Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage: Theory and Implementation

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

Abstract—To protect outsourced data in cloud storage against corruptions, adding fault tolerance to cloud storage, along with efficient data integrity checking and recovery procedures, 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 its intrinsic properties of fault tolerance and repair traffic saving. Our DIP scheme is designed under a mobile 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 a 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 further analyze the security strengths of our DIP scheme via mathematical models. 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

1 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 case their data have been accidentally corrupted or maliciously compromised by insider/outsider 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., [28]). Since it is typical to have a huge amount of archived data, whole-file checking becomes prohibitive. *Proof of retrievability* (POR) [16] and *proof of data possession* (PDP) [3] 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 [2] and vendor lock-ins [1]. As suggested in [1], [2], a plausible solution is to stripe data across multiple servers. Thus, to repair a failed server, we can (i) read data from the other surviving servers, (ii) reconstruct the corrupted data of the failed server, and (iii) write the reconstructed data to a new server. POR [16] and PDP [3] are originally proposed for the single-server case. MR-PDP [10] and HAIL [4] extend integrity checks to a multi-server setting using replication and erasure coding respectively. In particular, erasure coding (e.g., Reed-Solomon codes [21]) has a lower storage overhead than replication under the same fault tolerance level.

Field measurements [12], [22], [23] show that large-scale storage systems commonly experience disk/sector failures, some of which can result in permanent data loss. For example, the annualized replacement rate (ARR) for disks in production storage systems is around 2-4% [23]. Data loss events are also found in commercial cloud storage services [18], [26]. With the exponential growth of archival data, a small failure rate can imply significant data loss in archival storage [29]. This motivates us to explore high-performance recovery so as to reduce the window of vulnerability. *Regenerating codes* [11] 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*

- *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)*
- *An earlier conference version of the paper appeared in IEEE SRDS 2012 [7]. In this journal version, we conduct security analysis via mathematical modeling and include more evaluation results.*

over traditional erasure codes? A related approach is HAIL [4], 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 data during 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 protection tailored for regenerating codes.

In this paper, we design and implement a practical data integrity protection (DIP) scheme for regenerating-coding-based cloud storage. We augment the implementation of *functional minimum-storage regenerating (FMSR)* codes [15] and construct *FMSR-DIP* codes, which allow clients to remotely verify the integrity of random subsets of long-term archival data under a multi-server setting. FMSR-DIP codes preserve fault tolerance and repair traffic saving as in FMSR codes [15]. Also, we assume only a thin-cloud interface [27], meaning that servers only need to support standard read/write functionalities. This adds to the portability of FMSR-DIP codes and allows simple deployment in general types of storage services. By combining integrity checking and efficient recovery, FMSR-DIP codes provide a low-cost solution for maintaining data availability in cloud storage.

In summary, we make the following contributions:

- We design FMSR-DIP codes, which enable integrity protection, fault tolerance, and efficient recovery for cloud storage.
- We export several tunable parameters from FMSR-DIP codes, such that clients can make a trade-off between performance and security.
- We conduct mathematical analysis on the security of FMSR-DIP codes for different parameter choices.
- We implement FMSR-DIP codes, and evaluate their overhead over the existing FMSR codes 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.

The rest of the paper proceeds as follows. Section 2 reviews related work. Section 3 provides necessary preliminaries for our DIP design. Section 4 presents the design of FMSR-DIP codes. Section 5 presents the implementation details and discusses how parameters can be adjusted for different performance needs. Section 6 presents security analysis for FMSR-DIP codes. Section 7 reports evaluation results of FMSR-DIP codes in a cloud storage testbed. Finally, Section 8 concludes the paper. We also refer readers to our digital supplementary file for additional discussion of this work.

2 Related Work

We briefly summarize the most recent and closely related work here. Further literature review can be found in Section 1 of the supplementary file.

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. [16] and Ateniese et al. [3], giving rise to the similar notions *proof of retrievability* (POR) and *proof of data possession* (PDP) respectively. 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 [10], erasure coding [4], [24], and regenerating coding [6].

Specifically, although Chen et al. [6] also consider regenerating-coded storage, there are key differences with our work. First, their design extends the single-server compact POR scheme by Shacham et al. [25]. 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 [25]. Second, the storage scheme of [6] assumes that storage servers have encoding capabilities for generating encoded data, while we consider a thin-cloud setting [27] where servers only need to support standard read/write functionalities for portability and simplicity. The most closely related work to ours is HAIL [4], which stores data via erasure coding. As stated in Section 1, HAIL operates on a per-file basis and it is non-trivial to directly apply HAIL to regenerating codes. In addition, our work focuses more on the practical issues, such as how different parameters can be adjusted for the performance-security trade-off in practical deployment.

3 Preliminaries

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

3.1 FMSR Code Implementation

We first review FMSR codes in NCCloud [15], on which our DIP scheme is developed. FMSR codes belong 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 codes is that a specific piece can be reconstructed from data of size less than $|F|$. FMSR codes are built on *regenerating codes* [11], which minimize the repair traffic while preserving the MDS property.

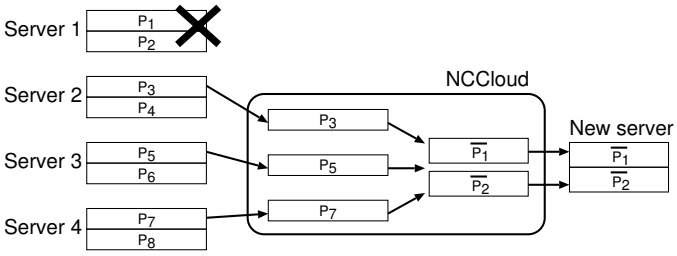


Fig. 1. Example of how a file is repaired in (4,2)-FMSR codes. Each of the code chunks $P_1, \cdots, P_8$ is a random linear combination of the native chunks. $\overline{P_1}$ and $\overline{P_2}$ are distinct random linear combinations of P_3, P_5 and P_7.

We consider a distributed storage setting in which a file is striped over n servers using an (n, k)-FMSR code. Each server can be a storage site or even a cloud storage provider, and is assumed to be independent of other servers. 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 [20]. The $n(n-k)$ code chunks are stored in 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 the encoding matrix [19].

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 read from the other surviving servers, so as to reconstruct the lost data. We assume that there is a *proxy* (NCCloud in our case) that handles the entire repair operation.

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 codes, 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 in 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 [15]. Figure 1 illustrates the repair operation for the (4,2)-FMSR code, in which the repair traffic is reduced by 25% to $0.75|F|$. It is shown that the repair traffic of FMSR codes can be further reduced to 50% for $k = n-2$ if n is large [15].

Note that FMSR codes are *non-systematic codes* that keep only code chunks rather than native chunks. Nevertheless, FMSR codes are mainly designed for long-term archival applications, where the read frequency is low and each read operation typically restores the entire file. In Section 2 of the supplementary file, we provide additional illustrations of how FMSR codes work.

3.2 Threat Model

We adopt the adversarial model in [4] 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 corrupted by 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.

3.3 Cryptographic Primitives

Our DIP scheme is built on several cryptographic primitives, whose detailed descriptions can be found in [13], [14]. 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)* [5], [9] 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. Both FMSR codes and AECC provide fault tolerance. The difference is that we apply FMSR codes to a file striped across servers, while we apply AECC to a single code chunk stored within a server.

4 Design

We present our design of DIP atop FMSR codes, and we call the augmented coding scheme *FMSR-DIP* codes. Please refer to Section 3 of the supplementary file for a summary of notations and an illustration of how FMSR-DIP code chunks are formed from FMSR code chunks.

4.1 Design Goals

We first state the design goals of FMSR-DIP codes.

Preserving regenerating code properties. We preserve the fault tolerance and repair traffic saving of FMSR codes, with up to a small constant overhead.

Thin-cloud storage [27]. Each server (or cloud storage provider) only needs to provide a basic interface for clients to read and write their stored files. No computation capabilities are required from the servers 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 uses only the PUT and GET commands to interact with each server.

Our thin-cloud setting enables our DIP scheme to be portable to general types of storage devices or services, since no implementation changes are required on the storage backend. It differs from other "thick" cloud storage services where servers have computational capabilities and are capable of aggregating the proofs of multiple checks (e.g., [3], [4]). Nevertheless, we address how our approach can be extended to thick cloud storage services in Section 4 of the supplementary file.

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 grow less important over time. Such flexibility should come without any additional penalties.

4.2 Notations

We now define notations for FMSR-DIP codes, based on the FMSR codes described in Section 3.1. 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 3.3, 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.

We define a *row* as a collection of all bytes that are at the same offset of all native chunks or FMSR code chunks. The rth rows of the native chunks and the FMSR code chunks correspond to the bytes $\{F_{jr}\}_{1\leq j\leq k(n-k)}$ and $\{P_{ir}\}_{1\leq i\leq n(n-k)}$, respectively. We can see that *the rth row of each of the FMSR code chunks is encoded by the rth row of the native chunks*. That is, we can construct the rth row of the FMSR code chunk P_i, where $1 \leq i \leq n(n-k)$, as follows: $P_{ir} = \sum_{j=1}^{k(n-k)} \alpha_{ij} F_{jr}$.

Each FMSR code chunk P_i from NCCloud is encoded by FMSR-DIP codes into P'_i. The rth row of the FMSR-DIP code chunks corresponds to the bytes $\{P'_{ir}\}_{1\leq i\leq n(n-k)}$. Figure 2 shows the layout of the FMSR-DIP code chunks based on the $(4, 2)$-FMSR code.

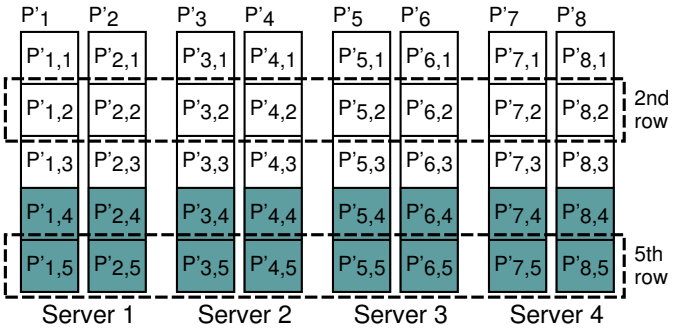


Fig. 2. Integration of DIP into the (4,2)-FMSR code. In this example, each FMSR code chunk P_i is of size three bytes. FMSR-DIP codes encode each chunk with (5,3)-AECC to give $\{P'_i\}$, so bytes $P'_{i,4}$ and $P'_{i,5}$ correspond to the AECC parities of the ith chunk. Then $P'_{1,2}, P'_{2,2}, \cdots,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

4.3 Basic Operations of FMSR-DIP Codes

Our goal is to augment the basic file operations Upload, Download, and Repair of NCCloud with the DIP feature. During Upload, FMSR-DIP codes expand the code chunk size by a factor of n'/k' (due to the AECC). During Download and Repair, FMSR-DIP codes maintain 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. In the following, we assume that FMSR-DIP codes operate in units of *bytes*. In Section 5, we discuss how we relax this assumption to trade security for performance.

4.3.1 Upload Operation

We first describe how we upload a file F to the servers.

Step 1: Generate the per-file secrets. Before uploading F, we generate per-file secrets κ_{ENC}, κ_{PRP}, κ_{PRF}, and κ_{MAC}. All secret keys can be securely stored on the client side without being revealed to any server. To reduce the key management overhead, we can derive multiple keys from a single secret using key derivation functions, as detailed in prior studies and standards (e.g., [8], [17]). In addition, to relieve the local storage burden, we can encrypt all file keys with a master key, and outsource the storage of the encrypted keys to the cloud. Since the files in the cloud are typically of large size, we expect that the secret keys only incur a small constant overhead.

Step 2: Encode the file using FMSR codes. NCCloud encodes F using the (n, k)-FMSR code to generate $n(n-k)$ code chunks $\{P_i\}$ of b bytes each, where $b = \frac{|F|}{k(n-k)}$. It also outputs a metadata file containing the file size $|F|$ and encoding coefficients $\{\alpha_{ij}\}$.

Step 3: Encode each code chunk with FMSR-DIP codes. Consider the ith code chunk P_i. We first apply AECC to the b bytes of the code chunk $\{P_{ir}\}_{1\leq r\leq b}$ to generate $b'-b$ parity bytes $\{P_{ir}\}_{b+1\leq r\leq b'}$, 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 codes alone. We apply the same AECC to each of the code chunks (i.e., with the same permutations and encoding parameters).

Then we apply 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: $M_i=\mathsf{MAC}(P'_{i1}||\ldots||P'_{ib})$, where $||$ denotes 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 upload the FMSR-DIP code chunks P'_i's to their respective servers. Then, we append n' and k' to the metadata file generated by NCCloud. We also append the MACs of all chunks to the metadata. Finally, the metadata is encrypted with κ_{ENC} and replicated to each server (it contributes to only a small storage overhead).

4.3.2 Check Operation

In the Check operation, we verify randomly chosen rows of bytes of the currently stored chunks in the servers.

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 code 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 $\{P'_{ir}\}_{1\leq i\leq n(n-k)}$, 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 $\mathbf{P}=[P_{ir}]_{1\leq i\leq n(n-k)}$. 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 codes, 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 it difficult for the adversary 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. As encoding coefficients are assumed to be correct, the MDS property of the FMSR code guarantees that $\mathbf{rank}(\tilde{\mathbf{A}})=k(n-k)$. In other words, 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 this enumeration approach is also used by HAIL [4].

The above error localization step assumes at most $n-k-1$ failed servers. If $n-k$ servers fail, we may recover the errors by downloading the full FMSR-DIP code chunks, as discussed in the Download operation.

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).

4.3.3 Download Operation

We download a file F from the servers as follows.

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

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

4.3.4 Repair Operation

If some server fails (e.g., when losing all data or having too much corrupted data that cannot be recovered), 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 3.2). 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 codes (see Figure 1 in Section 4.2).

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 codes 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).

5 Implementation

We implement FMSR-DIP codes atop NCCloud [15]. In this section, we address how our implementation can fine-tune various design parameters to trade security for performance. Please refer to Section 5 of the supplementary file for additional implementation details on how we integrate FMSR-DIP codes into NCCloud and how we instantiate the cryptographic primitives.

In Section 4, FMSR-DIP codes operate 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 codes can be extended to 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 codes.

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 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.

6 Security Analysis

In this section, we investigate the security guarantees of FMSR-DIP codes. In Section 6 of the supplementary file, we discuss how the security primitives affect the design of FMSR-DIP codes.

6.1 Security Guarantees

We now provide a step-by-step security analysis of FMSR-DIP codes.

Attack goal. Recall that an FMSR code chunk is encoded by (n', k')-AECC. The code chunk is divided into k' fragments and b/k' stripes. In our implementation, 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. 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. Our objective is to provide a mathematical framework that analyzes the security strength of FMSR-DIP codes for different parameter choices.

Attack approach. We assume that the PRPs are ideal (i.e., output random permutations), so the adversary can do no better than corrupt randomly within a fragment. We consider a set of adversarial corruption strategies of the following form: given an overall corruption rate p, the adversary corrupts only the first i fragments and spreads the corruptions evenly among these i fragments, for $i \in ((n'-k')/2, n']$. For example, with (110,100)-AECC, we have 105 corruption strategies. We refer to a strategy as Strategy i, in which the adversary corrupts only the first i fragments with rate $p_i = pn'/i$.

Our attack approach enables us to study two questions: (i) how FMSR-DIP codes can remain secure (i.e., no corrupted stripes can evade detection by our Check operation in each epoch under our threat model (see Section 3.2)) subject to different corruption rates, and (ii) how the design parameters of FMSR-DIP codes can be fine-tuned to address the performance-security trade-off. We do not know if there exists a better attack approach, among all possible corruption approaches that can be arbitrarily taken by an adversary. We pose the study of different possible corruption approaches as future work.

6.1.1 Corrupting an AECC Stripe

We first consider the case where the PRP block size is fixed at 1 byte. Considering only a single stripe, the number of corrupted bytes is governed by a binomial distribution, according to the corruption strategies that we define above. Denote the number of stripes as $N = b/k'$ and let $\mathcal{C}_i$ be the event that at least one stripe is corrupted and made unrecoverable using Strategy i. We now have

$$\Pr(\mathcal{C}_i) \leq N \times \left(1 - \sum_{j=0}^{\lfloor (n'-k')/2 \rfloor} \binom{i}{j} p_i^j (1-p_i)^{i-j}\right), \quad (1)$$

where the right hand side is obtained using the union bound and it approximates the actual value when the corruption rate p is low.

Now we consider the effect of the PRP block size. Denote B_P as the PRP block size. Instead of corrupting randomly within a fragment, the adversary can now do better by corrupting a specific offset in randomly chosen PRP blocks. For a stripe that is at this chosen offset, this means that the adversary can achieve an effective corruption rate $p \times B_P$, and there are N/B_P stripes in total at a specific offset of any PRP block. We can revise $\Pr(\mathcal{C}_i)$ to account for this.

$$\Pr(\mathcal{C}_i) \leq \frac{N}{B_P} \times \left(1 - \sum_{j=0}^{\lfloor (n'-k')/2 \rfloor} \binom{i}{j} (p_i B_P)^j (1-p_i B_P)^{i-j}\right). \quad (2)$$

6.1.2 Picking Bytes for Checking

Next we bound the probability that no corrupted bytes get picked during Check. Denote the check block size as B_C and the checking percentage as λ. Let $\mathcal{E}_i$ be the event that all corrupted bytes do not get picked during Check when Strategy i is used. For simplicity, we assume that the check blocks do not span across fragments. For each check block, the probability that it lands on a corrupted fragment is i/n'; for any fragment with corruption rate p_i, the probability that none of the corrupted bytes collide with the check block is $\prod_{x=0}^{p_i N-1} \frac{N-B_C-x}{N-x}$, which is approximately $((N-B_C)/N)^{p_i N}$ for small p_i (upper bound). Since the number of check blocks is $\lambda N n'/B_C$, we have

$$\Pr(\mathcal{E}_i) \leq \left(1 - \frac{i}{n'}\left(1 - \left(\frac{N-B_C}{N}\right)^{p_i N}\right)\right)^{\frac{\lambda N n'}{B_C}}. \quad (3)$$

This is an over-estimation especially when λ is not negligible compared to the code chunk size (say $\lambda > 5\%$). To see why, consider the case that two check blocks fall into the same corrupted fragment. When we place the second check block, the probability that none of the corrupted bytes collide with it is $\prod_{x=0}^{p_i N-1} \frac{N-2B_C-x}{N-B_C-x}$ instead of $\prod_{x=0}^{p_i N-1} \frac{N-B_C-x}{N-x}$ due to an existing check block taking up space.

Intuitively, the adversary has a higher evasion rate $\Pr(\mathcal{E}_i)$ if the corrupted bytes are more concentrated (e.g., in the first few fragments) and if we use a larger check block size. The reason is that when the corrupted bytes are more concentrated, a larger check block can capture more corrupted bytes, but at the same time we use fewer check blocks.

6.2 Putting It All Together

Let $\mathcal{S}_i$ be the event that the adversary uses Strategy i to successfully make at least one stripe unrecoverable without being detected by row checking. Then, $\Pr(\mathcal{S}_i) \leq \Pr(\mathcal{C}_i) \times \Pr(\mathcal{E}_i)$, which we can compute via Equations (2) and (3). Note that if i increases, the corruption rate $\Pr(\mathcal{C}_i)$ increases, but the evasion rate $\Pr(\mathcal{E}_i)$ also decreases. To search for the *maximum* $\Pr(\mathcal{S}_i)$, we enumerate different values of the corruption rate p for each Strategy i, and enumerate all possible strategies. In the following, we study how the maximum $\Pr(\mathcal{S}_i)$ varies for different parameter choices.

Scenario 1 (Impact of different values of checking percentage λ). First, we consider an FMSR code chunk of size 4MB, $n' = 110$, $k' = 100$, $B_P = 16$, $B_C = 16$, and different values of λ. Figure 3 shows that the maximum

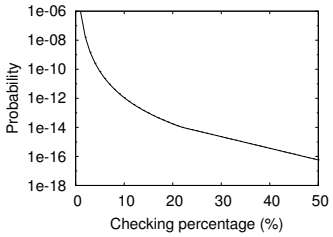


Fig. 3. Scenario 1: Max. $\Pr(\mathcal{S}_i)$ (in log scale) versus checking percentage λ.

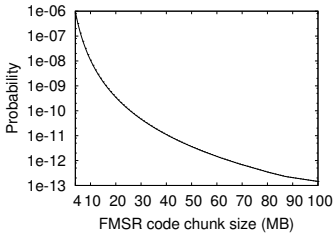


Fig. 4. Scenario 2: Max. $\Pr(\mathcal{S}_i)$ (in log scale) versus FMSR code chunk size.

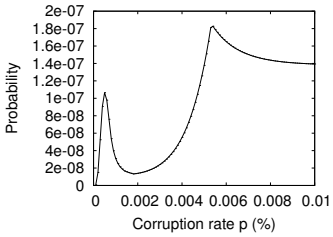


Fig. 5. Scenario 3: Max. $\Pr(\mathcal{S}_i)$ versus corruption rate p.

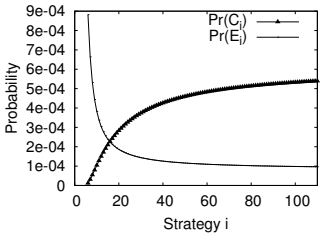


Fig. 6. Scenario 3: $\Pr(\mathcal{C}_i)$ and $\Pr(\mathcal{E}_i)$ versus Strategy i for $p = 0.0008\%$.

$\Pr(\mathcal{S}_i)$ is less than 10^{-6} when $\lambda = 1\%$, and it decreases as the checking percentage λ increases. As we explain above that Equation (3) is an over-estimation, we expect that the actual maximum $\Pr(\mathcal{S}_i)$ can be even smaller.

Scenario 2 (Impact of different FMSR code chunk sizes). It is beneficial to aggregate small files before backing them up to the cloud to reduce monetary and transfer overheads. For example, Cumulus [27] uploads chunks of size roughly 4MB by default [1]. Here, we consider the parameters $n' = 110$, $k' = 100$, $B_P = 16$, $B_C = 16$, $\lambda = 1\%$, and different values of FMSR code chunk sizes. Figure 4 shows the maximum $\Pr(\mathcal{S}_i)$ for different chunk sizes, and we find that the maximum $\Pr(\mathcal{S}_i)$ decreases as the chunk size increases. For example, when we increase the chunk size to 25MB, the maximum $\Pr(\mathcal{S}_i)$ is on the order of 10^{-10}, and if we further increase the chunk size to 100MB, the maximum $\Pr(\mathcal{S}_i)$ is on the order of 10^{-13}. The reason is that as the chunk increases, more bytes will be corrupted for the same corruption rate (i.e., $\Pr(\mathcal{C}_i)$ increases), but also more bytes will be sampled for checking for a given checking percentage (i.e., $\Pr(\mathcal{E}_i)$ decreases). We see that $\Pr(\mathcal{E}_i)$ decreases at a rate faster than the increase in $\Pr(\mathcal{C}_i)$, leading to a drop in the maximum $\Pr(\mathcal{S}_i)$. As a result, we can afford to use more efficient parameters (e.g., larger PRP block size and check block size) while maintaining acceptable security.

Scenario 3 (Impact of different values of corruption rate p). We now study how different values of p affect the $\Pr(\mathcal{S}_i)$. This time for a given value of p, we enumerate different strategies and select the strategy that gives the maximum $\Pr(\mathcal{S}_i)$. We consider an FMSR code chunk of size 100MB, $n' = 110$, $k' = 100$, $B_P = 256$, $B_C = 4096$, $\lambda = 1\%$. Then we have $N = 1,048,576$ and the adversary succeeds if he can corrupt 6 bytes in any stripe without being detected. Figure 5 plots the maximum $\Pr(\mathcal{S}_i)$ by varying the corruption rate p from 0.001% (i.e., about one corrupted byte per fragment on average) to 0.01%. Note that the upper bound of $\Pr(\mathcal{C}_i)$ increases beyond 1 when $p \geq 0.0054\%$, so we set the upper bound to 1 in such cases. We note that the maximum $\Pr(\mathcal{S}_i)$ shows a small peak at $p = 0.0008\%$, mainly because of the shift of the optimal strategy from Strategy 110 to Strategy 6. We note that the maximum $\Pr(\mathcal{S}_i)$ increases for $0.002\% \leq p \leq 0.0054\%$, and then decreases because $\Pr(\mathcal{C}_i)$ is upper-bounded by one while $\Pr(\mathcal{E}_i)$ continues to drop. From Figure 5, we note that $\Pr(\mathcal{S}_i)$ is less than 2×10^{-7} for all values of p. Figure 6 also plots the changes of the upper bounds of $\Pr(\mathcal{C}_i)$ and $\Pr(\mathcal{E}_i)$ across different strategies when $p = 0.0008\%$.

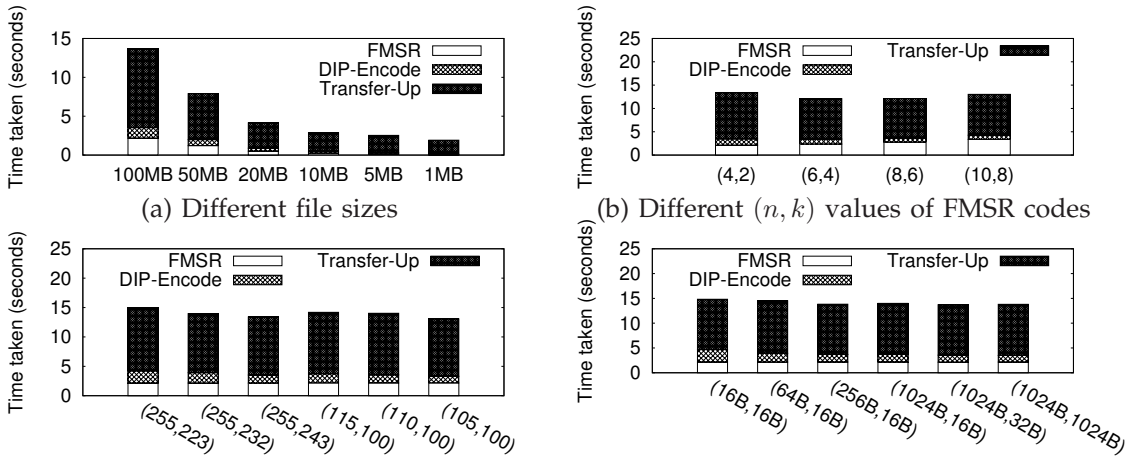
7 Evaluations

We evaluate the running time overhead of FMSR-DIP codes atop a local cloud storage testbed by measuring the running time overhead of DIP in the Upload and Check operations. The goal of our evaluation is to understand the overhead of FMSR-DIP codes over the original implementation of FMSR codes in NCCloud [15]. In Section 7 of the supplementary file, we present results for the Download and Repair operations, and also analyze the monetary cost overhead with the pricing models of different commercial cloud providers.

Setup. We conduct testbed experiments on a local cloud platform that is built on OpenStack Swift 1.4.2. We deploy FMSR-DIP on a machine equipped with two Intel Xeon E5530 Quad-Core CPUs (i.e., a total of eight cores), 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 measure 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 codes in normal usage. Our results are averaged over 40 runs.

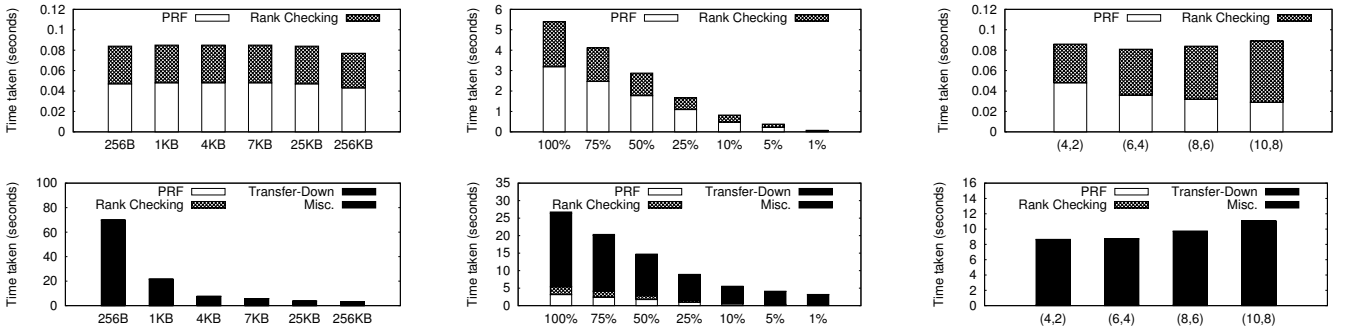
We exploit parallelism in our implementation. We spawn a DIP process for processing each FMSR-DIP code chunk (i.e., encoding an FMSR code chunk into an FMSR-DIP code chunk, or decoding an FMSR-DIP code chunk into an FMSR code chunk). All DIP processes are executed concurrently on our eight-core testbed. Our parallel implementation can achieve up to eight-fold speedup compared to our sequential evaluations depending on the number of chunks that needed to be processed (e.g., from Figure 7(a), encoding an 100MB file with default parameters takes 1.355s, compared to 8.547s re-running the same tests in sequential mode). For the baseline evaluations of FMSR-DIP codes in sequential mode, we refer readers to our conference paper [7].



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

(c) Different (n', k') values of AECC (d) Different block sizes of PRP and PRF

Fig. 7. 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 (c) Different (n, k) values of FMSR codes

Fig. 8. Running times of the Check operation on a local cloud for different sets of parameters (the PRF and Rank Checking times shown on the graphs in the upper row correspond to those of the graphs in the lower row).

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 codes, (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 code, (110,100)-AECC, and a block size of 256B for both PRP and PRF.

Figure 7 plots the running times of the Upload operation for different sets of parameters. 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 code chunks by NCCloud, (ii) "DIP-Encode", the time of encoding the FMSR code chunks with our DIP scheme, and (iii) "Transfer-Up", the network transfer time of uploading FMSR-DIP code chunks and metadata to the local cloud.

From Figure 7(a), we observe that the fractional overhead of DIP encoding increases with the file size, and it ranges from 3.76% (for 1MB) to 9.92% (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 to be 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 7(b) and 7(c), the DIP encoding time increases with the redundancy level (i.e., the ratio of the amount of the redundant data being stored to that of the original data) of each of the underlying FMSR codes and AECC. For instance, the DIP encoding time increases from 0.893s to 1.346s when the redundancy of FMSR codes increases from (10,8) to (4,2) (see Figure 7(b)); it increases from 1.378s to 2.081s when the redundancy of AECC increases from (255,243) to (255,223) (see Figure 7(c)). This is expected, as we now need to protect more stored data with DIP.

Figure 7(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 (i) the check block size, (ii) the checking percentage, and (iii) the (n, k) parameters of FMSR codes in the Check operation. Apart

from the default parameters in the evaluations of the Upload operation, we also use a check block size of 4KB and a checking percentage of 1% by default. In evaluating the effect of checking percentage, we use a check block size of 256KB.

Figure 8 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 8(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 8(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 to be reused when downloading data from the same file, so the connection setup overhead has less impact when the download size is large. This effect is also observed in Figure 8(c), where the download time increases with n (from 8.527s for (4,2)-FMSR to 10.950s for (10,8)-FMSR). The main reason for the increase is that more connections have to be established to download data and metadata from more chunks.

8 Conclusions

Given 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 the functional minimum-storage regenerating (FMSR) codes under a multi-server setting. We construct FMSR-DIP codes, which preserve the fault tolerance and repair traffic saving properties of FMSR codes. We analyze the security strength via mathematical modeling and evaluate the running time overhead via testbed experiments. We show how FMSR-DIP codes trade between performance and security under different parameter settings. The source code of the implementation of our FMSR-DIP codes is available at: **http://ansrlab.cse.cuhk.edu.hk/software/fmsrdip**.

Acknowledgments

This work is supported by grants from the University Grants Committee of Hong Kong (AoE/E-02/08 and ECS CUHK419212) and seed grants from the CUHK MoE-Microsoft Key Laboratory of Human-centric Computing and Interface Technologies.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proc. of ACM SoCC*, 2010.

[2] 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. of the ACM*, 53(4):50–58, 2010.

[3] 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.

[4] K. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proc. of ACM CCS*, 2009.

[5] K. Bowers, A. Juels, and A. Oprea. Proofs of Retrievability: Theory and Implementation. In *Proc. of ACM CCSW*, 2009.

[6] 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.

[7] H. C. H. Chen and P. P. C. Lee. Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage. In *Proc. of IEEE SRDS*, 2012.

[8] L. Chen. NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions (Revised). http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf, Oct 2009.

[9] R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *Proc. of ACM StorageSS*, 2008.

[10] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-Replica Provable Data Possession. In *Proc. of IEEE ICDCS*, 2008.

[11] 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.

[12] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[13] O. Goldreich. *Foundations of cryptography: Basic tools*, volume 1. Cambridge Univ Pr, 2001.

[14] O. Goldreich. *Foundations of cryptography: Basic applications*, volume 2. Cambridge Univ Pr, 2004.

[15] 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.

[16] A. Juels and B. Kaliski Jr. PORs: Proofs of Retrievability for Large Files. In *Proc. of ACM CCS*, 2007.

[17] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proc. of CRYPTO*, 2010.

[18] E. Naone. Are We Safeguarding Social Data? http://www.technologyreview.com/blog/editors/22924/, Feb 2009.

[19] 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.

[20] M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, Apr 1989.

[21] 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.

[22] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *Proc. of USENIX FAST*, Feb 2010.

[23] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of USENIX FAST*, Feb 2007.

[24] T. Schwarz and E. Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. In *Proc. of IEEE ICDCS*, 2006.

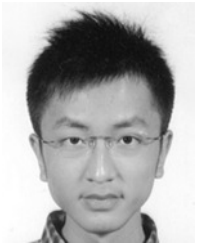
[25] H. Shacham and B. Waters. Compact Proofs of Retrievability. In *Proc. of ASIACRYPT*, 2008.

[26] TechCrunch. Online Backup Company Carbonite Loses Customers' Data, Blames And Sues Suppliers. http://techcrunch.com/2009/03/23/online-backup-company-carbonite-loses-customers-data-blames-and-sues-suppliers/, Mar 2009.

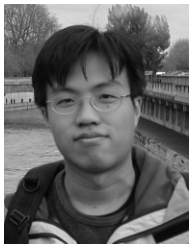
[27] M. Vrable, S. Savage, and G. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.

[28] Watson Hall Ltd. UK data retention requirements, 2009. https://www.watsonhall.com/resources/downloads/paper-uk-data-retention-requirements.pdf.

[29] A. Wildani, T. J. E. Schwarz, E. L. Miller, and D. D. Long. Protecting Against Rare Event Failures in Archival Systems. In *Proc. of IEEE MASCOTS*, 2009.



Henry C. H. Chen received his B.Eng. in Computer Engineering and M.Phil. in Computer Science and Engineering from the Chinese University of Hong Kong in 2010 and 2012 respectively. He is now a research assistant at the Chinese University of Hong Kong. His research interests are in security and applied cryptography.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an assistant professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including cloud storage, distributed systems and networks, operating systems, and security/resilience.