR-Data Integrity Protection

A .overview

Our goal is to augment the basic file operations Upload, Download, and Repair of NCCloud with the DIP feature. During Upload, F-DIP expands the code chunk size by a factor of n'/k' from the AECC. During Download and Repair, FMSR-DIP maintains the same transfer bandwidth requirements when the stored chunks are not corrupted. Also, we introduce an additional Check operation, which verifies the integrity of a small part of the stored chunks by downloading random rows from the servers and checking their consistencies.

B. Basic Operations

In the following discussion, we assume that F-DIP operations are byte oriented.

Upload operation. We first describe how we upload a file F to servers using FMSR-DIP.

Step 1: Generate the per-file secrets. Before uploading F, we generate per-file secrets κENC, κPRP, κPRF, and κMAC.

Step 2: Encode the file using FMSR. We have NCCloud encode F using (n, k)-FMSR to give n(n − k) code chunks {Pi} of b bytes each, where $b = \frac{|F|}{k(n-k)}$. NCCloud also outputs a metadata file containing the file size |F| and encoding coefficients $\{\alpha_{ij}\}$.

Step 3: Encode each code chunk with F-DIP. Consider the ith code chunk P_i. We first apply AECC to the b bytes of the code chunk $\{P_{ir}\} 1 \le r \le b$ to generate b'− b parity bytes $\{P_{ir}\} b+1 \le r \le b'$, where $b' = \frac{bn'}{k'}$. If b is not a multiple of k', then we simply pad the code chunk without affecting the correctness. AECC is used to recover a corrupted row that cannot be recovered by FMSR alone. We apply the same AECC to each of the code chunks. Then we apply the PRF to all b' bytes of the code chunk (including the AECC parities): $P'_{ir} = P_{ir} \oplus PRF(i||r)$, where ⊕ is the XOR operator, and i||r denotes the concatenation of chunk identifier i and row identifier r. PRF protects the integrity of each row, with the chunk and row identifiers as inputs. Finally, we compute a MAC Mi for the first b bytes of the code chunk with PRF: $Mi = MAC(P'_{i1} \| . . . \|P'_{ib})$, where || denotes the concatenation. Note that we do not include AECC parities in the MAC, as typically when we download a file, only the original FMSR code chunk needs to be downloaded and verified by the MAC. The parity bytes are downloaded only when error correction is needed.

Step 4: Update the metadata file and upload. We append n' and k' to the metadata file generated by NCCloud. We also append the MACs of all chunks to the metadata, so that we can retrieve valid and up-to-date MACs for verification. The metadata is encrypted with κENC and replicated to each server, with a small storage overhead (see Section VII). Finally, we upload the F-DIP-encoded chunks P'_i 's to their respective servers based on FMSR.

Check operation. In the Check operation, we verify randomly chosen rows of bytes based on the FMSR code chunks generated by NCCloud.

Step 1: Check the metadata file. We download a copy of the encrypted metadata from each server and check if all copies are identical. Since the metadata file is replicated across all servers, we can run majority voting to restore any corrupted file. We then decrypt the metadata file and retrieve the encoding coefficients $\{\alpha_{ij}\}$, the AECC parameters n' and k', and the MACs $\{M_i\}$.

Step 2: Sampling and row verification. Based on the FDIP-encoded chunk size b', we randomly generate [λb'] distinct indices, where λ ∈ [0, 1] is a tunable checking percentage. For each index r, we download the rth byte from each of the n(n − k) code chunks. Thus, we download λb' rows in total. For the rth row of bytes $\{P'_{ir}\} 1 \le I \le n(n-k)$, we remove the PRF, i.e., $P_{ir} = P'_{ir} \oplus PRF(i||r)$. We then check the consistency of $\{P_{ir}\}$ with respect to the encoding coefficients $\{\alpha_{ij}\}$ as follows. Denote encoding matrix $A = [\alpha_{ij}]\ n(n-k)\times k(n-k)$ and chunk vector $P = [P_{ir}]\ 1 \le I \le n(n-k)$. We construct a system of linear equations, denoted by an $n(n-k) \times k(n-k) + 1$ matrix $A|PT$, such that P^T is the rightmost column of the system. Then the system (and hence the rth row) is said to be consistent if rank $(A|P^T) = \text{rank}(A) = k(n-k)$, meaning that the rth row of bytes can be uniquely decoded to a correct solution that corresponds to the original native chunks. The idea of the above rank checking can be intuitively understood as follows. In FMSR, the k(n−k) code chunks from any k servers can be decoded to the original k(n − k) native chunks, so we must have rank(A) = k(n−k). If chunk vector P is error-free, then by solving the system of linear equations $A|P^T$ we can retrieve the corresponding bytes in the original k(n − k) native chunks, so $\text{rank}(A|P^T) = \text{rank}(A) = k(n-k)$ if the system is consistent. The PRF added to the code chunk obfuscates the bytes and makes the adversary more difficult to corrupt the bytes while maintaining the consistency of the system $A|P^T$. In case of inconsistency, we have rank $(A|PT) > k(n-k)$, while $\text{rank}(A) = k(n-k)$.

Step 3: Error localization. If the r th row is inconsistent, then we know that some bytes in the row are erroneous. We now attempt to localize the erroneous bytes in the row, assuming that there are at most $n-k-1$ failed servers. We first choose any k servers and pick the bytes $\{P_{\tilde{i}r}\}$ (with the PRF removed) for the $k(n-k)$ chunks on those k servers, where the values ˜i's denote the $k(n-k)$ indices of the chosen chunks. Denote encoding matrix $\tilde{A} = [\alpha_{\tilde{i}j}]_{k(n-k)\times k(n-k)}$ and chunk vector $\tilde{P} = [P_{\tilde{i}r}]$. Note that $\tilde{A}$ and $\tilde{P}$ can be viewed as the subsets of A and P defined above, respectively. The MDS property of FMSR guarantees that the system of linear equations formed from these $k(n-k)$ bytes gives a unique solution, as any k out of n servers suffice to recover the original file. However, this unique solution may not be correct, due to the presence of erroneous bytes in $\tilde{P}$. We now pick a chunk Ph from one of the remaining $n-k$ servers. We append its row of encoding coefficients $\{\alpha_{hj}\}$ and byte value P_{hr} to $\tilde{A}$ and $\tilde{P}$, respectively. Thus, we now consider the bytes of a subset of $k(n-k)+1$ code chunks. If $\text{rank}(\tilde{A}) = \text{rank}(\tilde{A}|\tilde{P}T)$, then the system is consistent, and we mark P_{hr} correct. We repeat this step for all the chunks from the remaining $n-k$ servers (setting h to be each of the chunks in turn). After all chunks are exhausted, we pick a new combination of the original k servers and repeat until all n k combinations have been tested. Bytes that are not marked correct at the end of all checks are marked as corrupted. Note that the above error localization step assumes at most $n-k-1$ failed servers. If $n-k$ servers fail, then we may recover the errors by downloading the full F-DIP chunks, as discussed in the Download operation below.

Step 4: Trigger repair. If a server has more than a user specified number of bytes marked as corrupted, we consider it a failed server and trigger the Repair operation.

Download operation. We now describe how we download a file F from servers.

Step 1: Check the metadata file. Refer to Step 1 of Check.

Step 2: Download and decode the F-DIP-encoded chunks for file F. To reconstruct file F, we download $k(n-k)$.

FMSR-DIP-encoded chunks from any k servers (without the AECC parities). After downloading a code chunk, we verify its integrity with the corresponding MAC. We strip the PRFs off the F-DIP-encoded chunks to form the FMSR code chunks, which are then passed to NCCloud for decoding if they are not corrupted. However, if we have a corrupted code chunk, then we can fix it with one of the following approaches:

• Download its AECC parities and apply error correction. Then we verify the corrected chunk with its MAC again.

• Download the $(n-k)$ code chunks from another server.

• A last resort is to download the code chunks from all n servers. We check all rows of the chunks including their AECC parities. The rows with a subset of the bytes marked correct can be recovered with FMSR; the rows with all bytes marked corrupted are treated as erasures and will be corrected with AECC. A file is deemed unrecoverable if there are insufficient code chunks that pass their MAC verifications.

Repair operation. If some server fails (e.g., when losing all data, or when having too much corrupted data that cannot be recovered), then we trigger the repair operation via NCCloud as follows.

Step 1: Check the metadata file. Refer to Step 1 of Check.

Step 2: Download and decode the needed chunks. This is similar to Step 2 of Download, as long as there are at most $n-k$ failed servers. In particular, if there is only one failed server, then instead of trying to download $k(n-k)$ chunks from any k servers, we download one chunk from all remaining $n-1$ servers as in FMSR (see Figure 1).

Step 3: Encode, update metadata, and upload. NCCloud generates $(n-k)$ chunks to store at the new server. Each chunk is encoded with F-DIP again (Step 3 of Upload) and uploaded to the new server. Finally the metadata is updated, encrypted and replicated to all servers (Step 4 of Upload).

C. Integration of DIP into NCCloud

We implement a standalone DIP module and a storage interface module, and integrate them with NCCloud as shown in Figure 4. In the Upload operation, NCCloud generates code chunks for a file based on FMSR. The code chunks will be temporarily stored in the local file system instead of being uploaded to the servers. The DIP module then reads the FMSR code chunks from the local file system, encodes them with DIP, and passes the resulting F-DIP code chunks to the storage interface module, which will upload the F-DIP chunks to multiple servers. In the Download operation, the DIP module checks the integrity of the chunks retrieved from the servers NCCloud DIP FMSR code chunks Storage interface F-DIP code chunks. Servers file before relaying the chunks to NCCloud for decoding.

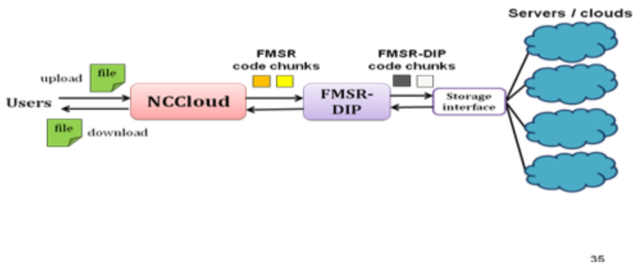

Fig. 4 Integration of FMSR-DIP into NCCloud

D. Instantiating Cryptographic Primitives

All cryptographic primitives use 128-bit secret keys. We require that all secret keys be securely stored on the client side without being revealed to any server. Since the files in the cloud are typically of large size, we expect that the secret keys only incur a small constant overhead. The primitives are instantiated as described below:

Symmetric encryption. We use AES-128 in cipher-block chaining (CBC) mode.

Pseudorandom function (PRF). We use AES-128 for PRF. The PRF input is first transformed to a plaintext block, which is then encrypted with AES-128.

Pseudorandom permutation (PRP). Our PRP implementation is based on AES-128, but applied in a different way as in PRF. Note that the domain size of the PRP is the number of elements to be permuted. To implement a PRP with a small and flexible domain size, we first create a list of indices from 0 to $d-1$, where d is the desired domain size of our PRP. Then we encrypt each index in turn with

AES-128 and sort the encrypted indices. Finally, the permutation is given by the positions of the original indices in the sorted list of encrypted indices.

Message authentication codes (MACs). We use HMACSHA-1 to compute MACs. Adversarial error-correcting codes (AECCs). We apply the systematic AECC, with two main differences. First, for efficiency, we do not encrypt the AECC parities, since we will apply PRF to the entire DIP-encoded chunk after applying AECC. PRF itself serves as an encryption. Second, and most notably, instead of applying a single PRP to the entire code chunk, we first divide the code chunk into k' fragments, and apply a different PRP to each fragment. The secret key of the PRP for each fragment is formed by the XOR-sum of a master PRP secret key and the fragment number. Applying PRP to a fragment rather than a chunk reduces the domain size and hence the overall memory usage. The trade-off is that we reduce the security protection, but it should suffice in practice. Also, our approach is more resilient to burst errors since each byte of a stripe is confined to its own fragment, while in permuting over the entire chunk, a stripe may have many of its bytes clustered together.

V. Experiments and Results

The following experiment is carried out on machine with an intel Pentium 1.86 GHz CPU and 4-GB RAM. The machine is running 64-bit Windows Operating system. We have conducted our experiments on Amazon S3 cloud service [8]. We create (n + 1) = 5 buckets on S3 cloud, which resembles cloud repository (one of them is used as spare node in repair), We focus on measuring the running time of each operation, for n = 4 and k = 2, for varying file sizes,

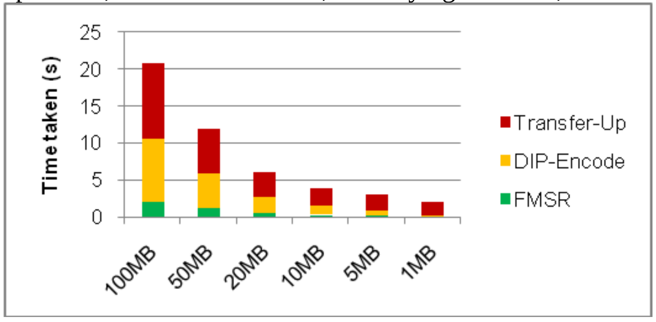

Fig. 5 Running Time of file upload operation

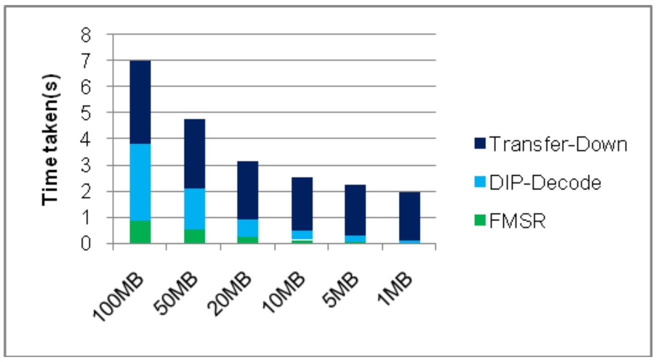

Fig. 6 Running Time of file download operation

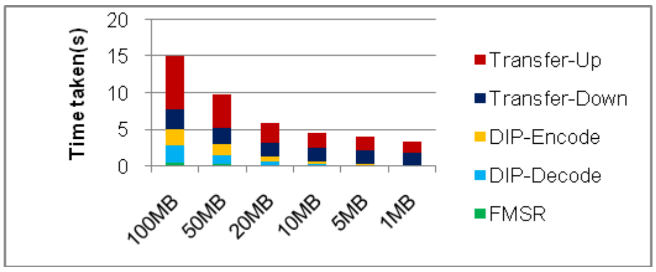

Fig. 7 Running Time of repair operation

VI. CONCLUSION

The paper presents the design and implementation of practical data integrity protection (DIP) scheme for functional minimum storage regenerating (FMSR) codes under a multi-server setting. Our DIP scheme preserves the fault tolerance and repair traffic saving properties of FMSR. To understand the practicality of the integration of FMSR and DIP, we analyze its security strengths and evaluate its running time overhead via our experiments.

REFERENCES

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, “RACS: A Case for Cloud Storage Diversity,” Proc. ACM First ACM Symp. Cloud Computing (SoCC ’10), 2010.
[2] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proc.of FAST*, 2012.
[3] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
[4] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. Soft-ware - Practice & Experience, 27(9):995–1012, Sep 1997.
[5] A.G. Dimakis, P.B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, “Network Coding for Distributed Storage Systems,” IEEE Trans. Information Theory, vol. 56, no. 9, pp. 4539-4551, Sept. 2010.
[6] S. E. Rouayheb and K. Ramchandran. Fractional Repetition Codes for Repair in Distributed Storage Systems. InProc. of Allerton, 2010.
[7] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A View of Cloud Computing,” Comm. the ACM, vol. 53, no. 4, pp. 50-58, 2010.
[8] Amazon Web Services, “Amazon S3,” http://aws.amazon.com/s3, 2013.
[9] Amazon Web Services, “Amazon S3 Availability Event: July 20, 2008,” http://status.aws.amazon.com/s3-20080720.html, July 2008.
[10] K.D. Bowers, A. Juels, and A. Oprea, “HAIL: A High-Availability and Integrity Layer for Cloud Storage,” Proc. 16th ACM Conf. Computer and Comm. Security (CCS ’09), 2009.
[11] A.G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, “A Survey on Network Codes for Distributed Storage,” Proc. IEEE, vol. 99, no. 3, pp. 476-489, Mar. 2011.
[12] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
[13] C. Suh and K. Ramchandran, “Exact-Repair MDS Code Construction Using Interference Alignment,” IEEE Trans. Information Theory, vol. 57, no. 3,pp.1425-1442,Mar.2011.

.