Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage

Henry C. H. Chen and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{*chchen,pclee*}@*cse.cuhk.edu.hk*

***Abstract*—To protect outsourced data in cloud storage against corruptions, enabling integrity protection, fault tolerance, and efficient recovery for cloud storage becomes critical. Regenerating codes provide fault tolerance by striping data across multiple servers, while using less repair traffic than traditional erasure codes during failure recovery. Therefore, we study the problem of remotely checking the integrity of regenerating-coded data against corruptions under a real-life 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 is designed under a Byzantine adversarial model, and enables a client to feasibly verify the integrity of random subsets of outsourced data against general or malicious corruptions. It works under the simple assumption of thin-cloud storage and allows different parameters to be fine-tuned for the performance-security trade-off. We implement and evaluate the overhead of our DIP scheme in a real cloud storage testbed under different parameter choices. We demonstrate that remote integrity checking can be feasibly integrated into regenerating codes in practical deployment.**

***Index Terms*—remote data checking, secure and trusted storage systems, implementation, experimentation**

I. Introduction

Cloud storage offers an on-demand data outsourcing service model, and is gaining popularity due to its elasticity and low maintenance cost. However, security concerns arise when data storage is outsourced to third-party cloud storage providers. It is 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 Byzantine attacks.

One major use of cloud storage is *long-term archival*, which represents a workload that is written once and rarely read. While the stored data is rarely read, it remains necessary to ensure its integrity for disaster recovery or compliance with legal requirements (e.g., [34]). Since it is typical to have a huge amount of archived data, whole-file checking becomes prohibitive. *Proof of retrievability* (POR) [23] and *proof of data possession* (PDP) [6] have thus been proposed to verify the integrity of a large file by spot-checking only a fraction of the file via various cryptographic primitives.

Suppose that we outsource storage to a *server*, which could be a storage site or a cloud storage provider. If we detect corruptions in our outsourced data (e.g., when a server crashes or is compromised), then we should *repair* the corrupted data and restore the original data. However, putting all data in a single server is susceptible to the single-point-of-failure problem [5] and vendor lock-ins [1]. As suggested in [1], [5], a plausible solution is to stripe data across multiple servers. Thus, to repair a failed server, we can (i) read data from other surviving servers, (ii) reconstruct the corrupted data of the failed server, and (iii) write the reconstructed data to a new server. POR and PDP are proposed for the single-server case. MR-PDP [16] and HAIL [11] extend integrity checks to a multi-server setting using *replication* and *erasure coding*, respectively. In particular, erasure coding (e.g., Reed-Solomon codes [29]) has less storage overhead than replication under the same fault-tolerance level.

Regenerating codes [17] have recently been proposed 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 (or corrupted) 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 [11], 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 [22] and construct *FMSR-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 [22]. Second, it assumes only the thin-cloud interface [32], i.e., the servers only need to support the standard read/write functionalities. Third, it exports several tunable parameters that allow clients to trade performance for security.

We implement FMSR-DIP, and evaluate its overhead over the existing FMSR implementation through extensive testbed experiments in a cloud storage environment. We evaluate the

running times of different basic operations, including upload, check, download, and repair, for different parameter choices of our DIP scheme. In addition, we evaluate the monetary cost overhead of FMSR-DIP should it be deployed in commercial clouds. Our work demonstrates the feasibility of enabling integrity protection, fault tolerance, and efficient recovery for cloud storage.

The rest of the paper proceeds as follows. Section II reviews related work in remote data integrity checking. Section III provides necessary preliminaries for our design. Section IV presents the design of FMSR-DIP. Section V provides the implementation details of FMSR-DIP, and how several parameters can be adjusted for different performance needs. Section VI gives preliminary security analysis for FMSR-DIP. Section VII reports evaluation results of FMSR-DIP in a cloud storage testbed. Finally, Section VIII concludes the paper.

II. Related Work

We consider the problem of checking the integrity of *static* data, which is typical in long-term archival storage systems. This problem is first considered under a single server scenario by Juels et al. [23] and Ateniese et al. [6], giving rise to the similar notions *proof of retrievability* (POR) and *proof of data possession* (PDP), respectively. The basic POR scheme [23] embeds a set of pseudorandom blocks into an encrypted file stored on the server, and the client can check if the server keeps the pseudorandom blocks later on. 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 [6] 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 allowing proofs to be aggregated. 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 complexity (e.g., [12], [18], [31]). Adding protection of *dynamic* files (i.e., files that can be updated after being stored) to PDP is considered in [7], [19]. Some studies focus on the public verifiability of efficient integrity checking schemes (e.g., [8], [27], [33]).

A major limitation of the above schemes is that they are designed for a *single* server setting. If the server is fully controlled by an adversary, then the above schemes can only provide detection of corrupted data, but cannot recover the original data. This leads to the design of efficient data checking schemes in a *multi-server* setting. By striping redundant data across multiple servers, the original files can still be recovered from a subset of servers even if some servers are down or compromised. Efficient data integrity checking has been proposed for different redundancy schemes, such as replication [16], erasure coding [11], [30], and regenerating coding [13]. We point out that although Chen et al. [13] also consider regenerating-coded storage, there are several key differences with our work. First, their design extends single-server compact POR by Shacham et al. [31]. However, such direct adaptation inherits some shortcomings of the single-server scheme such as a large storage overhead, as the amount of data stored increases with a more flexible checking granularity in the scheme of Shacham et al. [31]. We believe a better solution is possible by exploiting the cross-server redundancies in a multiple-server setting. Second, the storage scheme of [13] assumes that storage servers have the encoding capabilities of generating a random linear combination of the data, while we consider a thin-cloud setting [32] where servers only need to support standard read/write functionalities. Finally, the evaluations of [13] are conducted on a single desktop, while ours are conducted on a real cloud storage testbed. We expect that evaluating atop a real system can give a more comprehensive and realistic view of the actual performance of storage operations.

Multi-server (or multi-cloud) storage has been proposed and implemented to protect against data loss [9], [11], [22] and mitigate vendor lock-ins [1]. The closest related work to ours is HAIL [11], which stores data via erasure coding. As stated in Section I, as HAIL operates on a per-file basis, it is non-trivial to directly apply HAIL to regenerating codes. In addition, our work focuses more on the practical issues. We address how different parameters can be adjusted for the performance-security trade-off in practical deployment.

III. Preliminaries

In this section, we provide the background details, based on existing studies, for our data integrity protection (DIP) scheme. We first describe the *functional minimum storage regenerating (FMSR)* code implementation considered in this paper. Then we state the threat model and the cryptographic primitives being used in our DIP scheme.

A. FMSR Implementation

We first review the FMSR implementation [22]. FMSR belongs to *Maximum Distance Separable (MDS)* codes. An MDS code is defined by the parameters (n, k), where $k < n$. It encodes a file F of size $|F|$ into n pieces of size $|F|/k$ each. An (n, k)-MDS code states that the original file can be reconstructed from any k out of n pieces (i.e., the total size of data required is $|F|$). An extra feature of FMSR is that a specific piece can be reconstructed from data of size less than $|F|$. FMSR is built on *regenerating codes* [17], which minimize the repair bandwidth while preserving the MDS property based on the concept of network coding [2].

We consider a distributed storage setting in which a file is striped over n servers using an (n, k)-MDS code. Each server can be a storage site or even a cloud storage provider, and is independent of other servers. Suppose that one server fails. Our goal is to reconstruct the lost data of the failed server in a new server, so as to maintain the (n, k)-MDS fault tolerance. We define the *repair traffic* as the amount of data being *read* from other surviving servers so as to reconstruct the lost data

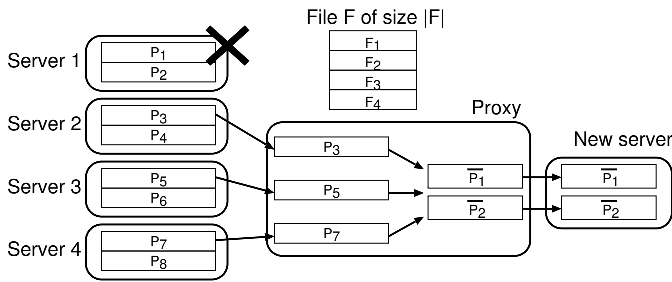


Fig. 1. An example of how a file is stored and repaired in (4,2)-FMSR. Each of the code chunks $P_1, \cdots, P_8$ is a random linear combination of the native chunks F_1, F_2, F_3, F_4. $\overline{P_1}$ and $\overline{P_2}$ are distinct random linear combinations of P_3, P_5 and P_7.

and write the reconstructed data to the new server. We assume that there is a *proxy* that handles the read, reconstruction, and write operations during repair.

Figure 1 shows how the FMSR works for $n = 4$ and $k = 2$. An (n, k)-FMSR code splits a file of size $|F|$ evenly into $k(n-k)$ *native chunks*, and encodes them into $n(n-k)$ *code chunks*, where each native and code chunk has size $\frac{|F|}{k(n-k)}$. Each code chunk, denoted by P_i (where $1 \leq i \leq n(n-k)$), is constructed by a random linear combination of the native chunks, similar to the idea in [28]. The $n(n-k)$ code chunks are stored on n servers (i.e., $n-k$ code chunks per server), where the $k(n-k)$ code chunks from any k servers can be decoded to reconstruct the original data. Decoding can be done by inverting an encoding matrix as described in [26].

Suppose now that one server fails and loses all data. The conventional repair method for a single-server failure is to simply reconstruct the whole file by contacting any k surviving servers, so the repair traffic is $|F|$. Note that this repair method applies to all (n, k)-MDS codes. On the other hand, in FMSR, we first randomly pick a chunk from each of the $(n-1)$ surviving servers, and then generate $(n-k)$ random linear combinations of these $(n-1)$ chunks to store on a new server. To guarantee that the MDS fault tolerance is preserved after multiple rounds of repair, NCCloud performs two-phase checking on the new code chunks generated in the repair operation [22]. In the case of (4,2)-FMSR (see Figure 1), the repair traffic is reduced by 25% to $0.75|F|$. It is shown that the repair traffic of FMSR can be further reduced to 50% for $k = n - 2$ if n is large [22].

Note that FMSR is a *non-systematic code* that keeps only code chunks rather than native chunks as in systematic codes. To access part of a file, the client needs to download and decode the entire file, and this is not suited to applications that need random reads of different parts of a file. Nevertheless, FMSR is suited to long-term archival applications, where the read frequency is low and each read operation typically restores the entire file.

B. Threat Model

We adopt the adversarial model in [11] as our threat model. We assume that an adversary is *mobile Byzantine*, meaning that the adversary compromises a subset of servers in different time *epochs* (i.e., mobile) and exhibits arbitrary behaviors on the data stored in the compromised servers (i.e., Byzantine). To ensure meaningful file availability, we assume that the adversary can compromise and corrupt data in at most $n - k$ out of the n servers in any epoch, subject to the (n, k)-MDS fault tolerance requirement. At the end of each epoch, the client can ask for randomly chosen parts of remotely stored data and run a probabilistic checking protocol to verify the data integrity. Servers under the control of the adversary may or may not correctly return data requested by the client. If corruption is detected, then the client may trigger the repair phase to repair corrupted data.

Instead of performing whole-file checking, which incurs a substantial transfer overhead, it is only feasible for the client to randomly sample data for integrity checking. The adversary may corrupt a small portion of data within the error-correcting capability in each epoch, but the level of corruption can render the errors unrecoverable after several epochs if they are not spotted early. This leads to *creeping corruption* [11]. Thus, it is necessary that the client can quickly spot the corrupted data without accessing the whole file.

C. Cryptographic Primitives

Our DIP scheme is built on several cryptographic primitives, whose detailed descriptions can be found in [20], [21]. 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. Design

We now present our design of DIP atop the FMSR code, and we call the new code *FMSR-DIP*. Our DIP scheme operates on the FMSR code chunks generated by NCCloud [22], which is deployed as a client-side proxy that stripes data among multiple servers (see Figure 1 in Section III).

A. Design Goals

We first state the design goals of FMSR-DIP.

Preservation of regenerating code properties. We preserve the fault tolerance requirement and repair traffic saving

of FMSR (with up to a small constant overhead) as compared to the conventional repair method in erasure codes.

Thin-cloud storage [32]. Each server (or cloud storage provider) only provides the basic interface for clients to read and write their stored files. No computation capabilities on the servers are required to support our DIP scheme. Specifically, most cloud storage providers nowadays provide a RESTful interface, which includes the commands PUT and GET. PUT allows writing to a file as a whole (no partial updates), and GET allows reading from a selected range of bytes of a file via a *range GET* request. Our DIP scheme should use only the PUT and GET commands to interact with each server.

Our approach is different from other smart storage services where servers can aggregate the proofs of multiple checks (e.g., [6], [11]). We note that several cloud providers nowadays provide both storage and computation services, with the data transfer between these services free-of-charge (e.g., Amazon S3 [4] and EC2 [3]). Thus, cloud storage servers with encoding capabilities can be achieved by combining these two services, with the additional expense of renting the computation service. However, this approach reduces portability [32] and introduces a complex cost model.

Flexibility. There should not be any limits on the number of possible challenges that the client can make, since files can be kept for long-term archival. Also, the challenge size should be *adjustable* with different parameter choices, and this is useful when we want to lower the detection rate when the stored data grows less important over time. Such flexibility should come without any additional penalties.

Cost minimization. The cloud storage usage fee is mainly charged based on the storage space, transfer bandwidth, and number of requests. To minimize the storage space and transfer bandwidth, we use AECC (which is also an MDS code). In particular, the storage overhead of FMSR-DIP should come only from the AECC applied to each code chunk. Also, to reduce the number of requests while being compatible with the thin-cloud setting, we seek to reduce the number of challenge/response pairs between the client and the servers.

B. Notation

We now define notation for FMSR-DIP, based on FMSR described in Section III-A. For an (n,k)-FMSR code, we define $\{\alpha_{ij}\}_{1\leq i\leq n(n-k),1\leq j\leq k(n-k)}$ as the set of encoding coefficients that encode $k(n-k)$ native chunks $\{F_j\}_{1\leq j\leq k(n-k)}$ into $n(n-k)$ code chunks $\{P_i\}_{1\leq i\leq n(n-k)}$. Thus, each code chunk P_i is formed by $P_i = \sum_{j=1}^{k(n-k)} \alpha_{ij}F_j$. All arithmetic operations are performed in the Galois Field GF(2^8).

We use the cryptographic primitives stated in Section III-C, and define *per-file* secret keys $\kappa_{\text{ENC}}, \kappa_{\text{PRF}}, \kappa_{\text{PRP}}$ and κ_{MAC} for the encryption, PRF, PRP, and MAC operations, respectively. The usage of these keys should be clear from the context and are omitted below for clarity. Also, we implement AECC as an (n',k') error-correcting code, which encodes k' fragments of data into n' fragments such that up to $\lfloor (n'-k')/2 \rfloor$ errors, or up to $n'-k'$ erasures, can be corrected.

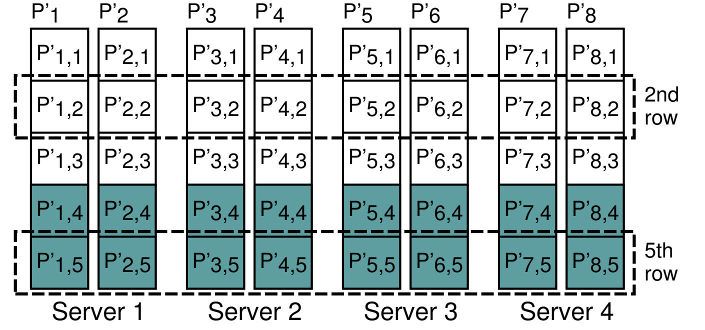


Fig. 2. Integration of DIP into (4,2)-FMSR. In this example, each FMSR code chunk P_i is of size three bytes. FMSR-DIP encodes each chunk with (5,3)-AECC to give $\{P'_i\}$, so bytes $P'_{i,4}$ and $P'_{i,5}$ are the AECC parities of the ith chunk. Then $P'_{1,2}, P'_{2,2}, \cdots, P'_{8,2}$ form the 2nd row and $P'_{1,5}, P'_{2,5}, \cdots, P'_{8,5}$ form the 5th row.

Each code chunk P_i from NCCloud is encoded by FMSR-DIP into P'_i. We define a *row* as a collection of all bytes that are at the same offset of all FMSR-DIP-encoded chunks. That is, the rth row corresponds to the bytes $\{P'_{ir}\}_{1\leq i\leq n(n-k)}$. Figure 2 shows the layout of FMSR-DIP-encoded chunks based on $(4,2)$-FMSR.

C. Overview of FMSR-DIP

Our goal is to augment the basic file operations Upload, Download, and Repair of NCCloud with the DIP feature. During Upload, FMSR-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 (with up to a small constant overhead) 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.

Unlike HAIL [11], which applies DIP to the whole file, we apply DIP to each FMSR code chunk generated by NCCloud. Thus, when NCCloud reconstructs new code chunks during the repair of a failed server, we can directly apply DIP to the new code chunks without accessing the whole file. This preserves the property of repair traffic saving of FMSR. We describe the details of the operations below to explain how our DIP scheme works.

D. Basic Operations

In the following discussion, we assume that FMSR-DIP operates in units of *bytes*. In Section V-C, we discuss how we relax this assumption to trade security for performance.

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 κ_{ENC}, κ_{PRP}, κ_{PRF}, and κ_{MAC}.

Step 2: Encode the file using FMSR. We have NCCloud encode F using (n,k)-FMSR to give $n(n-k)$ code chunks

$\{P_i\}$ 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 FMSR-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 (i.e., with the same permutations and encoding parameters).

Then we apply the PRF to all b' bytes of the code chunk (including the AECC parities): $P'_{ir} = P_{ir} \oplus \mathsf{PRF}(i||r)$, where $\oplus$ 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 M_i for the first b bytes of the code chunk with PRF: $M_i = \mathsf{MAC}(P'_{i1}|| \dots ||P'_{ib})$, 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 κ_{ENC} and replicated to each server, with a small storage overhead (see Section VII). Finally, we upload the FMSR-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 FMSR-DIP-encoded chunk size b', we randomly generate $\lfloor \lambda b' \rfloor$ distinct indices, where $\lambda \in (0, 1]$ is a tunable *checking percentage*. For each index r, we download the rth byte from each of the $n(n-k)$ code chunks (constituting a row). Thus, we download $\lambda b'$ rows in total.

For the rth row of bytes $\{P'_{ir}\}_{1 \le i \le n(n-k)}$, we remove the PRF, i.e., $P_{ir} = P'_{ir} \oplus \mathsf{PRF}(i||r)$. We then check the consistency of $\{P_{ir}\}$ with respect to the encoding coefficients $\{\alpha_{ij}\}$ as follows. Denote encoding matrix $\mathbf{A} = [\alpha_{ij}]_{n(n-k) \times k(n-k)}$ and chunk vector $\mathbf{P} = [P_{ir}]_{1 \le i \le n(n-k)}$. We construct a system of linear equations, denoted by an $n(n-k) \times k(n-k) + 1$ matrix $\mathbf{A}|\mathbf{P}^T$, such that $\mathbf{P}^T$ is the rightmost column of the system. Then the system (and hence the rth row) is said to be *consistent* if $\mathbf{rank}(\mathbf{A}|\mathbf{P}^T) = \mathbf{rank}(\mathbf{A}) = k(n-k)$, 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 $\mathbf{rank}(\mathbf{A}) = k(n-k)$. If chunk vector $\mathbf{P}$ is error-free, then by solving the system of linear equations $\mathbf{A}|\mathbf{P}^T$ we can retrieve the corresponding bytes in the original $k(n-k)$ native chunks, so $\mathbf{rank}(\mathbf{A}|P^T) = \mathbf{rank}(\mathbf{A}) = k(n-k)$ 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 $\mathbf{A}|\mathbf{P}^T$. In case of inconsistency, we have $\mathbf{rank}(\mathbf{A}|\mathbf{P}^T) > k(n-k)$, while $\mathbf{rank}(\mathbf{A}) = k(n-k)$.

Step 3: Error localization. If the rth 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 $\tilde{i}$'s denote the $k(n-k)$ indices of the chosen chunks. Denote encoding matrix $\tilde{\mathbf{A}} = [\alpha_{\tilde{i}j}]_{k(n-k) \times k(n-k)}$ and chunk vector $\tilde{\mathbf{P}} = [P_{\tilde{i}r}]$. Note that $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{P}}$ can be viewed as the subsets of $\mathbf{A}$ and $\mathbf{P}$ 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 $\tilde{\mathbf{P}}$.

We now pick a chunk P_h from one of the remaining $n-k$ servers. We append its row of encoding coefficients $\{\alpha_{hj}\}$ and byte value P_{hr} to $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{P}}$, respectively. Thus, we now consider the bytes of a subset of $k(n-k)+1$ code chunks. If $\mathbf{rank}(\tilde{\mathbf{A}}) = \mathbf{rank}(\tilde{\mathbf{A}}|\tilde{\mathbf{P}}^{\mathbf{T}})$, 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 $\binom{n}{k}$ 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 FMSR-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 (see below).

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 FMSR-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 FMSR-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 (see Section III-B). 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 in Section III).

Step 3: Encode, update metadata, and upload. NCCloud generates $(n-k)$ chunks to store at the new server. Each chunk is encoded with FMSR-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).

V. Implementation

In this section, we describe our FMSR-DIP implementation atop NCCloud [22] and how we instantiate cryptographic primitives. Also, we address how we fine-tune various design parameters to trade security for performance.

A. 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. The code chunks will be temporarily stored in the local filesystem instead of being uploaded to the servers as in [22]. 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 chunks to multiple servers (or a cloud-of-clouds [1], [9], [22]). In the Download operation, the DIP module checks the integrity of the chunks retrieved from the servers

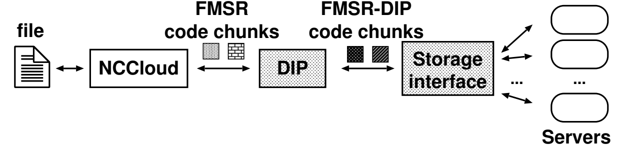


Fig. 3. Integration of FMSR-DIP into NCCloud.

before relaying the chunks to NCCloud for decoding. Note that we can issue a range GET request to download a selected range of bytes (see Section IV-A).

B. Instantiating Cryptographic Primitives

We implement all cryptographic operations using OpenSSL 1.0.0g [24]. 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. Section V-C discusses how the size of the PRF output can be fine-tuned.

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 [10]. 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 [12], at the expense of a larger storage overhead.

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

Adversarial error-correcting codes (AECCs). We apply the systematic AECC adapted from [15], [12] as described in Section III-C, 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 (see Section VI). Also, our approach is more resilient to burst errors since each byte of a stripe is confined to its own

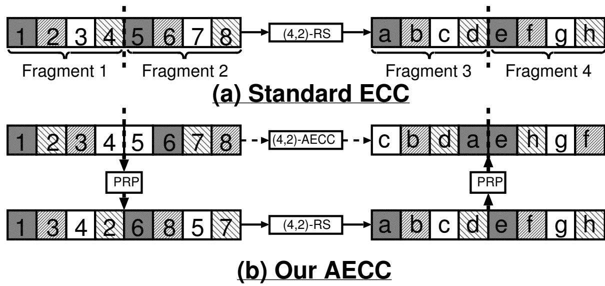


Fig. 4. Comparing standard ECC and our AECC (using for example (4,2)-Reed-Solomon encoding). The bytes of the same color correspond to the same stripe.

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 [35] 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, which is divided into b/k' stripes of k' bytes each, where the ith stripe (where $1 \leq i \leq b/k'$) 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.

C. *Trade-off Parameters*

In Section IV, FMSR-DIP operates in units of bytes. However, byte-level operations may make the implementation inefficient in practice, especially for large files. Here, we describe how FMSR-DIP can operate in units of *blocks* (i.e., a sequence of bytes) to trade security for performance. In the following, we describe the possible tunable parameters that are supported in FMSR-DIP.

PRP block size. Instead of permuting bytes, we can permute blocks of a tunable size (called the *PRP block size*). A larger PRP block size increases efficiency, but at the same time decreases security guarantees.

PRF block size. In a byte-level PRF operation, we can simply take the first byte of the AES-128 output as the PRF output. In fact, we can also compute a longer PRF and apply the PRF output to a block of bytes of a tunable size (called the *PRF block size*). To extend the PRF beyond the AES block size (16 bytes), we can pad the nonce with a chain of input blocks of 16 bytes each, and encrypt them using CBC mode. However, setting the PRF block size to larger than 16 bytes shows minimal performance improvement, as AES is invoked once for every 16 bytes of input in CBC mode and the total number of AES invocations remains the same for a larger PRF block size.

Check block size. Reading data from cloud storage is priced based on the number of GET requests. In the Check operation, downloading one byte per request will incur a huge monetary overhead. To reduce the number of GET requests, we can check a block of bytes of a tunable size (called the *check block size*). The checked blocks at the same offset of all code chunks will contain multiple rows of bytes. Although not necessary, it is recommended to set the check block size as a multiple of the PRF block size, so as to align with the PRF block operations.

AECC parameters. The AECC parameters (n', k') control the error tolerance within a code chunk and the domain size of the PRP being used in AECC. Given the same k', a larger n' implies better protection, but introduces a higher computational overhead.

Checking percentage. The checking percentage λ defines the percentage of data of a file to be checked in the Check operation. A larger λ implies more robust checking, at the expense of both higher monetary and performance overheads with more data to download and check.

VI. Security Analysis

In this section, we elaborate the design choices of FMSR-DIP and investigate its security guarantees.

A. *Uses of Security Primitives*

We briefly summarize the effects of various security primitives used in FMSR-DIP.

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* adversary [11], which can possibly corrupt data on all servers over time via creeping corruption (see Section III-B).

Symmetric encryption. We encrypt the metadata to hide the FMSR encoding coefficients. 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 (see Section III-C).

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.

B. *Security Guarantees*

We provide a sketch of analysis of the robustness of FMSR-DIP against adversarial attacks. In the interest of space, we refer readers to a more detailed mathematical analysis in the full version of this paper [14].

Recall from Section V-B that an FMSR code chunk is encoded by (n', k')-AECC. The code chunk is divided into k' fragments and b/k' stripes. Each fragment is permuted by a PRP of size b/k', and then each stripe is encoded by an (n', k')-ECC to give a total of n' bytes each, so the code chunk is encoded by (n', k')-AECC into n' fragments (see Figure 4 in Section V-B). Each stripe can correct up to $n' - k'$ erasures or $\lfloor (n' - k')/2 \rfloor$ errors.

We assume that the goal of an adversary is to make at least any one stripe *unrecoverable* by corrupting more than $(n' - k')/2$ bytes from the same stripe, while evading detection by our probabilistic row verification in the Check operation. Note that there is a trade-off of choosing how many bytes to corrupt. A higher corruption rate means that the adversary can corrupt more bytes in a stripe, but the corruption is also easier to be detected by our row verification.

Here, we conduct simulations to understand whether it is likely for the adversary to make a stripe unrecoverable while evading detection. In the simulations, we consider a 100MB chunk encoded with $(110, 100)$-AECC, a checking percentage of 0.4% and a check block size of 4KB (i.e., one check block per fragment). If any corrupted byte overlaps with a check block, then the corrupted byte is detected. We experiment with various corruption rates, each for 3,000,000 runs. We have yet to find any run in which the adversary succeeds in making a stripe unrecoverable while evading detection.

VII. Evaluations

We evaluate the practicality of FMSR-DIP in a real storage setting by measuring the overhead of DIP in the Upload, Check, Download, and Repair operations. We empirically evaluate the running time overhead atop a local cloud storage testbed, and we further analyze the cost overhead with the pricing models of different commercial cloud providers.

A. Running Time Analysis

We first conduct testbed experiments on a local cloud platform that is built on OpenStack Swift 1.4.2 [25]. We deploy our FMSR-DIP implementation in single-threaded mode on a machine equipped with Intel Xeon E5620, 16GB RAM, and 64-bit Ubuntu 11.04. The machine is connected via a Gigabit switch to an OpenStack Swift platform that is attached with 15 nodes. We create multiple containers on Swift, such that each container mimics a storage server.

We focus on measuring the running time of each operation. We assume that all file objects being processed remain intact (i.e., without corruptions) throughout an operation, so that we can measure the overhead of FMSR-DIP in normal usage. Our results are averaged over 40 runs.

Upload. We investigate the effects of four sets of parameters on the running time of the Upload operation, including (i) the input file size, (ii) the (n, k) parameters of FMSR, (iii) the (n', k') parameters of AECC, and (iv) the block sizes of PRP and PRF. We vary one set of these parameters each time, while fixing the other three sets at default values. By default, we use a 100MB file, (4,2)-FMSR, (110,100)-AECC, and a block size of 256B for both PRP and PRF.

Figure 5 plots the running times of the Upload operation for different sets of parameters, along with their 95% confidence intervals. We further break down each running time result into three parts denoted by different labels: (i) "FMSR", the time of encoding a file into FMSR chunks by NCCloud, (ii) "DIP-Encode", the time of encoding the FMSR chunks with our DIP scheme, and (iii) "Transfer-Up", the network transfer time of uploading FMSR-DIP chunks and metadata to the local cloud.

From Figure 5(a), we observe that the fractional overhead of DIP encoding increases with the file size, and it ranges from 9.36% (for 1MB) to 41.2% (for 100MB) of the overall time of Upload. The reason is that for larger files, the connection setup overhead of the data transmission becomes less dominant. We expect that the fractional overhead of DIP encoding is smaller when the servers are deployed over the Internet, where the transmission time plays a bigger part in the Upload operation.

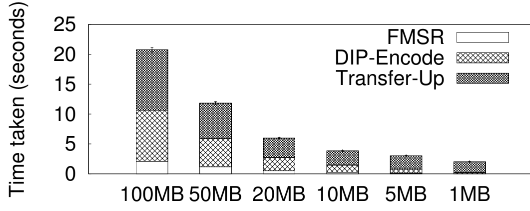
As shown in Figures 5(b) and 5(c), the DIP encoding time increases with the redundancy levels of the underlying FMSR and AECC implementations. For instance, the DIP encoding time increases from 6.432s to 8.467s when the redundancy of FMSR increases from (12,10) to (4,2) (see Figure 5(b)); it increases from 8.758s to 13.947s when the redundancy of AECC increases from (255,243) to (255,232) (see Figure 5(c)). Figure 5(d) shows that increasing the PRP and PRF block sizes can reduce the DIP encoding time, yet we observe that the overhead reduction of PRF is less prominent than that of PRP. The reason is that we implement PRF based on AES, a block cipher that must be invoked for every 16 bytes (AES block size) of the file. Note that one should not make the PRP block size too large, as an FMSR code chunk is padded to a multiple of ($k' \times$ PRP block size) before being encoded by DIP.

Check. We evaluate the effects of the check block size and the checking percentage on the Check operation. By default, we use the check block size of 256KB and the checking percentage of 1%. We then vary one set of parameters each time in our evaluations.

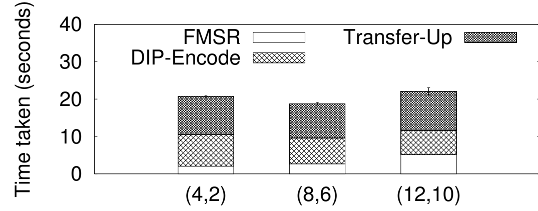
Figure 6 shows the results. The transfer time of downloading data from the local cloud (denoted by "Transfer-Down") dominates the total running time of Check (which includes the computations of PRF and rank checking). Figure 6(a) shows that when the check block size is small, the TCP connection does not have enough time to speed up when downloading each block, resulting in a much longer download time. For instance, the download time for the check block size of 256KB is 3.130s, while that for the check block size of 1KB 21.523s, which is about seven times longer. On the other hand, Figure 6(b) shows that the overall Check time increases with the checking percentage, but in a sub-linear rate. We note that Swift allows a connection session to be reused when downloading data from the same file, so the connection setup overhead has less impact when the download size is large.

Download and Repair. We now measure the total running times of the Download and Repair operations. Here, we only consider the effects of different file sizes, while other parameters use the same default values as in Upload. In the Repair operation, we consider the repair of a single failed server, which we simulate by setting the path of one of the Swift containers to a non-existent location.

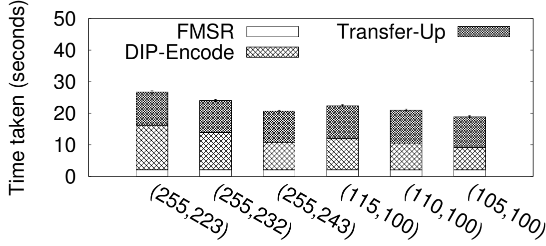
Figures 7 and 8 show the running times of Download and Repair, respectively. In Download, the DIP-Decode part accounts for 4.9% (for 1MB) to 42.2% (for 100MB) for the



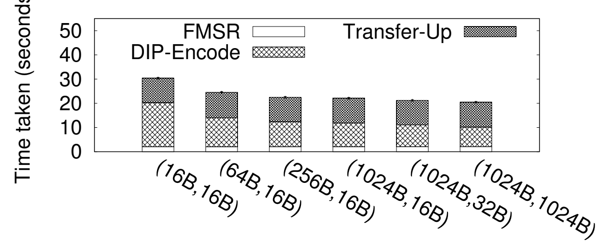
(a) Different file sizes



(b) Different (n, k) values of FMSR

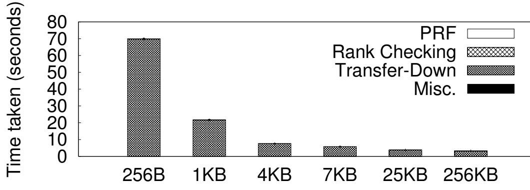


(c) Different (n', k') values of AECC

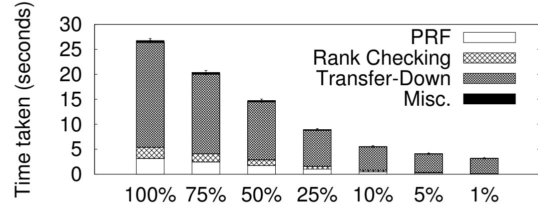


(d) Different block sizes of PRP and PRF

Fig. 5. Running times of the Upload operation on a local cloud for different sets of parameters.



(a) Different check block sizes



(b) Different checking percentage values

Fig. 6. Running times of the Check operation on a local cloud for different sets of parameters.

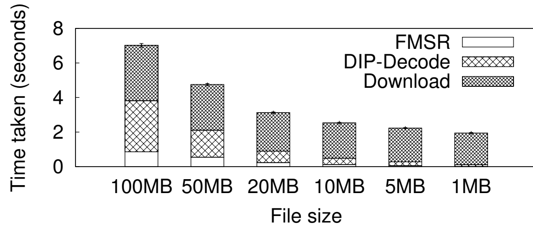


Fig. 7. Running time of the entire Download operation on a local cloud.

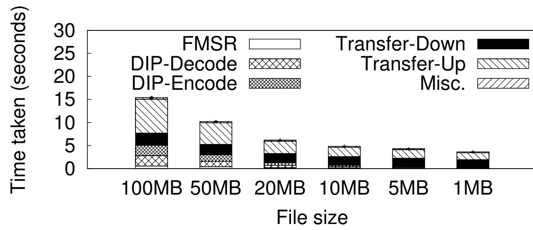


Fig. 8. Running time of the entire Repair operation on a local cloud.

overall Download time, while in Repair, the DIP-Decode and DIP-Encode parts altogether account for 3.4% (for 1MB) to 29.1% (for 100MB) for the overall Repair time.

B. Monetary Cost Analysis

We now describe the monetary overhead of FMSR-DIP in each of the operations compared to the original FMSR implementation in NCCloud [22].

Upload. The major source of the monetary overhead of our DIP scheme compared to NCCloud is (n', k')-AECC, which expands the stored data and increases the storage cost by roughly n'/k' (note that the inbound transfer cost is free for all commercial cloud providers that we consider). The cost due to the expanded file metadata is a negligible constant if the file size is large enough. For example, when using (4,2)-FMSR, our encrypted metadata size is 320B, which is 160B more than the current NCCloud implementation. Furthermore, some cloud providers such as Rackspace and Azure allow a small metadata to be associated with an uploaded object for free.

Check. Since NCCloud does not support the Check operation, we briefly discuss the sources of the Check cost. The Check cost is composed of the download bandwidth cost and the GET request cost. To minimize the download bandwidth cost, we can reduce the checking percentage. To minimize the GET request cost, we can set a larger check block size in order to save on the per-request cost, with a trade-off of less security protection.

Download. When no corrupted data is detected, we do not have to download the AECC parities. Thus, the monetary cost incurred by DIP is similar to NCCloud. Our DIP scheme adds a small constant overhead (independent of the file size) in downloading the metadata, which now has a larger size than the original NCCloud implementation.

Repair. The major monetary overhead again comes from (n', k')-AECC in encoding the new FMSR code blocks. As discussed above, if there is no corrupted data in surviving servers, we preserve the network transfer cost of NCCloud when downloading data from the surviving servers (aside from the small constant metadata traffic). Also, the inbound transfer cost of writing reconstructed FMSR-DIP chunks to a new server is free for many commercial cloud storage providers [22]. Therefore, we still preserve the cost saving property of the repair operation in NCCloud when compared to the conventional repair method (by up to 50% for RAID-6 [22]).

C. Summary

We study how different parameters influence the overhead of FMSR-DIP. For Upload, which is a regular operation in archival storage, the DIP encoding part contributes up to about 40% of the running time with our default parameters. For the monetary cost, the overhead mainly comes from AECC, which expands our stored data by roughly n'/k' times. In particular, we preserve the data transfer cost of NCCloud during repair, which is the most significant advantage of FMSR over traditional erasure codes.

We may further reduce the running time of our DIP scheme. In the evaluations, our DIP scheme runs in single-threaded mode. Since our DIP encoding module works on a per-chunk basis, we can parallelize the DIP encoding operations in multi-threaded mode. We plan to explore the multi-threaded implementation in future work.

VIII. Conclusions

Seeing the popularity of outsourcing archival storage to the cloud, it is desirable to enable clients to verify the integrity of their data in the cloud. We design and implement a 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 strength, evaluate its running time overhead via testbed experiments, and conduct monetary cost analysis. The source code of our DIP implementation is available at: **http://ansrlab.cse.cuhk.edu.hk/software/fmsrdip**.

Acknowledgment

This work was supported by grant AoE/E-02/08 from the University Grants Committee of Hong Kong.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proc. of ACM SoCC*, 2010.
[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans. on Information Theory*, 46(4):1204–1216, Jul 2000.
[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.
[4] Amazon Simple Storage Service. http://aws.amazon.com/s3/.
[5] 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. *Communications of the ACM*, 53(4):50–58, 2010.
[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song. Remote Data Checking Using Provable Data Possession. *ACM Trans. on Information and System Security*, 14:12:1–12:34, May 2011.
[7] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and Efficient Provable Data Possession. In *Proc of SecureComm*, 2008.
[8] G. Ateniese, S. Kamara, and J. Katz. Proofs of Storage from Homomorphic Identification Protocols. In *Proc. of ASIACRYPT*, 2009.
[9] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of ACM EuroSys*, 2011.
[10] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Topics in Cryptology – CT-RSA 2002*, volume 2271 of *LNCS*, pages 114–130. Springer, 2002.
[11] K. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proc. of ACM CCS*, 2009.
[12] K. Bowers, A. Juels, and A. Oprea. Proofs of Retrievability: Theory and Implementation. In *Proc. of ACM CCSW*, 2009.
[13] B. Chen, R. Curtmola, G. Ateniese, and R. Burns. Remote Data Checking for Network Coding-Based Distributed Storage Systems. In *Proc. of ACM CCSW*, 2010.
[14] H. C. H. Chen and P. P. C. Lee. Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage. Technical report, CUHK, 2012.
[15] R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *Proc. of ACM StorageSS*, 2008.
[16] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-Replica Provable Data Possession. In *Proc. of IEEE ICDCS*, 2008.
[17] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.
[18] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of Retrievability via Hardness Amplification. In *Proc. of TCC*, 2009.
[19] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic Provable Data Possession. In *Proc. of ACM CCS*, 2009.
[20] O. Goldreich. *Foundations of cryptography: Basic tools*, volume 1. Cambridge Univ Pr, 2001.
[21] O. Goldreich. *Foundations of cryptography: Basic applications*, volume 2. Cambridge Univ Pr, 2004.
[22] Y. Hu, H. Chen, P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proc. of USENIX FAST*, 2012.
[23] A. Juels and B. Kaliski Jr. PORs: Proofs of Retrievability for Large Files. In *Proc. of ACM CCS*, 2007.
[24] OpenSSL. http://www.openssl.org/.
[25] OpenStack Object Storage. http://www.openstack.org/projects/storage/.
[26] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software - Practice & Experience*, 27(9):995–1012, Sep 1997.
[27] R. Popa, J. Lorch, D. Molnar, H. Wang, and L. Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proc. of USENIX ATC*, 2011.
[28] M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, Apr 1989.
[29] 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.
[30] T. Schwarz and E. Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. In *Proc. of IEEE ICDCS*, 2006.
[31] H. Shacham and B. Waters. Compact Proofs of Retrievability. In *Proc. of ASIACRYPT*, 2008.
[32] M. Vrable, S. Savage, and G. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.
[33] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. In *Proc. of IEEE INFOCOM*, 2010.
[34] Watson Hall Ltd. UK data retention requirements, 2009. https://www.watsonhall.com/resources/downloads/paper-uk-data-retention-requirements.pdf.
[35] zfec. http://pypi.python.org/pypi/.