Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage: Theory and Implementation (Supplementary File)

Henry C. H. Chen and Patrick P. C. Lee

1 Additional Related Work

This section supplements Section 2 of the main paper with other related studies.

Data integrity protection is first considered in memory management systems [8]. In online memory checking, a user checks whether each read/write operation in an unreliable memory device is correct, using a small, trusted (and possibly private) memory device. Naor and Rothblum [16] improve the efficiency of online memory checking. Instead of reading all bits of a file during checking, they sample bits from the file. The sampling idea is also used in our data checking scheme.

Proof of retrievability (POR) [15] and *proof of data possession* (PDP) [3] are recent works that explore efficient data integrity checking (i.e., through sampling instead of whole-file checking) in single-server archival storage systems. POR [15] embeds a set of pseudorandom blocks into an encrypted file stored in the server, and the client can later check if the server keeps the pseudorandom blocks. Error correcting codes are also included in the stored file to allow recovery of a small amount of errors within a file. However, the number of checks that the client can issue is limited by the number of the embedded random blocks. On the other hand, PDP [3] allows the client to keep a small amount of metadata. The client can then challenge the server against a set of random file blocks to see if the server returns the proofs that match the metadata on the client side. Both schemes can further minimize the network transfer bandwidth by aggregating proofs into smaller messages. However, such aggregation techniques require the servers to have certain encoding capabilities.

Several follow-up studies on POR and PDP improve their computation and communication complexities (e.g., [10], [12], [22]). Adding protection of *dynamic* files (i.e., files that can be updated after being stored) to PDP is considered in [4], [13]. Some studies focus on the public verifiability of efficient integrity checking schemes (e.g., [5], [20], [23]).

Multi-server (or multi-cloud) storage has been proposed and implemented to protect against data loss [6], [9], [14], [19] and mitigate vendor lock-ins [1]. Oggier et al. [17] consider exact regenerating codes with collaborative repairs, and verify the integrity of fully-regenerated chunks with trusted hashes to guard against adversarial corruptions. Zhu et al. [26] propose a cooperative PDP scheme for multi-cloud storage, but it is difficult to deploy the scheme in practice as it requires the cooperation of different multiple cloud providers.

2 Illustrations of FMSR Codes

This section supplements Section 3.1 of the main paper with additional illustrations of FMSR codes on NCCloud [14]. Figure 1(a) illustrates the entire file upload process in NCCloud using FMSR codes for $n = 4$ and $k = 2$, while Figure 1(b) shows how a file is encoded inside NCCloud using matrix multiplication.

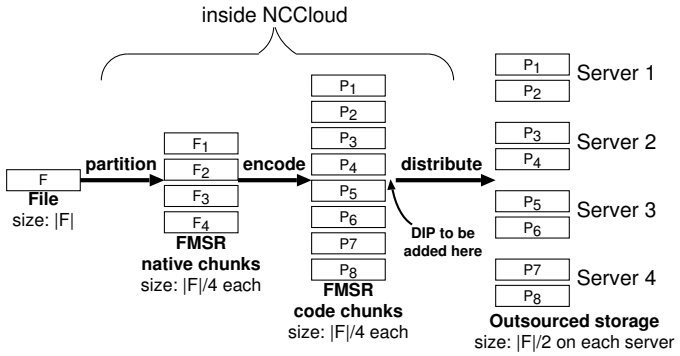
3 Illustrations of FMSR-DIP Codes

This section supplements Section 4 of the main paper with additional illustrations of FMSR-DIP codes. First, we summarize the notations used in Table 1. Figure 2 shows an overview of how we augment an FMSR code chunk into an FMSR-DIP code chunk.

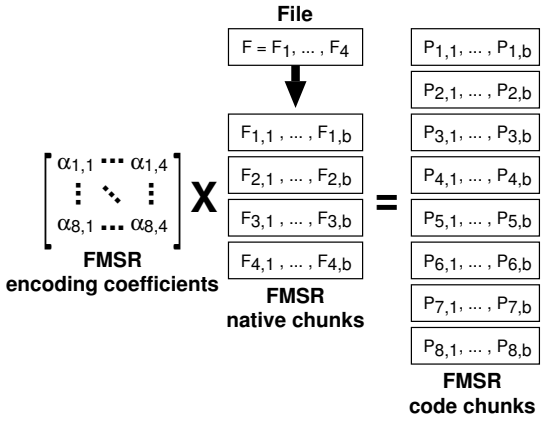
4 Aggregating Rows in Thick Cloud Storage

This section supplements Section 4.1 of the main paper. Our work assumes the thin-cloud setting in which servers only need to support basic read/write operations. Here, we briefly remark on how FMSR-DIP codes can be modified to utilize server-side encoding capabilities when such functionalities are available in thick

• *H. Chen and P. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong (Emails: {chchen,pclee}@cse.cuhk.edu.hk)*



(a) File upload process



(b) Encoding process using matrix multiplication

Fig. 1. An example of the file upload and encoding operations in NCCloud using a (4,2)-FMSR code. Data is stored in four servers, in which the data of any two servers suffice to recover the original file. Each server stores two code chunks of total size $|F|/2$.

TABLE 1
A summary of the key notations.

Symbols	Meaning
n, k	Parameters for FMSR codes
n', k'	Parameters for AECC
b	FMSR chunk size
b'	FMSR-DIP chunk size
F	File to be backed up
$\|F\|$	Original file size
$\{F_i\}$	FMSR native chunks (i.e., partitions of original file)
$\{P_i\}$	FMSR code chunks
$\{P'_i\}$	FMSR-DIP code chunks
$\{\alpha_{ij}\}$	FMSR encoding coefficients
λ	Checking percentage

cloud storage services. Our goal is to aggregate the downloaded bytes during the Check operation, so as to reduce the amount of data transfer.

The idea is as follows. During the Check operation, instead of downloading all the randomly selected bytes from the servers, we can divide the random indices into groups and request each server to return the XOR-sum of all the selected bytes on a group-by-group basis for each chunk. Thus, for each group, we download one byte from each code chunk. Due to the *distributive* nature

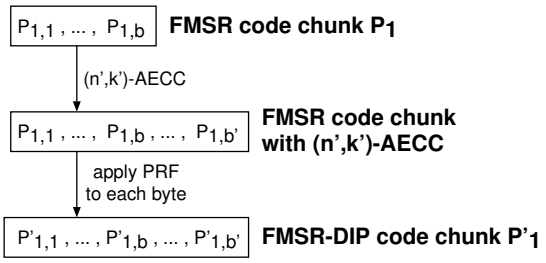


Fig. 2. An overview of how an FMSR code chunk P_1 is augmented into an FMSR-DIP code chunk P'_1.

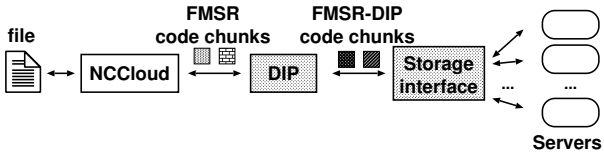


Fig. 3. Integration of the DIP scheme into NCCloud.

of finite field operations, *the XOR-sum of the selected bytes of the code chunks can be decoded to the XOR-sum of the corresponding bytes of the native chunks*. We can then perform rank checking on the XOR'ed bytes instead.

To illustrate, suppose that in the Check operation, we apply the XOR-sum to the r_1th and r_2th rows of the FMSR-DIP code chunks, i.e., $\{P'_{ir_1} \oplus P'_{ir_2}\}_{1 \le i \le n(n-k)}$. First, we remove the PRF, i.e., $P_{ir_1} \oplus P_{ir_2} = P'_{ir_1} \oplus P'_{ir_2} \oplus \mathsf{PRF}(i||r_1) \oplus \mathsf{PRF}(i||r_2)$. Note that:

$$\begin{aligned} P_{ir_1} \oplus P_{ir_2} &= \textstyle\sum_{j=1}^{k(n-k)} \alpha_{ij} F_{jr_1} \oplus \sum_{j=1}^{k(n-k)} \alpha_{ij} F_{jr_2} \\ &= \textstyle\sum_{j=1}^{k(n-k)} \alpha_{ij} (F_{jr_1} \oplus F_{jr_2}). \end{aligned}$$

If $P_{ir_1} \oplus P_{ir_2}$ is error-free, then it can be correctly decoded into the XOR-sum of the r_1th and r_2th rows of bytes of the native chunks. We then apply rank checking to verify $P_{ir_1} \oplus P_{ir_2}$ as before.

We can further reduce data transfer by allowing servers to XOR bytes across different chunks. Note that the XOR-sum of the r_1th and r_2th row of different chunks P_i and $P_{i'}$ (that is $P_{ir_1} \oplus P_{ir_2} \oplus P_{i'r_1} \oplus P_{i'r_2}$) is the following:

$$\begin{aligned} &\left(\textstyle\sum_{j=1}^{k(n-k)} \alpha_{ij} (F_{jr_1} \oplus F_{jr_2})\right) \oplus \left(\textstyle\sum_{j=1}^{k(n-k)} \alpha_{i'j} (F_{jr_1} \oplus F_{jr_2})\right) \\ &= \textstyle\sum_{j=1}^{k(n-k)} (\alpha_{ij} \oplus \alpha_{i'j})(F_{jr_1} \oplus F_{jr_2}). \end{aligned}$$

5 Additional Implementation Details

This section supplements Section 5 of the main paper with additional implementation details.

5.1 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 3. In the Upload operation, NCCloud generates code chunks for a file based on FMSR codes. The code chunks will be temporarily stored in the local filesystem instead of being uploaded to the servers as

in [14]. The DIP module then reads the FMSR code chunks from the local filesystem, encodes them with DIP, and passes the resulting FMSR-DIP code chunks to the storage interface module, which will upload the FMSR-DIP code chunks to multiple servers (or a cloud-of-clouds [1], [6], [14]). In the Download operation, the DIP module checks the integrity of the chunks retrieved from the servers before relaying the chunks to NCCloud for decoding. Note that we can issue a range GET request to download a selected range of bytes.

Our DIP module mostly operates on a per-chunk basis. Thus, it can harness parallelism in today's multi-core technologies by concurrently encoding different code chunks. For example, a (4,2)-FMSR code will create eight FMSR code chunks, each of which can be encoded into an FMSR-DIP code chunk with a dedicated process. In an eight-core machine, we can have up to eight-fold speed-up over the sequential approach. This can significantly speed up the entire DIP operation.

Our current implementation uses a *modular* approach that separates the FMSR code module (i.e., NCCloud) and the DIP module. It is possible to combine the two modules into a single design to reduce the overhead, so as to eliminate the passing of FMSR code chunks between NCCloud and our DIP module using a local staging directory. In addition, we may exploit certain inherent properties of such combination to further reduce the computational overhead, and we pose this issue as future work. On the other hand, our modular approach allows us to *flexibly* enable DIP on demand in real deployment.

5.2 Instantiating Cryptographic Primitives

We implement all cryptographic operations using OpenSSL 1.0.0g [18]. All cryptographic primitives use 128-bit secret keys. 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 follow the approach in Method 1 of [7]. 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. A more efficient way can be used to generate a small PRP [10], at the expense of a larger storage overhead.

Message authentication codes (MACs). We use HMAC-SHA-1 to compute MACs.

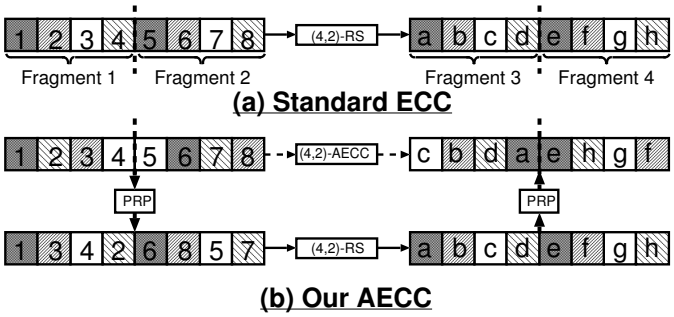


Fig. 4. Comparing standard ECC and our AECC (using for example (4,2)-Reed-Solomon encoding). The bytes of the same shade correspond to the same stripe.

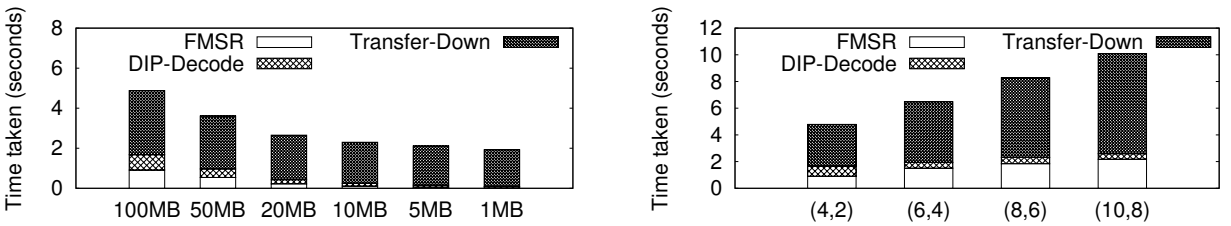
Adversarial error-correcting codes (AECCs). We apply the systematic AECC adapted from [10], [11] with two main differences. First, for efficiency, we do not encrypt the AECC parities, since we will apply PRF to the entire DIP-encoded chunk after applying AECC. PRF itself serves as an encryption. Second, 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. 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.

Figure 4 shows our AECC implementation, in which we use zfec [25] for the underlying systematic ECC (based on Reed-Solomon codes). We first apply a PRP to each of the k' fragments within the FMSR code chunk. We then apply systematic ECC to the permuted chunk (divided into b/k' stripes of k' bytes each), where the ith stripe ($1 \leq i \leq b/k'$) comprises the bytes in the ith positions of all fragments. Finally, we permute each fragment of the ECC parities, and append the permuted parities to the code chunk.

6 Uses of Security Primitives

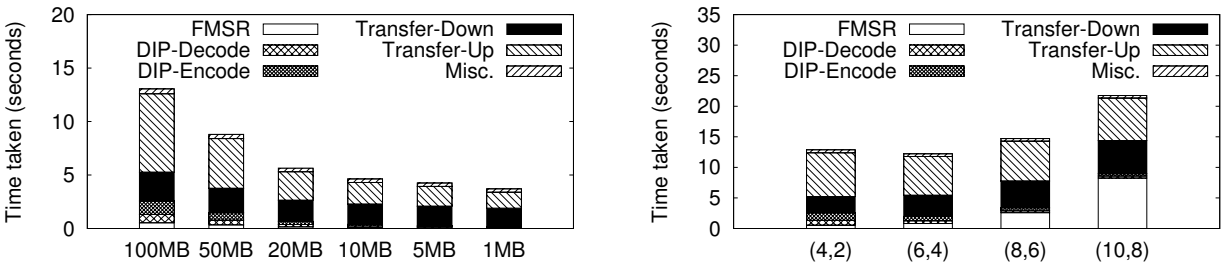
This section supplements Section 6 of the main paper with additional details about the effects of the various security primitives used in FMSR-DIP codes.

Pseudorandom function (PRF). The effect of applying PRF on the data is similar to encrypting the data. It randomizes the data so that it is infeasible for the adversary to manipulate the original data and hence corrupt the data in such a way that the corrupted bytes form consistent systems of linear equations during the Check operation. PRF is important for guarding against a *mobile*



(a) Different file sizes (b) Different (n, k) values of FMSR codes

Fig. 5. Running time of the entire Download operation on a local cloud.



(a) Different file sizes (b) Different (n, k) values of FMSR codes

Fig. 6. Running time of the entire Repair operation on a local cloud.

Byzantine adversary [9], which can possibly corrupt data on all servers over time via creeping corruption [9].

To illustrate the necessity of PRFs in our construction, consider the following attack on our construction in the absence of PRFs. Remember that our adversary is mobile, and thus can zero out bytes at a specific offset on all chunks (i.e., a row) across multiple epochs. Note that a random linear combination of zeroes is always zero, so a row of zeroes is always consistent and decodable (to give zeroes). If the adversary corrupts only a few rows at a time, it would be near impossible for us to detect the attack before too many rows are corrupted, rendering our data irrecoverable.

Symmetric encryption. We encrypt the metadata to hide the encoding coefficients of FMSR codes. This protects against the scenario where the PRF values can be recovered with known encoding coefficients and original file content.

Adversarial error-correcting codes (AECC). We use AECC to randomize the stripe structure, so that it is infeasible for the adversary to deterministically render chunks unrecoverable.

Message authentication codes (MAC). We include the MACs of individual chunks as metadata, and replicate them to all servers to allow integrity verification of any chunks.

7 Additional Evaluation Results

This section supplements Section 7 of the main paper. Here, we present evaluation results of the running time overhead of FMSR-DIP codes in the Download and Repair operations. We further analyze the monetary cost overhead with the pricing models of different commercial cloud providers.

7.1 Running Time Overhead of Download and Repair

Figures 5 and 6 show the running times of Download and Repair, respectively. Under the (4,2)-FMSR code, in Download, the DIP-Decode part accounts for 3.80% (for 1MB) to 15.7% (for 100MB) of the overall Download time, while in Repair, the DIP-Decode and DIP-Encode parts altogether account for 3.19% (for 1MB) to 15.7% (for 100MB) of the overall Repair time. Using a 100MB file and varying (n, k) for FMSR codes, in Download, the DIP-Decode part accounts for 4.14% (for (10,8)-FMSR) to 15.8% (for (4,2)-FMSR) of the overall Download time, while in Repair, the DIP-Decode and DIP-Encode parts altogether account for 3.24% (for (10,8)-FMSR) to 15.8% (for (4,2)-FMSR) of the overall Repair time.