PoR^2 – Proofs of Retrievability and Redundancy

Ludovic Barman

School of Computer and Communication Sciences

Master Thesis

August 2015

Supervisor
Dr. Ghassan Karame
NEC Laboratories

Supervisor
Dr. Philippe Oechslin
EPFL / LASEC

Abstract

In this work, we analyze the issue of data replication in the cloud. Namely, today's cloud provider do not take any liability for data loss, and do not provide means for users e.g., to verify that their files are correctly stored, that their files are appropriately replicated, etc. Several cryptographic protocols, Provable Data Possession and Proof of Retrievability, allow to check efficiently the integrity of a remote file; recent work have extended those protocols to include multiple replicas, thus allowing to strongly assert the risk of data loss. However, a common trait of those schemes is that the user has to create and send multiple replicas. Not only this incurs a significant burden on the users, but this does not allow the service provider to verify that the uploaded content corresponds to replicas, which might allow malicious clients to abuse the system and upload data at the cost of replica storage. We argue that this fact provides strong disincentives to cloud providers from adopting such protocols.

In this work, we address this problem, and we explore the solution space for constructing efficient and secure proofs of retrievability and replication in the cloud. Our constructs provide assurance to users that their files are retrievable in their entirety, *and* that they are appropriately replicated as agreed. In this respect, we discuss, analyze, and implement a number of constructs, for proofs of retrievability and replication in the cloud, in which the service provider replicates the files by himself, thus abiding by the existing cloud model. We implement a prototype of our constructs in a realistic cloud setting, and compare their performance.

Acknowledgments

Foremost, I would like to express my sincere gratitude to Dr. Ghassan Karame, my supervisor at NEC Laboratories Europe, for his continuous support and his expertise. His guidance during the whole project was greatly appreciated, and has been essential numerous times. I also thank Dr. Frederik Armknecht and Dr. Jens-Matthias Bohli for their collaboration in this project.

Contents

Chapter 1

Introduction

1.1 Motivation

As confirmed by being in the top three of technologies spending increase in 2015 [18], cloud computing technologies have been and are still gaining traction; both business and private uses of the cloud are expected to keep increasing [19], and rightly so : the solution offers many advantages for small and medium entreprises (such as reduced IT costs and increased collaboration), as well as for private users (easy way to backup and share data).

In particular, cloud-based storage is extremely appealing for mobile devices users, as it reduces the storage cost and simplifies the data accessibility, backup and synchronization between multiple devices. As more and more users rely on cloud services for SaaS and file storage [30], data availability become a critical concern; mobile users are completely dependent of the availability of the service to access the data, as often the personal device of the user does not have enough storage to keep a local copy of all the data.

The solution to this problem is redundancy: to reduce the risk of data loss in case of hardware failure, of malware attacks, or any hazard to the infrastructure, cloud provider often rely on a network of interconnected servers, possibly in different geographic areas [3], to replicate and serve the data; should one of them fail, the data remains accessible through the replicas.

In popular cloud services (Dropbox, SkyDrive, iCloud, etc.), data replication is often controlled entirely by the cloud provider, and no or very little liability is accepted for data loss [24] by these providers. This leaves the users in an uncomfortable place where they cannot really assess, nor control the risks they incur. For example, Amazon S3 provides redundancy by default [2], and proposes a cheaper version with less redundancy [1]. In addition, it charges less for storing replicas than for original data [4]; unfortunately, in Amazon S3, users have no way of verifying the replication undergone by his data and are forced to blindly believe them.

To palliate to this problem, Ateniese et al. [7] introduced in 2007 the first Provable Data Possession (PDP) protocol, which enables a user to verify that his data is correctly and entirely stored in the cloud. Numerous solutions have been proposed after that [6, 8, 9, 12, 15, 21]. Examples include Proofs of Retrievability (POR) [15, 21], which in addition to checking that the data is completely and correctly stored, allow the user to reconstitute the data given a number of successful run of the protocol.

Originally designed to work with a single file, those protocols have recently been extended to include file replicas [15, 17]; those new schemes allow to efficiently verify that the cloud

provider is storing R replicas of the file, hence asserting the redundancy level of the file. However, all those solutions share a common system model where the users need to create and send themselves the R replicas to the cloud. For instance, to store a 1 GB file with a redundancy of 3, the users needs to process and upload around 3 GB of data. Notice that this might not be workable for a significant fraction of users who are equipped low-power devices and limited bandwidth.

Even worse, since the replicas are generated by the user (and this process involves the reliance on secret keys that are only known to the user), the server cannot verify that the received data are indeed replicas. This allows a malicious user to abuse this system, and upload some other meaningful data *instead* of the replicas. This misbehavior is especially beneficial when the cloud provider charges less for storing a replica that for storing an "original file" of the same size, as Amazon S3 does [2, 4]. These limitations constitute strong incentives for cloud provider *not* to implement existing multi-replica PDP/POR schemes.

1.2 Goals

In this work, we address the aforementioned limitations, and we explore the solution space to construct efficient proofs of retrievability and replication in the cloud. Namely, our goal is to design proofs which enable users to check *(1)* the retrievability of their stored files and *(2)* the correctness of the data replication performed by the cloud. Here, we stress that any practical solution should conform with the current cloud model where the cloud provider is involved in the data replication process. Users, and rightly so, should only process and upload once, irrespective of the replication performed by the cloud provider on their data.

1.3 Contributions

In this thesis, we make the following contributions:

- We analyze the solution space for constructing proofs of retrievability and replication. In this respect, we sketch three novel solutions to construct such proofs.
- We analyze the security and performance of our proposals, and we extract relevant lessons with respect to the design of performant and secure proofs of retrievability and replication.
- By leveraging our observations, we devise and analyze a fourth solution, Mirror, which establishes a strong balance between the security and performance of the system.
- We evaluate a prototype implementation of all four devised solutions in a realistic cloud setting, and we compare the performance of these solutions to existing multi-replica PDP schemes in the area, such as the one due to Curtomla et al. [15].

1.4 Thesis Structure

This document is structured as follow : Chapter 2 gives a survey of the existing work in the area, and discusses the different techniques used. Chapter 3 introduces the model, the adversary and the various building blocks used to design solutions; Chapter 4 presents and analyze four proposed protocols. In chapter 6 we discuss the security aspects of the final protocol, and we compare it against a reference scheme in Chapter 7.

Chapter 2

Background and Related work

Introduction. In this chapter, we will present the relevant related work for Provable Data Possession (PDP) and Proof of Retrievability (POR). Those related work will give capital insights on how to construct our solution Mirror.

Both PDP and POR protocols offer ways to check that a pre-agreed file F is stored on a remote location. Where PDP only proves that the file exists and was not altered on the remote location, POR adds the possibility to extract the file given a number of successful runs of the verification protocol; in that sense, POR gives stronger guarantees than PDP.

Definitions:

PDP. Provable Data Possession. Aims to validate the integrity of outsourced data efficiently and securely.

POR. Proof of Retrievability. Aims to provide evidence that a verifier can retrieve and reconstruct the entire outsourced data.

Common architecture. All the following schemes uses the same architecture : a *client*, or *user*, wants to store a file F on the cloud. For this, he contacts a *cloud services provider*, *service provider*, or simply *server*. This service provider owns a collection of storage locations in diverse geographical areas; depending on the user needs, the file F and possible *replicas* of F are to be stored by the server in a subset of its storage location.

Protocols. Before anything, we consider that the environment runs a protocol KeyGen which distributes the appropriates keys to the different parties; in particular, both the client and the server have a way to authenticate each other.

The user starts by running a protocol Store, which aims to prepare F and send it to the server, possibly along additional information such as *authentication tags*; in that case, the user generally keeps the tags and deletes the file F afterwards. In some cases, the user can run a protocol Replicate which creates one replica $F^{(r)}$ from F. In other cases, the server has access to a similar protocol Replicate which has the same aim.

Once the Store (and the possible multiple instances of Replicate) have all finished, the server ought to hold the file F, some replicas $F^{(1)} \cdots F^{(r)}$, possibly some additional information. The

user checks that this statement is true by running a classical three way challenge-response protocol called VERIFY with the server, composed of the procedures CHALLENGE, ANSWERCHALLENGE, USERVERIFY.

Table 2.1 Procedures invokes in a standard PDP / POR protocol

Client		Server
STORE	$\xrightarrow{F, \{\Gamma_i\}}$	
REPLICATE	$\xrightarrow{<OR>}$	REPLICATE
CHALLENGE	$\xrightarrow{Q}$	ANSWERCHALLENGE
USERVERIFY	$\xleftarrow{\quad}$	

Extractability In addition, we present the concept of *extractability*, strongly tied to POR protols. Shacham et al. [29] first introducted the notion of POR : the authors advocate that there should be strong guarantees that if a prover answers successfully a non-negligible fraction ϵ of execution of the VERIFY protocol, then there exists an algorithm called EXTRACTOR, runnable by the client, that recovers the file (except with negligible probability). The authors demonstrate that this algorithm exists for their scheme, and that asymptotically, it runs in $O(n/\epsilon)$, with n the number of blocks. Notice that generally speaking, PDP schemes do not consider the case of retrieving the file with a cheating server.

Error-correction for better guarantees An interesting idea presented in [21] is the idea to apply error-correcting code on F before running any protocol; in schemes with probabilistic VERIFY protocols, it guarantees that if the user can retrieve with high probability a portion ϵ of a file F, then all the information can be recovered, even if one bit is flipped.

2.0.1 Single replica

Let us start by introducing several options for *single-replica* provable data possession schemes. This will allow us to better understand how to build an efficient *multi-replica* scheme. This section is adapted from [9].

2.0.1.1 Trivial solution

The most trivial solution works using Message Authentication Codes. First, the user computes $\Gamma = \text{MAC}(F, K)$ with his secret key K. He sends the file, keeps Γ, and may delete the file from its local storage. To check if the server still possess the original unaltered file, the client retrieves $\widetilde{F}$ from the server, recomputes the MAC and compares it with the locally-stored Γ.

Table 2.2 Simplest form of PDP

Client		Server
STORE:		
Computes MAC of file		
$\Gamma \leftarrow \text{MAC}_K(F)$	$\xrightarrow{F}$	stores F
Stores Γ, deletes F		
VERIFY:		
Requests F	$\xrightarrow{\quad}$	
	$\xleftarrow{\widetilde{F}}$	
Recomputes MAC		
$\Gamma' \leftarrow \text{MAC}_K(\widetilde{F})$		
Accepts if $\Gamma = \Gamma'$		

The scheme presented in 2.2 is deterministic, the storage overhead for the server is null, and of one MAC signature for the user (which is one hash, 256 bits for SHA2). The VERIFY procedure is communication intensive, as it scales linearly with the file size; this alone makes the scheme completely impractical.

2.0.1.2 Zeng et al.

An easy way to reduce the communication cost of the VERIFY procedure is to divide the file F into n blocks, where the size of one block is a pre-determined constant. This was proposed in [9, 33]. Let us denote by m_i the i^{th} block of F. The client applies one MAC per block, that is $\forall i \in [1, n],\ \Gamma_i = \text{MAC}_K(m_i)$, sends the blocks, keeps all the n Γ_i's, and may delete F. The VERIFY procedure becomes probabilistic; the client randomly select l challenges in the set $[1, n]$, sends them to the server who answers with the file blocks. The client then recomputes l MAC's, compare them with the stored MAC, and is convinced that the server has the file if every new and old MAC's are equals.

The scheme presented in 2.3 has a probabilistic VERIFY procedure : the probability that even one missing block goes undetected decreases linearly with l, the number of challenged blocks. The communication cost of the VERIFY procedure is $l \cdot (32\text{bits}^1 + \text{BLOCK\_SIZE bits})$, and now scales with l rather than the file size, which is better for big files (a typical value for BLOCK_SIZE would be 8KB). The overhead for the client is n times worse than the trivial solution.

2.0.1.3 Fihlo et al.

Another possibility is to exploit the properties of the RSA scheme, as done in [20]. The client has knowledge of N, p, q, whereas the server only knows N. The client first creates an integer

[1] Bit length of a standard integer

Table 2.3 Improved version with blocks.

Client		Server
STORE:		
Divides F into n blocks		
Computes MAC for the blocks		
$\forall i \in [1,n],\ \Gamma_i \leftarrow \text{MAC}_K(m_i)$	$\xrightarrow{\quad F \quad}$	stores F
Stores the Γ_i, deletes F		
VERIFY:		
Selects l indices $i_1 \cdots i_l$ in $[1,n]$		
Requests $m_{i_1} \cdots m_{i_l}$	$\xrightarrow{\quad i_1 \cdots i_l \quad}$	
	$\xleftarrow{\quad \widetilde{m_{i_1}} \cdots \widetilde{m_{i_l}} \quad}$	Sends the blocks
Accepts if for $i_1..i_l,\ \Gamma_i = \text{MAC}_K(\widetilde{m_i})$		

m containing the whole file F (i.e. $m = F$). He picks a random a in $\mathbb{Z}_N$, and computes the tag $\Gamma = a^m \mod N$ (which is moderately easy with the knowledge of $\phi(N)$). He sends m, and keeps a, Γ, and may delete m. To run the VERIFY protocol, he picks a random r in $\mathbb{Z}_N$, and sends $A = a^r \mod N$. The server computes $B = A^m \mod N$ and sends it to the client, who accepts if $\Gamma^r = B \mod N$.

Table 2.4 RSA-based version.

Client		Server
STORE:		
Process F as an integer m		
Pick a at random in $\mathbb{Z}_N$		
Compute $\Gamma \leftarrow a^{m \bmod \phi(N)} \mod N$	$\xrightarrow{\quad F \quad}$	stores F
Stores a, Γ, deletes F		
VERIFY:		
Pick r at random in $\mathbb{Z}_N$		
Compute $A \leftarrow a^r \mod N$	$\xrightarrow{\quad A \quad}$	Compute $B \leftarrow A^m \mod N$
	$\xleftarrow{\quad B \quad}$	
Accepts if $\Gamma^r = B \mod N$		

This solution allows for a deterministic verification, and uses computation with trapdoors to make computations easy for the client. It has a storage overhead of $2 \cdot \|N\| = 2 \cdot 2048$ bits= 0.5KB on the client side, 0 on the server side; the communication cost is $2 \cdot 2048$bits= 0.5KB, which is way better than the previous protocol. However, this time the server has a significant computation cost for answering the challenge, one modular exponentiation with an exponent of $\log_2(m) = \|F\|$ bits, done in $O(\log(\|F\|))$ via fast modular exponentiation, due to the fact that the exponent is the entire file F.

2.0.1.4 Sebé et al.

The problem of the previous scheme is the exponentiation over the entire file, which is computationally intensive for the prover. As seen before, we can divide the file in blocks, and apply the scheme *per block*. It is done in [28], as follow : the file F is divided in n blocks $b_1 \cdots b_n$, and for each b_i, compute $\Gamma_i = b_i \mod \phi(N)$.

Concerning the VERIFY protocol, there is a better solution than the previous block-based scheme : Let l be the number of challenged blocks, in $[1, n]$. First, the verifier generates a random a and a random seed s, and sends them to the server. Using a PRNG with s as seed, the server computes l pseudo-random values $c_1 \cdots c_l$, then computes $r = \sum_{i=1}^{l} c_i \cdot m_i$ and sends $R = a^r \mod N$ to the client. The client verifies by first generating the same pseudo-random values with the PRNG and s, then by computing $r' = \sum_{i=1}^{l} c_i \cdot m_i \mod \phi(N)$, and finally he accepts if $R = a^{r'} \mod N$.

Table 2.5 RSA-based version with aggregated challenges

Client		Server
STORE:		
Divides F into n blocks $m_1 \cdots m_n$		
For each block a at random in $\mathbb{Z}_N$		
$\forall i \in [1, n],\ \Gamma_i \leftarrow m_i \mod \phi(N)$	$\{m_i\}$ ⟶	stores $\{m_i\}$
Stores a, $\{\Gamma_i\}$, deletes F		
VERIFY:		
Pick $a, s \in \mathbb{Z}_N \setminus \{1, N-1\}$	a, s ⟶	
$c_1 \cdots c_l \leftarrow \text{PRNG}_S()$		$c_1 \cdots c_l \leftarrow \text{PRNG}_s()$
		$r \leftarrow \sum_{i=1}^{l} c_i \cdot m_i$
$\widetilde{r} \leftarrow \sum_{i=1}^{l} c_i \cdot m_i \mod \phi(N)$	⟵ R	$R \leftarrow a^r \mod N$
Accepts if $R = a^{\widetilde{r}} \mod N$		

The novelty here is the ability to *aggregate the validation for several blocks* into one value, R, which led us to a communication cost of $3 \cdot 2048$ bits $= 0.77$KB for deterministic validation with any file size. In addition, in comparison to the previous scheme, the modular exponentiation is done with a smaller exponent, roughly of $2 \cdot$ BLOCK_SIZE. The storage overhead for the client is $n \cdot 2048$ bits, 0 for the server. The computational load on the server (and the client) can be tuned by varying l, the number of challenged blocks.

2.0.1.5 Juels-Kaliski

In the work of Juels-Kaliski [21], the file is first encrypted per blocks, then predetermined pseudo-random sequences, the *sentinels*, are inserted at specific position in the encrypted file.

The server receives the encrypted data with the sentinels, which he is unable to distinguish from the encrypted data. To VERIFY, the client asks to retrieve a specific chunk of the file stored by the server, that is, one sentinel; if the retrieved sentinel matches the inserted one, the test passes. This is repeated as many times as needed.

About encryption. This scheme is arguably computationally more intensive than the previous one due mainly to encryption; in fact, the encryption of the file is desirable with any scheme, for privacy purposes, and is in general assumed or suggested [9, 15, 21]

Table 2.6 Sentinel-based version (Juels Kaliski).

Client		Server
STORE:		
$F' = \text{ENC}_{k_{ENC}}(F)$		
$s_1 \cdots s_{\text{N\_SENTINEL}} = \text{PRF}_{k_{PRF}}()$		
Insert the s_i in F' at various positions		
Pick a at random in $\mathbb{Z}_N$		
Compute $\Gamma \leftarrow a^{m \mod \phi(N)} \mod N$	$\xrightarrow{F'}$	stores F'
Stores a, Γ, deletes F, F'		
VERIFY:		
Pick r at random in $\mathbb{Z}_N$		
Compute $A \leftarrow a^r \mod N$	$\xrightarrow{A}$	Compute $B \leftarrow A^m \mod N$
	$\xleftarrow{B}$	
Accepts if $\Gamma^r = B \mod N$		

The storage overhead for the client is only the seed of the pseudo-random function used to generate the sentinels, and the seed used to generate the sentinels positions (for instance $2 \cdot 256$ bits); on the server side however, the sentinels increase the file size; it depends on the number of inserted sentinels, in [21] the proposed concrete value for a 2GB file is 1% of the file size. The network communication is relatively efficient (a typical sentinel is 128 bits long, a challenge with q queried sentinels is therefore roughly $q \cdot (32 + 128)$ bits long, for querying the q locations and q answers).

The main problem with this scheme is that *the number of execution of the* VERIFY *protocol is bounded*, once all sentinels of a file have been queried, the subsequent call to VERIFY cannot provide proof of retrievability, as the server can trivially store the sent sentinels instead of the file and use them to answer subsequent challenges.

2.0.1.6 SW-POR

The scheme from Shacham and Waters [29] is intended to be a follow-up to the work of Juels-Kaliski [21]. It allows an unbound number of uses of the VERIFY protocol, has a stateless

verifier (which leads to a variant with public verifier), includes the notion of the Extractor as presented earlier, and proposes one of the first model (along with the Juels-Kaliski model) including a proof of security.

Two variants are presented in SW-POR, one with private verifiability (which means that only the entity who executed STORE can VERIFY), one with public verifiability. The latter is interesting, as it allows to outsource the VERIFY procedure to another service. In both variant, an error-correcting code is applied to the file F, which is then divided into n block, each block itself divided in s sectors; we denote by m_i, j the i^{th} block, j^{th} sector of F, with $i, j \in [1, n]$.

SW-POR : Private. The variant with private verifiability works as follow. With a PRF keyed with k_{PRF}, the client generates $\alpha_1 \cdots \alpha_s$ from $\mathbb{Z}_p$. Then, he sets $\tau = \tau_0 \| \text{MAC}_k(\tau_0)$, with $\tau_0 = n \| \text{ENC}_{k_{\text{enc}}}(k_{\text{PRF}} \| \alpha_1 \| \cdots \| \alpha_s)$. For each $i \in [1, n]$, the client computes $\sigma_i = \text{PRF}(i) + \sum_{j=1}^{s} \alpha_j \cdot m_{i,j}$, with the PRF keyed as before. Finally, he sends the $\{m_{i,j}\}$ as well as the $\{\sigma_i\}$ to the server, and keeps the $\{\sigma_i\}$. To VERIFY, the client sends a challenge of the form $Q = \{(i, v_i)\}$, where the first element of the tuple is a block index, and the second element is a random number in $\mathbb{Z}_p$. The server computes $\sigma = \sum_{(i,v_i) \in Q} v_i \cdot \sigma_i$, and for all $j \in [1, s]$, $\mu_j = \sum_{(i,v_i) \in Q} v_i \cdot m_{i,j}$. The client checks that $\sigma = \sum_{(i,v_i) \in Q} v_i \cdot \text{PRF}(i) + \sum_{j=1}^{s} \alpha_j \cdot \mu_{i,j}$, with the PRF keyed as before, and accepts if the equality holds.

Table 2.7 SW-POR : Private

Client		Server
STORE:		
Apply error-correcting code on F		
Divides F into n blocks and s sectors, as m_{ij}		
$\alpha_1 \cdots \alpha_s \leftarrow \text{PRF}_{k_{PRF}}()$		
$\tau_0 \leftarrow n \| \text{ENC}_{k_{\text{enc}}}(k_{\text{PRF}} \| \alpha_1 \| \cdots \| \alpha_s)$		
$\tau \leftarrow \tau_0 \| \text{MAC}_k(\tau_0)$		
$\forall i \in [1, n],\ \sigma_i \leftarrow \text{PRF}_{k_{PRF}}(i) + \sum_{j=1}^{s} \alpha_j \cdot m_{i,j}$	$\xrightarrow{\{m_{ij}\}, \{\sigma_i\}}$	stores $\{m_{ij}\}, \{\sigma_i\}$
Stores τ, deletes $F, \{\sigma_i\}$		
VERIFY:		
Choose l indices $i_1 \cdots i_l$ in $[1, n]$		
Pick $v_{i_1} \cdots v_{i_l}$ at random in $\mathbb{Z}_p$		
$Q \leftarrow \{(i_k, v_{i_k})\}\ \forall\ k \in [1, l]$	$\xrightarrow{Q}$	$\sigma \leftarrow \sum_{(i,v_i) \in Q} v_i \cdot \sigma_i$
		$\forall j \in [1, s],$
		$\mu_j \leftarrow \sum_{(i,v_i) \in Q} v_i \cdot m_{i,j}$
	$\xleftarrow{\sigma, \{\mu_j\}}$	
$\sigma' \leftarrow \sum_{(i,v_i) \in Q} v_i \cdot \text{PRF}_{k_{PRF}}(i) + \sum_{j=1}^{s} \alpha_j \cdot \mu_{i,j}$		
Accepts if $\sigma = \sigma'$		

SW-POR : Public. The variant with public verifiability is rather similar, but involves BLS signatures [11] to remove the need for k_{PRF} for the verifier. As usually with BLS, let $e : G \times G \mapsto G$ be a bi-linear map, g a generator of G, and $H : \{0,1\}^* \mapsto G$ be a BLS hash function. The client generates a random signing key pair (spk, ssk), and a random $\alpha \in \mathbb{Z}_p$, and computes $v = g^\alpha$. The secret key is (α, ssk), the public key is (v, spk).

First, the client selects a random file name *name* from a sufficiently large domain, picks s random elements $u_1 \cdots u_s$ from G, computes $\tau_0 = \text{name}\|n\|u_0\| \cdots \|u_{s-1}$, and finally $\tau = \tau_0\|\text{SIGN}(\tau_0)$. For each $i \in [1,n]$, he computes $\sigma_i = \left(\text{H}(\text{name}\|i) \cdot \prod_{j=0}^{s-1} u_j^{m_{i,j}}\right)^\alpha$. Finally, he sends the $\{m_{i,j}\}$ as well as the $\{\sigma_i\}$ to the server, and keeps the $\{\sigma_i\}$. To VERIFY, the verifier sends $Q = \{(i, v_i)\}$ as before. The server computes $\sigma = \sum_{(i,v_i)\in Q} \sigma_i^{v_i}$, and for all $j \in [1,s]$, $\mu_j = \sum_{(i,v_i)\in Q} v_i m_{i,j}$. The verifier checks that $e(\sigma, g) = e\left(\prod_{(i,v_i)\in Q} \text{H}(\text{name}\|i)^{v_i} \cdot \prod_{j=0}^{s-1} u_j^{\mu_j}, v\right)$ and accepts if the equality holds.

Table 2.8 SW-POR : Public

Client		Server
STORE:		
Apply error-correcting code on F		
Divides F into n blocks and s sectors, as m_{ij}		
Pick $u_1 \cdots u_s$ at random from G		
$\tau_0 \leftarrow \text{name}\|n\|u_1\| \cdots \|u_s$		
$\tau \leftarrow \tau_0\|\text{SIGN}(\tau_0)$		
$\forall i \in [1,n], \sigma_i \leftarrow \left(\text{H}(\text{name}\|i) \cdot \prod_{j=1}^{s} u_j^{m_{i,j}}\right)^\alpha$	$\xrightarrow{\{m_{ij}\}, \{\sigma_i\}}$	stores $\{m_{ij}\}, \{\sigma_i\}$
Stores $\tau, \{\sigma_i\}$, deletes F		
VERIFY:		
Choose l indices $i_1 \cdots i_l$ in $[1,n]$		
Pick $v_{i_1} \cdots v_{i_l}$ at random in $\mathbb{Z}_p$		
$Q \leftarrow \{(i_k, v_{i_k})\}\ \forall\ k \in [1,l]$	$\xrightarrow{Q}$	$\sigma \leftarrow \sum_{(i,v_i)\in Q} \sigma_i^{v_i}$
		$\forall j \in [1,s]$,
		$\mu_j \leftarrow \sum_{(i,v_i)\in Q} v_i m_{i,j}$
	$\xleftarrow{\sigma, \{\mu_j\}}$	
$\sigma' \leftarrow e\left(\prod_{(i,v_i)\in Q} \text{H}(\text{name}\|i)^{v_i} \cdot \prod_{j=1}^{s} u_j^{\mu_j},\ v\right)$		
Accepts if $e(\sigma, g) = \sigma'$		

This scheme combines several desirable properties : the communication cost scales with l and s, and is of roughly $l \cdot (32 + 2048) + 2048 + s \cdot 2048 = (l + s) \cdot 2048 + l \cdot 32 + 2048$ bits, and it can simply be made constant by sending two seeds and a number and having an expansion function instead of Q, which would give a communication cost of $2 \cdot 2048 + 32 + 2048 + s \cdot 2048 = (s + 3) \cdot 2048 + 32$ bits. The storage overhead for the client depends only on the number of sectors s, and is roughly $(s + 2) \cdot 2048$ bits (not including the file name); the storage overhead for the server is of $n \cdot 2048$ bits. The computational complexity is moderate (lots of operations

modulo n, but nothing as intensive as exponentiating over the whole file, as done in 2.4). In addition, it has *extractability*, which guarantees the retrievability of the file if a non-negligible percentage of runs of the VERIFY protocol succeed.

2.0.1.7 OPOR

The publicly verifiable version of SW-POR [29] allows for anyone to check if the cloud server still possess a given file; it allows the client, often computationally bounded, to delegate the VERIFY protocol to another party. The problem is that this party, the Auditor, needs to be trusted by the client; The Service Provider and the Auditor can easily collude to produce valid challenge/response pairs in advance. This is the problem addressed with OPOR [6], which stands for Outsourced Proofs of Retrievability. In this system, composed of three entities (User, Auditor, and Service Provider), the Auditor and the User agrees on a SLA, mainly on how many verification will be done. The auditor then periodically challenge the Service Provider, and stores the answer. By a clever cryptographic property of the scheme, should the User want to verify that the Auditor did his task, the stored answers from the server can be aggregated into one short answer, proving to the user that all those challenge/response protocol did succeed. In addition, to prevent the Auditor and the Service Provider on colluding and pre-agreeing on some challenges, the randomness used to consistute the challenges comes from the Bitcoin block chain, believed to be unpredictable in advance, and which "randomness history" is public for any point in time. Therefore, the User can further audit the Auditor, by checking that he used the correct randomness.

The scheme shows a clever way of aggregating several responses into one, easy-to-compute final proof, which is here used for several challenges on the same file, but could be used for different replicas of a same file, for instance. Unfortunately, in itself the scheme does not account at all for multiple replicas.

Important lessons from single-replica schemes. First, efficiency is the main goal to achieve, as there exist trivial solution from cryptographic primitives. The efficiency is measured via three factors:

1. the bandwidth usage
2. the storage overhead
3. computational complexity

With regards to the primitives, we especially want the protocols CHALLENGE, ANSWERCHALLENGE, and USERVERIFY to be as efficient as possible, and if possible unbounded in the number of use. Most scheme presented have *private* verifiability : only the entity running STORE is able to VERIFY. *Public* verifiability means that anyone, given public knowledge only, can VERIFY on behalf of a client, which is an advantage.

Dividing the file in blocks (which allows for a probabilistic VERIFY procedure and better computational efficiency) allows some schemes to challenge multiple blocks while having a

constant bandwidth usage for the challenge (by aggregating them), which is a very desirable property.

Extractability is definitely an interesting capacity of some schemes, and is desirable if does not hinders too much performance. One requirement for extractability when challenging replicas is that the original file F must be recoverable from any replica, which is not always the case [15]. In particular, [29] uses blocks and *sectors* smaller than the modulus to process the file, where [15] process the file based only on blocks much bigger than the modulus used, which destroys information and prevent to reconstruct F from a replica.

2.0.2 Multi replica

Introduction. The problem of having a *multiple copies of one file* is very relevant in today's cloud architecture; users want the insurance that their file are not too vulnerable to hardware crashes, and guarantees against data loss is a desired thing. This problem can be solved by having several replicas of one file, preferably stored on different devices owned by the cloud provider. Different level of guarantees could be given, for instance :

1. Each replica can be stored anywhere, but need to be distinct from each other.
2. Each replica is on a distinct storage device.
3. Each replica is in a distinct geographic region.

Conventions. In what follow, (single-replica) PDP / POR protocols are often used as a building block for more complex constructions. In the absence of further precision, the reader may consider private-verifiability SW-POR [29], as it is a efficient and well-known scheme.

2.0.2.1 Trivial solution

The easiest solution uses symmetric encryption on the client side to create R different replicas. First, the user generates $k_1 \cdots k_R$ keys for a symmetric encryption scheme (for instance AES-CBC). From the file F, he creates $\widetilde{F}_i = \text{ENC}_{k_i}(F)$ for every $i \in [1, R]$. Then, he trivially uses R times a PDP/POR protocol, one for each $\widetilde{F}_i$. To Verify the R replicas, again, the client runs R times the PDP Verify protocol.

Table 2.9 Simplest form of multi-replica PDP/POR

Client		Server
Store:		
Pick $k_1 \cdots k_R$ at random		
Compute $\widetilde{F}^{(i)} = \text{ENC}_{k_i}(F)\ \forall i \in [1, R]$		
Runs Store of the PDP on $\widetilde{F}^{(i)}\ \forall i \in [1, R]$	$\longrightarrow$	
Stores the tags Γ_i, deletes the $\widetilde{F}^{(i)}$ and F		
Verify:		
Runs Verify of the PDP on $\widetilde{F}^{(i)}\ \forall i \in [1, R]$	$\longrightarrow$	
	$\longleftarrow$	
Accepts if all instances of Verify succeeded		

The scheme presented in 2.9 involves trivially R times the resources needed for the underlining POR : the communication cost, the storage overhead for both the client and the server, as well as the computation time, scales linearly with R. The scheme makes no assumption on the replica location, and the burden of creating the replica is fully taken by the user. The server is

forced to store the different replicas (instead of storing $\widetilde{R} < R$) trivially because the $\widetilde{F}^{(i)}$ are different by property of the encryption scheme used; he cannot use another replica to answer a challenge issued to a replica, and is immediately detected if one replica is missing.

2.0.2.2 MR-PDP

The Multi-Replica Provable Data Possession scheme from Curtmola et al. [15] is one of the first to take into account multiple file copies. The main improvement is that challenging multiple replica is done concurrently, hence making it more efficient than R times the cost of a single check. In what follows, we briefly describe the scheme; we refer the interested reader to the full paper for details.

The client has knowledge of p, q two strong primes, of a pair e, d such that $e \cdot d = 1 \mod N$ where $N = p \cdot q$, of g a generator of QR_N, has access to a hash function $h : \{0,1\}^* \to \mathrm{QR}_N$ and to a PRF ψ. First, an error-correcting code is applied to F, where F is preferably encrypted. The file F is split in blocks $m_1 \cdots m_n$. The user picks a random $v \in \mathbb{Z}_N$, and computes the authentication tags $\Gamma_i = (h(v\|i) \cdot g^{m_i})^d$ for $i \in [1, n]$. He creates the replica u by computing $m_{u,i} = b_i \oplus \psi(u\|i)$, for $i \in [1, n]$. After creating R replicas, the client sends to the server the file $\{m_i\}$, the R replicas $\{m_{u,i}\}$ for $u \in [1, R]$, and the tags $\{\Gamma_i\}$.

To Verify, the client picks $s \in \mathbb{Z}_N$, computes $g_s = g^s \mod N$, and sends a challenge of the form $Q = \{(i, v_i)\}$, where the first element of the tuple is a block index, and the second element is a random number in $\mathbb{Z}_N$, along with g_s. The server computes $\sigma = {g_s}^{\sum_{(i,u)\in Q} m_{u,i}}$ and for all $\Gamma = \prod_{(i,u)\in Q} \Gamma_i$. The client checks that σ is equal to $\left(\frac{T^e}{\prod h(v\|i)} g^{\sum_{(i,u)\in Q} \psi(u\|i)}\right)^s$. If all computations are done correctly, $\left(\frac{T^e}{\prod h(v\|i)}\right)^s$ is equal to ${g_s}^{\sum_{(i,u)\in Q} m_i}$, which leads to $\sigma \stackrel{?}{=} {g_s}^{\sum_{(i,u)\in Q} m_i} \cdot {g_s}^{\sum_{(i,u)\in Q} \psi(u\|i)} = {g_s}^{\sum_{(i,u)\in Q} m_{u,i}}$. The client accepts if the equality holds.

2.0.2.3 EMC-PDP

The scheme from Barsoum et al. [9] relies on BLS signatures [11] and on ciphers to generate the replicas. It is inspired from, and fixes a problem in MR-PDP [15] (where the service provider can use blocks from other files to answer a challenge) by changing the tags creation; In addition, it proposes an efficient way of checking *all* the outsourced data in one query by aggregating not only the blocks in one file, but over all the replicas.

Two version of the scheme are presented in [9], we described here the deterministic version. The storage overhead for the client comprises the BLS parameters, the symmetric key K used for the blocks creation, and the pair of key x and y; in total, depending on the group size for BLS, we can upper bound by $5 \cdot 256$KB. In particular, notice that the client does not need to keep tags. On the server side, the overhead is also upper bounded by $n \cdot 256$ bits for standard groups size of BLS. The network communication is also highly efficient (only the replicas and the tags are sent), and the communication for the Verify protocol is constant, of $3 \cdot 256$KB.

Table 2.10 MR-PDP

Client		Server
Set $p = 2p' + 1$, p' prime $q = 2q' + 1$, q' prime $N = pq$ Pick g a generator of QR_N Let $h : \{0,1\}^* \to \mathrm{QR}_N$ be a hash function Let ψ be a pseudo random function		
STORE: Divides F into n blocks $m_1 \cdots m_n$ Computes the tags : Pick $v \in \mathbb{Z}_N$: $\Gamma_i \leftarrow (h(v\|i) \cdot g^{m_i})^d$ for $i \in [1,n]$	$\xrightarrow{\{m_{u,i}\}, \{\Gamma_i\}}$	stores the $\{m_i\}, \{\Gamma_i\}$
REPLICATE (replica u): $m_{u,i} \leftarrow b_i \oplus \psi(u\|i)$, for $i \in [1,n]$	$\xrightarrow{\{m_{u,i}\}}$	stores the $\{m_{u,i}\}$
VERIFY: Choose l indices $i_1 \cdots i_l$ in $[1,n]$ Choose l indices $u_1 \cdots u_l$ in $[1,R]$ Pick $s \in \mathbb{Z}_N$, set $g_s = g^s \mod N$ $Q \leftarrow \{(i_k, v_{i_k}), g_s\} \; \forall \; k \in [1,l]$	$\xrightarrow{Q}$	$\sigma \leftarrow {g_s}^{\sum_{(i,u)\in Q} m_{u,i}}$ $\Gamma \leftarrow \prod_{(i,u)\in Q} \Gamma_i$
	$\xleftarrow{\sigma, \Gamma}$	
$\sigma' \leftarrow \left(\frac{T^e}{\prod h(v\|i)} g^{\sum_{(i,u)\in Q} \psi(u\|i)}\right)^s$ Accepts if $\sigma = \sigma'$		

2.0.2.4 LoST

The authors of [32] propose a way to prove the location of stored data in addition to the classical provable data possession. The main goal is to prove that a given data is stored within a geographic region, which might be requested by law for medical data for instance. When it is not the case, the user is able to detect it, and then take legal measures against the cloud provider. The scheme is intended to protect against an rational service provider who is willing to reduce his cost (it is also briefly demontrasted that when the service provider is fully malicious, no guarantees can be given about the location of the data or copies of it). First, the data is located by recording the response time of the servers and using a set of trusted servers whose location is known. In addition, in the case where the file must be stored several times among multiple server, the scheme propose a version of the SW-POR scheme with recoding, the ability to produce new, distinct replicas from the source file or another replica; each server

Table 2.11 EMC-PDP

Client		Server
Let K be key for a symmetric cipher Enc		
Let $e : G_1 \times G_2 \mapsto G_T$ be a bi-linear map		
Let u a generator of G_1, g of G_2		
Let $x \in \mathbb{Z}_p$ be a secret key, and		
$y = g^x \in G_2$ a public key.		
$q = 2q' + 1$, q' prime		
Let $H : \{0,1\}^* \to G_1$ be a hash function		
Let ψ be a pseudo random function		
REPLICATE ():		
$F^{(i)} \leftarrow \text{Enc}_K(i \Vert F)$ for each replica $i \in [1, R]$	$\xrightarrow{\{F^{(i)}\}}$	stores the $\{F^{(i)}\}$
CREATETAGS :		
$F_{\text{id}} \leftarrow \text{Filename} \Vert R \Vert n \Vert u$		
$\sigma_{i,j} \leftarrow (H(F_{\text{id}}) \cdot u^{m_{i,j}})^x$ for $i \in [1, R], j \in [1, n]$	$\xrightarrow{\{\sigma_{i,j}\}}$	stores the $\{\sigma_{i,j}\}$
VERIFY:		
Pick $k \in \mathbb{Z}_N$		
Generate $R \times n$ values $r_{i,j}$ from ψ_k	$\xrightarrow{k}$	Also computes $r_{i,j}$
		$\sigma \leftarrow \prod_{i=1}^{R} \prod_{j=1}^{n} \sigma_{i,j}^{r_{i,j}}$
		$\mu \leftarrow \sum_{i=1}^{R} \sum_{j=1}^{n} r_{i,j} \cdot b_{i,j}$
	$\xleftarrow{\sigma, \mu}$	
$Q \leftarrow \sum_{i=1}^{R} \sum_{j=1}^{n} r_{i,j}$		
Accepts if $e(\sigma, g) = e(H(F_{\text{id}})^Q \cdot u^\mu, y)$		

holds on replica, which is different every other replica; this prevents one server to lie about its location by having another server answering instead.

The scheme unfortunately requires that no "extra copies" of the data is made, that is, copies the user is unaware of (which would make the localisation of the data impossible); this requirement is extremely strong. In addition, the recoding, while not being particularly a light operation, is not conceived as a time-taking problem, and no analysis is done about the ability to recode "on the fly" a replica. This would allow a server to answer when another server is queried, allowing to cheat about the data location.

2.0.2.5 RAFT

The idea behind RAFT [13] is to provide guarantees about the fault-tolerence of a file stored in the cloud. Where other multi-replica PDP/POR schemes focused on verifying the replication

on a logical level, RAFT checks that the file is replicated on different hard drives. By careful parameter tuning, they achieve to distinguish between a honest cloud provider with c drives, or a dishonest "cheap and lazy" non-malicious adversary that uses less than c drives. The main technique is to request several blocks of a file (or its replicas) that should be on different drives ; by timing the response, and knowing the pre-agreed data mapping, the client can compute the expected response time; a dishonest will be slower than expected, since at least two IO reads will be sequential on one of the hard drives. By a clever lock-step mechanism, the query is iterated on the server, reducing the variance of the drive access times.

The scheme is dependant on the network traffic in the timing measurement, and require a good approximation of the true transmission time to correctly interpret results; this transmission times varies by hours and days; It can be smoothed out through increasing the number of runs of the VERIFY protocol. In addition, it requires the cloud provider to respect a pre-agreed mapping, which might hinder the practicability of the cloud server operations, such as hardward upgrades and maintenance.

Chapter 3

Model and Building blocks

This chapter formally defines the model in which we build Mirror. The model is adapted from [5].

3.1 Model

In this section, we introduce a formal model for Proofs of Retrievability and Replication, PoR^2.

We consider a user $\mathcal{U}$ that want to outsource the storage of a file F, and cloud storage provider $\mathcal{SP}$ that owns a collection of servers $\mathcal{S}_i$. The user agrees on a storage policy with $\mathcal{SP}$, in which the number of replicas R the cloud provider should keep is specified. In particular, they agree on :

Condition 1: $\mathcal{SP}$ will store the file F in its entirety.

Condition 2: $\mathcal{SP}$ will store R additional replicas of F in their entirety.

A PoR^2 protocol aims to ensure both conditions are respected. Condition 1 directly hints that we will need a (single-replica) PDP or a POR scheme, with the usual Store, Challenge, AnswerChallenge, and UserVerify procedure. For Condition 2, we add to PoR^2 a procedure called Replicate for generating the replicas, and a procedure CheckReplica for checking the correctness of the replicas (in the case where the user runs Replicate and the server wants to check if the replicas are correct).

In what follows, we give a formal description of the procedures involved in a PoR^2 scheme. This model adapts and extends the original POR model [29].

3.1.1 The Store procedure

The Store procedure is executed by the user. It takes as input a file $\widetilde{F}$ to be outsourced, and some security parameter κ. It produces a file F to be stored by the server, one or several authentication tags Γ, and some public parameters Π used to generate replicas. Formally, we have :

$$(F, \Gamma, \Pi) \leftarrow \textsc{Store}(\widetilde{F}, \kappa)$$

Depending of whether the scheme is privatly or publicly verifiable, the file tag Γ are kept by the user, or made public.

3.1.2 The Replicate procedure

The Replicate procedure is executed either by the user $\mathcal{U}$ or the server $\mathcal{S}$. It takes as input the file F, the file tags Γ, and the replications parameters Π outputted by Store, and produces a collection of replicas $F^{(1)}, \cdots, F^{(R)}$, where R was agreed on beforehand (or is part of Π). In addition, the procedure may create some *replica tag* Γ^* which allows to validate the correctness of the copies. Formally, we have :

$$(F^{(1)}, \cdots, F^{(R)}, \Gamma^*) \leftarrow \text{Replicate}(F, \Gamma, \Pi)$$

Notice that it captures the cases when the user generates the replicas and send all of them to the server, as done in [15], and when the server is doing the replication itself.

3.1.3 The CheckReplica procedure

This procedure is used by the server $\mathcal{S}$ when the user $\mathcal{U}$ runs Replicate. As mentionned before, the user might try to use the "replicas" to store other data (*instead* of redundantly storing F), which is not desirable for the server when the user is charged less for storing replicas. This is the aim of the CheckReplica procedure, which gives a way for the server to validate that the so-claimed replicas $F^{(1)}, \cdots, F^{(R)}$ are indeed only replicas of F, and does not contain additional information. It takes as input the original file F and one replica $F^{(r)}$ along with its index r, the copy parameters Π, the file tag Γ and the replica tag Γ^*, and outputs a binary decision whether $F^{(r)}$ is believed to be a correct replica of F according to the Replicate procedure and the parameters Π.

$$\text{dec} \in \{\text{accept}, \text{reject}\} \leftarrow \text{CheckReplica}(F, F^{(r)}, r, \Pi, \Gamma, \Gamma^*)$$

It holds that for every replica $F^{(r)}$ that is the correct output of Replicate, along with the correct parameters, CheckReplica accepts.

When the server itself runs Replicate, this procedure is not needed.

3.1.4 The Verify protocol

This protocol regroups the following procedures : Challenge, AnswerChallenge and UserVerify. A verifier $\mathcal{V}$, that is, the user if the scheme is privately verifiable, and the server $\mathcal{S}$ execute an interactive protocol to convince the verifier that both the outsourced file F and the R replicas $F^{(1)}, \cdots, F^{(R)}$ are correctly stored. We define the Verify protocol as such :

$$\text{dec} \in \{\text{accept}, \text{reject}\} \leftarrow \text{Verify}(\mathcal{U} : \Gamma, \mathcal{S} : F, F^{(1)}, \cdots, F^{(R)}, \Gamma^*)$$

No requirement on location. We do not put a constraint on *where* each replica needs to be stored; The cloud provider is free to decide this parameters, and the replicas do not need to be on a individual hard drives, or on a different geographic area, as was proposed in [10, 13, 32].

3.2 Adversary model

Similar to the existing work in the area of PDP and POR [32], [31], the adversary we consider is *rational*, i.e., we consider the *rational attacker model*. Such attacker only deviates from the protocol if doing so grants him a concrete advantage, in opposition to an attacker who would deviate arbitrarily from the protocol for the sake of it.

To be more precise, a rational service provider aims to provide its service using less storage than it would require if done honestly, which would save him storage cost; in addition, he wants to do so without being detected by the users, which would results in litiges that could negate the aforementionned gain. It also implies that if the service provider cannot save any ressources by misbehaving, then he will simply behave according to the protocol.

In a similar manner, the goal of the rational user is to reduce his overall cost. For instance, in schemes where the users create and upload the replicas, if the storage provider charges less for replicas than for storing the original file[1], a rational user might try to encode additional information in the replicas (or put other files instead of the replicas).

3.3 Security goals

We define three notions that a PoR[2] scheme must guarantee :

- **Extractability** : Given a number of successful runs of VERIFY between the user and the service provider, the user can recover the uploaded file F. This may include the recovery of a replica $F^{(r)}$ as an intermediate step.
- **Storage allocation** : The service provider allocates and uses at least as much storage as it is necessary to store F and the replicas $F^{(1)}, \cdots, F^{(r)}$, and is able to efficiently serve this data.
- **Correct replication** : $F^{(1)}, \cdots, F^{(r)}$ are correct replicas (that is, outputted by REPLICATE given Π) of F.

We further specify the presented notions.

Extractability. This is used in the scenario of a malicious cloud provider. Extractability guarantees that if the VERIFY protocol (on the uploaded file F) succeed enough time, the user is able to recover F. Intuitively, it implies that when running VERIFY, the server gives the user some information about F, and that we can recover the whole file F by running enough time the protocol. This is was first defined in [21], followed by [29]. Formally, the requirement is that there must exist an algorithm called EXTRACTOR which is able to extract F in interaction with the adversary (given a number of successful runs of VERIFY). This algorithm has access to the secret keys of the honest parties, here the user, and has non-black-box access to the machine implementing the corrupted parties.

[1] as it is the case with Amazon S3 [2, 4]

Storage allocation. This is used in the scenario of a malicious cloud provider. Let us denote by SA the storage allocated by the service provider for the file F and its replicas $F^{(1)}, \cdots, F^{(R)}$. Clearly, to fulfill the SLA agreed with the user (which specifies that the service provider should keep R redundant copies of F), we want this value to be as close as possible to $\|F\| \cup \|F^{(1)}\| \cup \cdots \cup \|F^{(r)}\|$. Let us define :

$$\rho = \frac{\text{SA}}{\|F\| \cup \|F^{(1)}\| \cup \cdots \cup \|F^{(r)}\|}$$

We want ρ as close to 1 as possible — notice that a value greater than 1 does not make sense for the rational cloud provider. We can now define the storage allocation requirement : the protocol Verify accepts only if ρ is close to 1.

Correct replication. This is used in the scenario of a malicious user. When the user creates the replicas, the cloud provider want the insurance that he does not encode additional data. Since Replicate might be a probabilistic process, and so could CheckReplica, our requirement is also probabilistic. Let F, Π, Γ be the output of a run of Store, and $F^{(r)}$ and Γ^* be the output of a run of Replicate. We want that CheckReplica (F, $F^{(r)}$, Π, Γ, Γ^*) always accept (i.e., with probability 1). Similarly, when $F^{(r)}$ and Γ^* are *not* the output of a run of Replicate, we want that CheckReplica (F, $F^{(r)}$, Π, Γ, Γ^*) rejects with a probability close to 1.

3.4 Building blocks

Before we discuss our schemes, we start by overviewing the main building blocks which we will rely on throughout this thesis.

3.4.1 RSA Time-Lock puzzles

This puzzle was proposed by Rivest *et al.* in [26].

Setup. Let p, q be two safe primes, ie. $p = 2p' + 1$ and $q = 2q' + 1$, with p', q' primes, and let $N = pq$.

Creation of the cipher. The cipher C is created from the message M such as $C = M + X^{a^t} \mod N$, with X in $\mathbb{Z}_N$, and a and t some large integers.

Decryption. C, X, a, t are sent to the recipient, who can decrypt after computing the puzzle $X^{a^t} \mod N$ which takes $\approx t \cdot \log a$ multiplications in $\mathbb{Z}_N$.

The puzzle can also be used as a challenge between a prover and a verifier, as proposed in [14]. The setup is the same as above; the verifier picks a composite X in $\mathbb{Z}_N$ as well as a very large integer t, and asks the prover to compute $R = X^t \mod N$, which takes $\approx \log(t)$

multiplications. The verifier, knowing the factorization of n, can easily verify using the trapdoor given by Euler's formula. He computes $R' = X^{x \mod \phi(n)} \mod N$ and accepts if $R = R'$.

3.4.2 Feedback Shift Register

A Feedback Shift Register is a well-understood mathematical structure that produces pseudo-random numbers, and is often used as a building block for stream ciphers as it produces long sequences based on a short initial state.

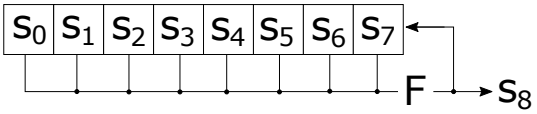


Figure 3.1: A FSR of length $\lambda = 8$

More formally, a Feedback Shift Register of length λ over a field $\mathbb{F}$ consists of an internal state $S = (s_1 \cdots s_\lambda)$, and a update function $F : \mathbb{F}^\lambda \to \mathbb{F}$. The register is initialized with a state S_0. At each clock tick, the state is updated in the following manner :

$$S_t = F(S_{t-1})$$

At each update, the oldest value in the state will be discarded, and the output of the feedback function will be appended to the state, in the following fashion :

$$S_0 = (s_1, s_2, s_3, \cdots s_\lambda)$$

$$S_1 = (s_2, s_3, \cdots s_\lambda, F(S_0))$$

$$S_2 = (s_3, \cdots x_\lambda, F(S_0), F(S_1))$$

$$\cdots$$

We refer the readers to [23] for additional information on FSRs.

Linear FSR. When the function $F(s_1, \cdots, s_\lambda)$ has the form $\sum_{i=1}^{\lambda} c_i s_i$, the feedback shift register is *linear*, and this subclass of FSR is called LFSR.

Output sequence. Let us denote by $(F(S_0), F(S_1), \cdots)$ the output sequence of a FSR. For convenience, we use the notation $(s_{\lambda+1}, s_{\lambda+2}, \cdots)$ as a strict equivalent, with $s_t = F(S_{t-\lambda-1})$ for $t > \lambda$. This output sequence is finite (and will repeat at some point), and its period is bounded. As long as the initial state is not all zeros, obtaining a FSR with maximal period is easily done by choosing a fitting feedback polynomial.

Feedback polynomials. An important notion about LFSR is that they can be represented with a *feedback polynomial*; for a linear feedback function $F(s_i, \cdots, s_{i+\lambda}) = \sum_{j=1}^{\lambda} c_j \cdot s_{i+j}$, the feedback polynomial $f(x) \in \mathbb{F}(x)$ is defined as :

$$f(x) = x^{\lambda} - \sum_{i=1}^{\lambda} c_i \cdot x^{i-1}$$

That is, the feedback polynomial represents the coefficients of the feedback function. Furthermore, any polynomial $f^*(x)$ multiple of $f(x)$ is also a feedback polynomial for the same sequence.

Puzzle from two LFSR. The idea is to construct a small LFSR of length λ from a random feedback polynomial $f(x)$ of degree $\lambda + 1$. Then, a random polynomial $q(x)$ of degree $\lambda^* - \lambda$ is selected and multiplied with $f(x)$ to obtain $f^*(x)$ of degree $\lambda^* + 1$. This second polynomial, of arbitrary length, allows to construct a LFSR of length λ^* which will output the same sequence than the first LFSR. The computations in the second case are arbitrarily more costly, as they include $\lambda^* - \lambda$ additional terms in the computation of the next element.

Chapter 4

Overview and Possible solutions

In this chapter, we explore several novel schemes. Each of them was a candidate for Mirror, and is an alternative to the final Mirror, with different trade-offs.

4.1 Overview

The objective of Mirror is to provide a verifiable replication mechanism for the cloud providers, without requiring that the users construct the replicas. In Mirror, the user sends the original file F and its authentication tags (corresponding to the original file only, as done in existing POR/PDP schemes), along with compact puzzles. The solution of these puzzles will then be combined with the original files in order to create the replicas.

The requirements for the puzzles are as follow :

1. They need to require noticeable time to compute for the cloud provider
2. They need to require significantly less time for the user, i.e. by using a trapdoor.
3. They need to have solutions that are at least as large as the required storage for replicas.
4. They can be efficiently combined with the sectors of the original file in order to create the replicas while respecting the homomorphic properties needed for compact proofs. [1]

With those requirements, Mirror creates a considerable incentive for a rational provider for correctly creating and storing the replicas; otherwise, the cheating provider will be detected with high probability.

Mirror is inspired by the private version of SW-POR [29], as it is an efficient and widely-known single-replica scheme.

4.1.1 Chapter Organization

This chapter is structured as follow. Below are presented three different schemes; each corresponding section is divided as follow :

[1] This restricts the possibilities in terms of type of puzzle; for instance, hash-based puzzles cannot be efficiently combined with the authentication tag of each data sector.

1. First, for all schemes but the first, a comparison to the previous scheme is done in preamble. We explain the idea that is behind this new scheme, and put it in contrast with the weaknesses of the previous scheme.
2. Then, the protocol is described.
3. Finally, we analyze the performances of the scheme; we show plots representing the latency of certain operations given the size of the input file. In addition, we present the distributions of the most costly mathematical operations.

4.1.2 Methodology

To perform our analysis, we relied on implementations made in Scala [27]; we relied on SHA256 for the default hash function, and on the JVM built-in random number generator.

We deployed our implementation on a private network consisting of two machines directly connected to each other; their description is given in table 4.1. The speed of the link between the machines was set to 100Mbps, and to emulate a realistic Wide Area Network simulation, we used the tool Netem [25] to shape the traffic following a Pareto distribution with a mean of 20ms, and a variance of 4ms as done in [16].

Table 4.1 Implementation Settings

Parameter	**Value**
Number of Machines	2
Machine Type	Intel Xeon E5-2640
Number of cores per machine	24 cores
RAM per machine	32 GB
Network speed	100 Mbps
Mean packet latency	10 ms
Variance in packet latency	4 ms

Each data point in the plots is averaged over 10 independent measurements; where appropriate, we include the corresponding 95% confidence intervals.

Default parameters. The defaults parameters that will be used for all those schemes are defined Table 4.2.

Table 4.2 Default values

Parameter	**(also called)**	**Value**
BlockSize		8192 KB
SectorSize		256 KB
RSA Modulus Length		2048 bits
PRF KeyLength		256 bits
NumberOfQueriedBlocks	l	100

4.2 Additive scheme

This scheme is called additive due to the operation done during the replication; the puzzle is *added* to the sector $m_{i,j}$ to obtain the replica $m_{i,j}^{(r)}$.

4.2.1 Protocol

Store. The user first divides the file into blocks and sectors of fixed size, and picks $s + n$ secrets, $\alpha_1 \cdots \alpha_s$ and $\text{secret}_1 \cdots \text{secret}_n$ using a keyed PRF. He computes one tag per block :

$$\sigma_i \leftarrow \text{secret}_i + \sum_{i=1}^{n} \alpha_i m_i \mod N$$

He then sends the file F along with the tags σ_i to the service provider, and may delete F. He keeps the tags σ_i along with the key for the PRF.

The user also picks a prime p_r and a big[2] exponent E_r for each desired replica r. He sends the p_r and the E_r to the server.

Replicate. Upon reception, the service provider will start computing a number of R of replicas. For each replica r, block i and sector j, the server will compute and store

$$m_{i,j}^{(r)} \leftarrow m_{i,j} + p_r^{E_r^{(j-1)\cdot s+i}} \mod N$$

where s is the number of sector. This constitutes a puzzle for the server, which does not know the factorization of N, and will have to compute a the moderately intensive operation $p^{Ex} \mod N$. This will prevent the server from cheating and replicating "on the fly" when challenged later on.

Challenge. The challenge protocol is very close to what is presented in the related work; we want to communicate a collection of indices $i_1 \cdots i_l$ representing some blocks of the file, and a collection of random numbers $v_1 \cdots v_l$, one per block. To make the communication cost of the Challenge protocol constant, we assume the client and the server both have access to a keyed pseudo-random function, and we transmit only two seeds $\text{seed}_i, \text{seed}_v$, that will generate numbers in the range $[1, \text{NumberOfBlocks}]$ and $[1, \text{RSAModulus}]$, respectively. The size of the collection $i_1 \cdots i_l$, that is l, is fixed and known to both parties.

AnswerChallenge. First, the server computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. Then, he computes the following values σ and μ_j for each sector $j \in [1, s]$.

$$\sigma \leftarrow \sum_{(i,v_i)\in Q} \sigma_i \cdot v_i \qquad \text{and} \qquad \mu_j \leftarrow \sum_{(i,v_i)\in Q} v_i \cdot m_{i,j}^{(r)}$$

He transmits the s values of μ_j and the value of σ to the client.

[2]This is a puzzle for the server. The size of E can be tuned to achieve a certain expected time to solve the puzzle.

UserVerify. The client also computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. He computes the Left Hand Side as follow :

$$\text{LHS} \leftarrow \sum_{j=1}^{s} \alpha_j \cdot \mu_j^* + \sum_{(i,v_i) \in Q} v_j \cdot \text{secret}_i$$

For the Right Hand Side, he first computes P_j, $j \in [1, s]$:

$$P_j \leftarrow \sum_{(i,v_i) \in Q} v_i \cdot p_r^{E_r{}^{(i-1)\cdot s+j} \mod \phi(N)} \mod N$$

And finally computes :

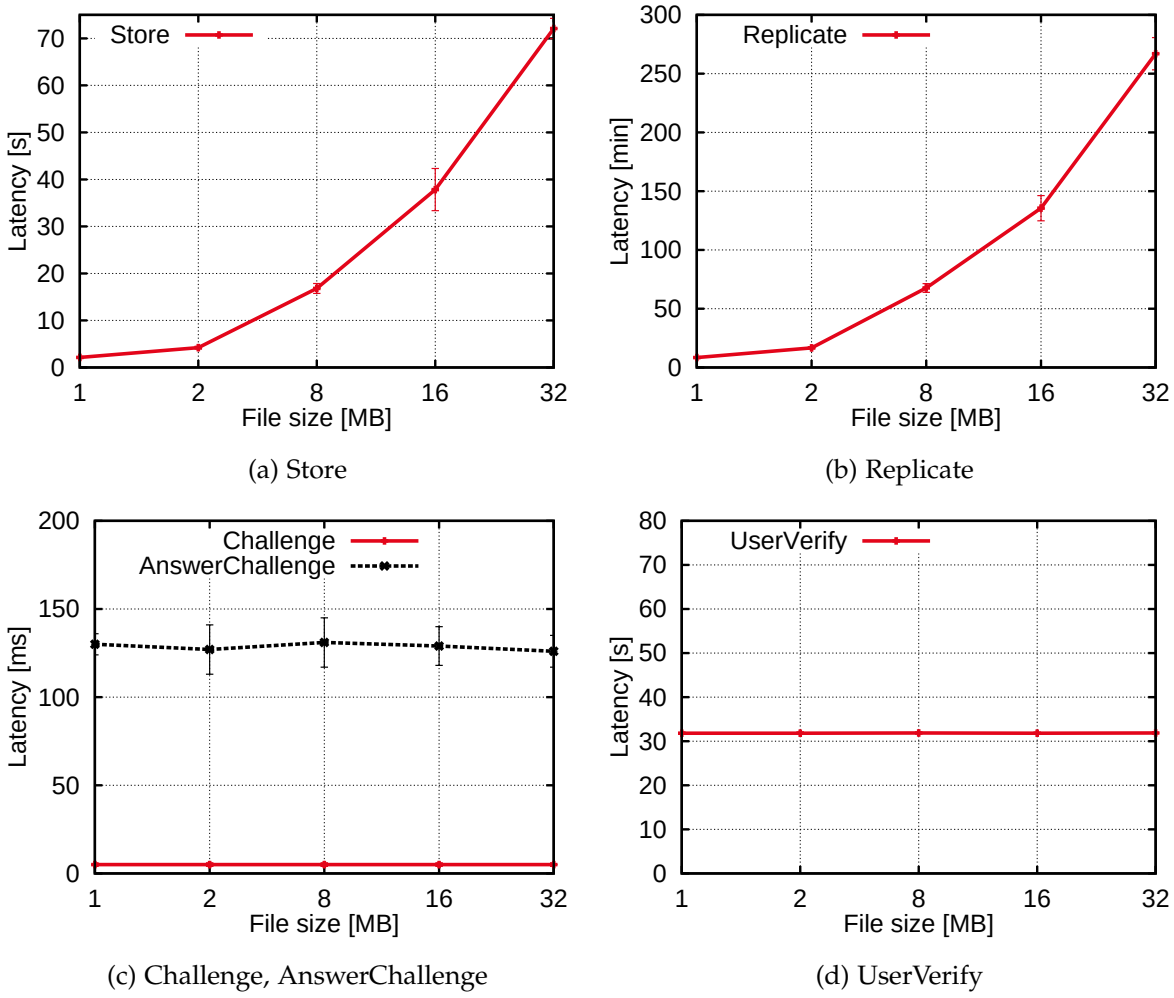
$$\text{RHS} \leftarrow \sigma + \sum_{j=1}^{s} \alpha_j \cdot P_j$$

He accepts if the equality holds, ie. if LHS = RHS, and if the response time of the server if below a threshold T_{thresh} (to be defined later on).

4.2.2 Performance

Table 4.3 Additive scheme — Default values

Parameter	(also called)	Value
BigExponentSize	$\|\|E\|\|$	10000
PuzzleTime	Δ_{Puzzle}	0.1 s



(a) Store

(b) Replicate

(c) Challenge, AnswerChallenge

(d) UserVerify

Figure 4.1: Performance of the additive scheme

In the following table, BS is the block size, and TS is the size of one tag.

Table 4.4 Additive scheme — Performance analysis

	IO [Byte]	Mod Pow	Mult	Add	Mod
Store	$n \cdot (\text{BS} + \text{TS})$	0	$n \cdot s$	$n \cdot s$	$2 \cdot n \cdot s$
Replicate	$2 \cdot n \cdot \text{BS}$	$n \cdot s$	0	$n \cdot s$	$n \cdot s$
Challenge	0	0	0	0	0
AnswerChallenge	$l \cdot (\text{BS} + \text{TS})$	0	$l \cdot (s+1)$	$l \cdot (s+1)$	$2 \cdot l \cdot s$
UserVerify	0	$2 \cdot l \cdot s$	$s \cdot (l+1)$	$s \cdot (l+1)$	$2 \cdot l \cdot s$

4.2.3 Discussion

Advantages. Theoretically, the scheme works and has the required functionalities : The file is only sent once; the storage overhead are acceptable for the client (of $n \cdot 256$B for the tags, plus 256B for the PRF key) and the server (of $n \cdot 256$B for the tags, *independent of the number of replicas* R). The STORE procedure has an acceptable complexity of $s + 1$ addition and s multiplications in $\mathbb{Z}_N$.

Disadvantages. Recall that we want each puzzle to be moderately intensive for the server, but the verification to be fast for the user. Here we use exponentiations of the form $p^{E^x} \mod n$ as a puzzle for the server; let Δ_{Server} be the time needed for one of those exponentiation (approximately defined with the bit length of p, E, x), then the server needs $\Delta_{\text{Server}} \cdot s \cdot n$ such puzzle to compute to completely replicate a file. By changing the size of E, we can adapt the overall difficulty (and hence overall replication time) for a given file size. When verifying, the user can first compute $E^x \mod \phi(n)$, greatly simplifying the computation. Let Δ_{Client} be the time needed to compute an exponentiation of the form $p^{E^x \mod \phi(n)} \mod n$. Among $s + l$ multiplication and $s + l + 1$ additions in $\mathbb{Z}_N$, the client has $s + l$ such exponentiations to compute. As the experiment will show, this is still rather intensive for the client (in the order of $30s$ for $l = 100$ and an exponent E of 10000 bits).

4.2.4 Security.

Puzzle hardness. In this scheme, the puzzle is a modular exponentiation of the form $p^E \mod N$, with a large $E \gg N$. This puzzle, proposed in [22], is inherently a sequential process, and is hard to parallelize; this prevent a malicious service provider with a high number of cores to be substantially faster than a mono-core machine to solve one puzzle.

If the client did not replicate correctly x sectors, he will have to redo x modular multiplication to answer the challenge. We assume that the number of threads is bounded; hence, by setting l large enough, the answer time, almost constant for l smaller or equal to the number of cores, rapidly increases when above the number of threads, as the puzzles queue up. Hence, for l big enough, the client is able to observe a significant deviation from the norm in the server answer time.

To translate this argument to multiple replicas, notice that each replica uses a different p_r (and a different E_r); to the best of our knowledge, this represent independent problems, and no tricks allow to speed up the computation of $p_j^{E_j} \mod N$ given $p_i^{E_i} \mod N$.

Extractability. The argument given in [29] transports almost directly to our scheme. For the proof a extractability, notice that for each successful run of VERIFY, the client learns $\mu_j = \sum_{(i,v_i)\in Q} v_i \cdot m_{i,j}^{(r)} \mod N$, for $j \in [1, s]$, and known v_i. Given the number of challenge l, for a number successful runs x of VERIFY (with the same query Q), the client gets a system of linear equations with $l \cdot r$ unknowns, and $s \cdot x$ equations. Hence, for $x \geq \frac{l \cdot r}{s}$, the client can find

the correct missing values $m_{i,j}^{(r)} \in [0, N[$ that validates the equations system, and reconstitute the r replicas and/or the original file (if it is in the set of queried replicas).

If the client only recovers a replica $F^{(r)}$ with the extractor algorithm, he can easily recover the original file F since REPLICATE is a deterministic function of p_r and E_r both stored by the client.

Storage allocation. Here, we explore the case where the service provider does not store correctly the file (i.e. only a certain percentage is correctly stored), and investigate the case where he needs to recompute the replication puzzle to answer the challenge.

Define the percentage of correctly stored sectors of a given file as ρ (e.g. $\rho = 1$ indicates that the file is correctly stored). Hence, the probability that a challenged sector is incorrectly stored and needs to be recomputed is $1 - \rho$. Since the users queries $l \cdot s$ sectors at each run of VERIFY, the expected number of values that need to be recomputed is $(1 - \rho) \cdot l \cdot s$. Let us define the puzzle time per sector, i.e. the time to compute $p_r^{E_r^{(j-1)\cdot s+j}}$, as Δ_{sector}. Hence, the server needs $(1 - \rho) \cdot l \cdot s \cdot \Delta_{\text{sector}}$ only to recompute the missing values before starting to answer the challenge.

Recall that the user is monitoring the time needed for the server to answer, and only accepts if it is below a certain threshold T_{thresh}. We tune E_r such that $\Delta_{\text{sector}} \gg \frac{T_{\text{thresh}}}{(1-\rho)\cdot l \cdot s}$. With a standard divide-and-choose approach, and a number of challenges that is linear given a security parameters κ, we can achieve that the probability that the verifier accepts given the incorrectly stored fraction of file $1 - \rho$ is negligible in κ.

Numerical example. We use a 64MB file, a block size of 8KB, a sector size of 256B, and a number of challenge $l = 100$. Recall that we use erasure coding; Hence, the file is only non-retrievable when it misses a certain percentage of its data, fixed here to 1%. Hence, we consider the file lost when more than 1% is missing, and want to detect when this is the case. Since the challenge selects $l \cdot s$ sectors, the expected number of missing sectors is $1\% \cdot 32 \cdot 100 = 32$ sectors per challenge. With a puzzle time of 100ms (corresponding approximately to an exponent of 10000 bits), the expected response time of the cheating service provider will be 3.2s, well above the honest response time of 150ms. The user will detect it with overwhelming probability.

Correct replication. This critera does not apply in this situation; the server being responsible for creating the replicas, the malicious user cannot try to encode additional information, and the *correct replication* criteria is trivially respected.

4.3 Multiplicative scheme

This second attempt aims to reduce the cost on the user for verifying a challenge. It is called multiplicative because the puzzle p^{E^x} is *multiplied* to the data block $m_{i,j}$ during the replication.

Key changes. The essence of the scheme remains the same, but as the name suggest, the two main mathematical operations used in the additive scheme, the addition and the multiplication, have been replaced by, respectively, the multiplication and the exponentiation.

4.3.1 Protocol

Store. This procedure is almost identical to the additive scheme. The user first divides the file into blocks and sectors of fixed size, and picks $s + n$ secrets, $\alpha_1 \cdots \alpha_s$ and $\text{secret}_1 \cdots \text{secret}_n$ using a keyed PRF. He computes the tags as follow :

$$\sigma_i \leftarrow \text{secret}_i \prod_{i=1}^{n} m_i^{\alpha_i} \mod N$$

He then sends the file F along with the tags σ_i to the service provider, and may delete F. The user also picks a prime p_r and a big exponent E_r for each desired replica r. He sends the p_r and the E_r to the server.

Replicate. This procedure is the same as on in the additive scheme, excepted for the actual formula : here the replica's sectors are computed as follow :

$$m_{i,j}^{(r)} \leftarrow m_{i,j} \cdot p_r^{E_r^{(j-1)\cdot s+i}} \mod N$$

Challenge. Here, we use the same technique as in the additive scheme : we transmit only two seeds $\text{seed}_i, \text{seed}_v$, that will generate numbers in the range $[1, \textsc{NumberOfBlocks}]$ and $[1, \text{RSA}\textsc{Modulus}]$, respectively. The size of the collection $i_1 \cdots i_l$, that is l, is fixed and known to both parties.

AnswerChallenge. First, the server computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. Then, he computes the following values σ and μ_j for each sector $j \in [1, s]$.

$$\sigma \leftarrow \prod_{(i,v_i)\in Q} \sigma_i^{v_i} \qquad \text{and} \qquad \mu_j \leftarrow \prod_{(i,v_i)\in Q} \left(m_{i,j}^{(r)}\right)^{v_i}$$

He transmits the s values of μ_j and the value of σ to the client.

UserVerify. The client also computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. He computes the Left Hand Side as follow :

$$\text{LHS} \leftarrow \prod_{j=1}^{s} {\mu_j}^{\alpha_j} \cdot \prod_{(i,v_i)\in Q} {\text{secret}_i}^{v_j}$$

For the Right Hand Side, similar to what is done in the additive scheme, he can computes P_j, $j \in [1, s]$:

$$P_j \leftarrow \prod_{(i,v_i)\in Q} {p_r}^{v_k E^{(i-1)\cdot s+j} \mod \phi(N)} \mod N$$

And finally computes :

$$\text{RHS} \leftarrow \sigma \prod_{j=1}^{s} P_j^{\alpha_j}$$

But notice that there is a much more efficient way to compute the RHS :

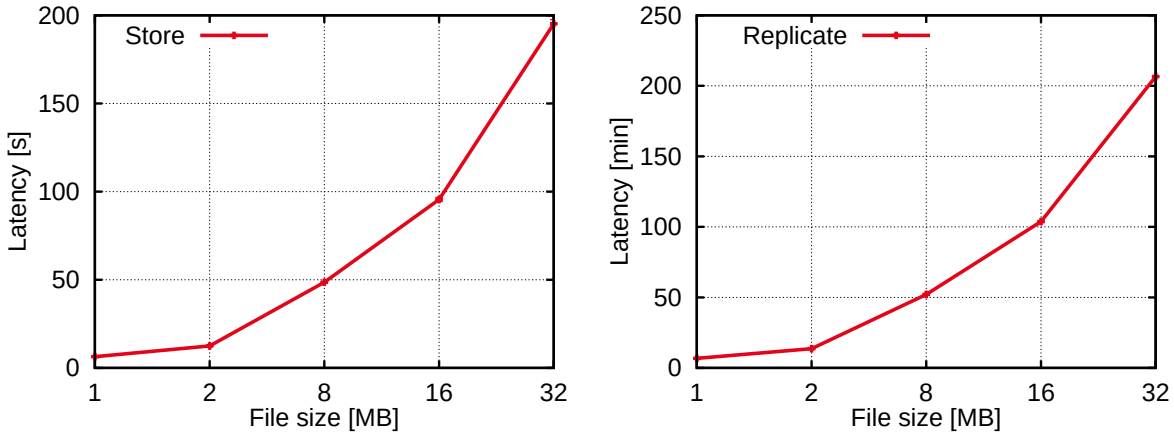
$$\text{RHS} \leftarrow \sigma \cdot {p_r}^{\sum_{j=1}^{s} \sum_{(i,v_i)\in Q} \alpha_j \cdot v_k \cdot E^{(i-s)\cdot s+j} \mod \phi(N)} \mod N$$

He accepts if the equality holds, ie. if LHS = RHS, and if the response time of the server if below a threshold T_{thresh} (to be defined later on).

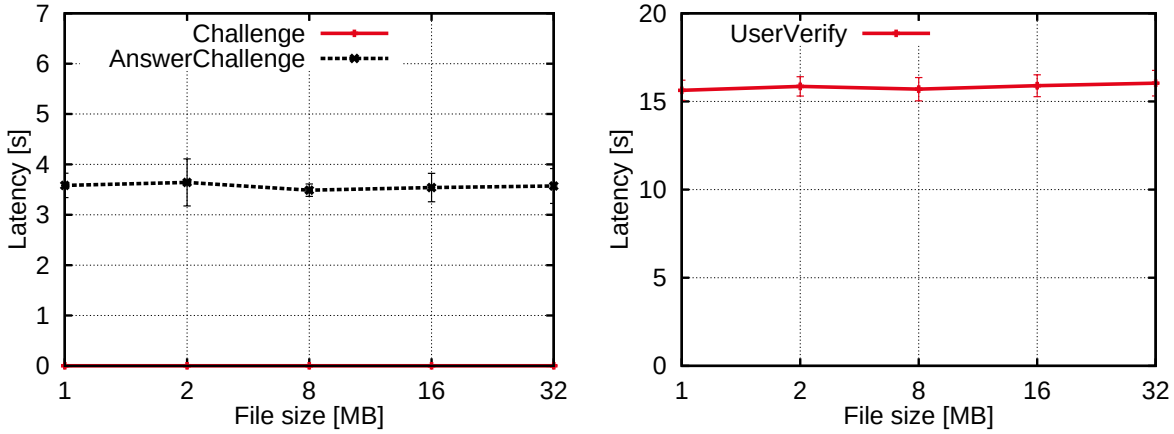
4.3.2 Performance

Table 4.5 Multiplicative scheme — Default values

Parameter	(also called)	Value
BIGEXPONENTSIZE	$\|\|E\|\|$	10000
PUZZLETIME	Δ_{Puzzle}	0.1 s



(a) Store (b) Replicate



(c) Challenge, AnswerChallenge (d) UserVerify

Figure 4.2: Performance of the multiplicative scheme

Table 4.6 Multiplicative scheme — Performance summary

	IO [Byte]	Mod Pow	Mult	Add	Mod
STORE	$n \cdot (\text{BS} + \text{TS})$	$n \cdot s$	$n \cdot s$	0	$n \cdot s$
REPLICATE	$2 \cdot n \cdot \text{BS}$	$n \cdot s$	$n \cdot s$	0	$n \cdot s$
CHALLENGE	0	0	0	0	0
ANSWERCHALLENGE	$l \cdot (\text{BS} + \text{TS})$	$s \cdot (l+1)$	$s \cdot (l+1)$	0	$s \cdot (l+1)$
USERVERIFY	0	$l \cdot s + 3s + l$	$l \cdot s + 2s + l$	$l \cdot s$	$l \cdot s + 3s + l$

4.3.3 Discussion

Advantages. This new scheme is almost twice as fast as the additive scheme in the operation UserVerify, due to the possible aggregation of puzzles. The puzzle time did not change significantly, since the main operation is computing the puzzle, which is the same than in the additive scheme.

Disadvantages. Unfortunately, the Store is now 2.5 times slower, due to the speed of the exponentiation and the multiplication, with respect to multiplication and addition.

4.3.4 Security.

Puzzle hardness. Since the puzzle is exactly the same as in the additive scheme (only the way of mixing the puzzle with the sectors change), the argument is exactly the same as in the additive scheme.

Extractability. Again, the argument given in [29] transports almost directly to our scheme. For the proof a extractability, notice that for each successful run of Verify, the client learns $\mu_j = \prod_{(i,v_i)\in Q} \left(m_{i,j}^{(r)}\right)^{v_i} \mod N$, for $j \in [1,s]$, and known v_i. Let us denote by $\mu^{(0)} := (\mu_1, \cdots, \mu_s)$. Similar to the case in the additive scheme, with x values $\mu^{(j)}$, we have $s \cdot x$ equations with $l \cdot r$ unknown. For $x \geq \frac{l \cdot r}{s}$, we use standard gaussian elimination, and reconstitute the r replicas.

There is, however, a small different with the additive scheme; in the former, we had a linear system of equations, which has a unique solution in $\mathbb{Z}_N$ for x large enough. In the multiplicative scheme however, the unknowns $m_{i,i}$ are raised to the power v_i, and depending on the order of $m_{i,j}$, several solutions in $\mathbb{Z}_N$ may exists. This is fixed by imposing a fixed pattern at the beginning of each data sector, which allows to recognize the correct root.

If the client only recovers a replica $F^{(r)}$ with the extractor algorithm, he can easily recover the original file F since Replicate is a deterministic function of p_r and E_r both stored by the client.

Storage allocation. The argument is identical to the one done in the additive scheme.

Correct replication. This critera does not apply in this situation; the server being responsible for creating the replicas, the malicious user cannot try to encode additional information, and the *correct replication* criteria is trivially respected.

4.4 Scheme with precomputation

This third attempt aims to reduce the overall replication time, while maintaining a non-negligible time to replicate one sector (which prevents the server from not replicating, and computing the answer on-the-fly when challenged).

Key changes. The replication time in the additive and multiplicative scheme does not scale as wished; Let us call the time to replicate a sector Δ_{Sector}. A rough estimate of the overall replication time is $\Delta_{\text{File}} = n \cdot s \cdot \Delta_{\text{Sector}}$. Notice that Δ_{Sector} should be fixed big enough to allow the user to detect if a given proportion of sector is not stored correctly at the service provider, and depends on the the time needed for the honest service provider to compute an answer, as well as the number of challenged blocks and sectors, but it does not really depends on the file size. Because of the linearity in the relation between Δ_{Sector} and Δ_{File}, it is really hard to bound the replication time for the (honest) server.

The idea explored here is to separate the puzzle issued to the service provider in two parts : first, a pre-computation P_1, that ideally could be tuned as whished, and an actual replication phase P_2 that requires P_1. Here, we would have again the relation $\Delta_{P_2} = n \cdot s \cdot \Delta_{\text{Sector}}$, but this time the overall puzzle time $\Delta_{\text{Replication}}$ would be the sum of Δ_{P_1} and Δ_{P_2}; the hope that a clever choice of pre-computation and replication procedure allows better control over $\Delta_{\text{Replication}}$.

With this change, Δ_{Sector} could be much smaller, as a cheating server not storing correctly a sector would need $\Delta_{P_1} + \Delta_{\text{Sector}}$ to answer a single challenged sector, since replicating any sector would require the knowledge of the pre-computation P_1.

Pre-computation P_1. The idea used here is to make the service provider pre-compute a range of exponentials of the form $p, p^E, p^{E^2}, \cdots, p^{E^\Lambda}$, and storing the results in memory; to replicate, each sector would be blinded with a coefficient b, which is computed as the product of some of those pre-computed exponentials. To be more accurate, each blinding factor would correspond to a set of w indices, distributed at random in $[0, \Lambda]$; once the exponentials pre-computed, replicating a sector would mean fetching w values in the table and multiplying them. Hence, Δ_{Sector} would mainly depend on w, which is a free parameter, and Δ_{P_1} would depend mainly on Λ, which is also free to be fixed as wished.

4.4.1 Protocol

Store. This algorithm is the same as in the multiplicative scheme; the tags creation is done as follow :

$$\sigma_i \leftarrow \text{secret}_i \prod_{i=1}^{n} m_i^{\alpha_i} \mod n$$

Replicate. First, for each replica r, the server first pre-compute $\text{Powers}_{p_r} = [p_r^{E^0}, p_r^{E^1}, \cdots, p_r^{E^\Lambda}]$, for a pre-agreed Λ, which typically depends on (and is larger than) the total number of

sectors; for instance, $\Lambda = ns\lambda$, with λ being a percentage higher than 100%. Let's denote by $\text{Powers}_{p_r}(i) = p_r^{E^i}$, and a PRF $\Omega(i,j)$ that output a list $i_1, i_2 \cdots i_w$ of indices in $[0, \Lambda]$. This phase is made of $\Lambda + 1$ modular exponentiations; let Δ_{exp} be the time to compute x^E, then the server can compute all the exponentials in $\Lambda \cdot \Delta_{\text{exp}}$, since $(p_r^{E^x})^E = p_r^{E^{x+1}}$.

Then, for each block i and sector j, the server will compute and store

$$m_{i,j}^{(r)} \leftarrow m_{i,j} \cdot \prod_{(i_k \in \Omega(i,j))} \text{Powers}_{p_r}(i_k) \mod N$$

Challenge. As usual, the clients sends two seeds $\text{seed}_i, \text{seed}_v$, that will generate numbers in the range $[1, \text{NumberOfBlocks}]$ and $[1, \text{RSAModulus}]$, respectively.

AnswerChallenge. First, the server computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. Then, he computes the following values σ and μ_j for each sector $j \in [1, s]$.

$$\sigma \leftarrow \sum_{(i,v_i) \in Q} \sigma_i^{v_i} \qquad \text{and} \qquad \mu_j \leftarrow \sum_{(i,v_i) \in Q} \left(m_{i,j}^{(r)} \right)^{v_i}$$

He transmits the s values of μ_j and the value of σ to the client.

UserVerify. The client also computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. He computes the Left Hand Side as follow :

$$\text{LHS} \leftarrow \prod_{j=1}^{s} {\mu_j}^{\alpha_j} \prod_{(i,v_i) \in Q} {\text{secret}_i}^{v_j}$$

For the Right Hand Side, he computes :

$$\text{RHS} \leftarrow \sigma \prod_{j=1}^{s} \prod_{(i,v_i) \in Q} {p_r}^{v_i \alpha_j \sum_{(i,v_i) \in Q} E^i \mod \phi(N)}$$

Which can also be computed more efficiently as :

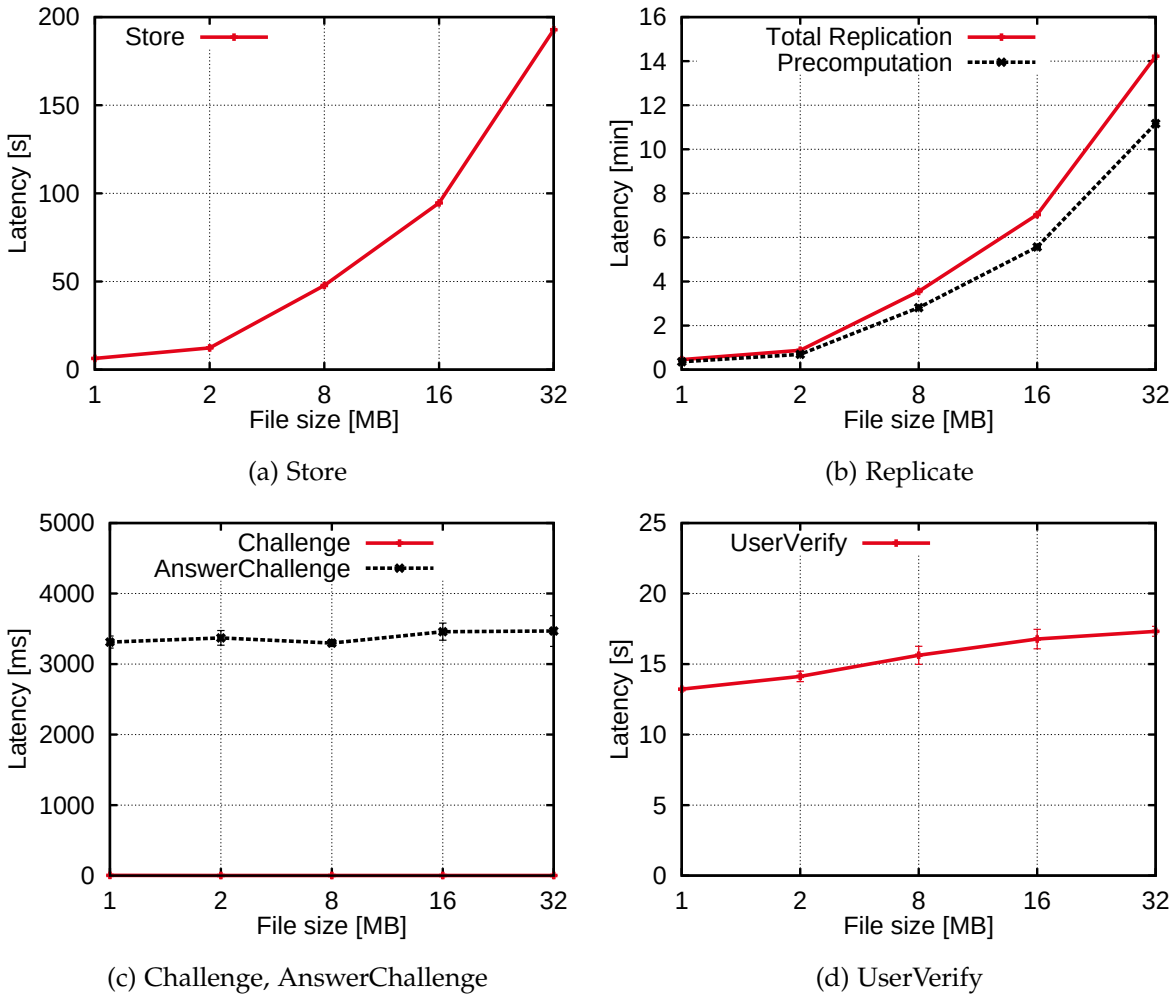
$$\text{RHS} \leftarrow \sigma \cdot {p_r}^{\sum_{j=1}^{s} \sum_{(i,v_i) \in Q} \alpha_j \cdot v_k \cdot \sum_{(i,v_i) \in Q} E^i \mod \phi(N)} \mod N$$

He accepts if the equality holds, ie. if LHS = RHS, and if the response time of the server if below a threshold T_{thresh} (to be defined later on).

4.4.2 Performance

Table 4.7 Scheme with precomputation — Default values

Parameter	(also called)	Value
NumberOfMixedInPrimesPerSector	w	35
PercentageOfPuzzleToGenerate	λ	170
TotalNumberOfPuzzleToGenerate	Λ	$ns\lambda$



(a) Store

(b) Replicate

(c) Challenge, AnswerChallenge

(d) UserVerify

Figure 4.3: Performance of the scheme with precomputation

Table 4.8 Scheme with precomputation — Performance summary

	IO [Byte]	Mod Pow	Mult	Add	Mod
Store	$n \cdot (\mathrm{BS} + \mathrm{TS})$	$n \cdot s$	$n \cdot s$	0	$n \cdot s$
Replicate	$2 \cdot n \cdot \mathrm{BS}$	Λ	$n \cdot s \cdot (w+1)$	0	$n \cdot s \cdot (w+1)$
Challenge	0	0	0	0	0
AnswerChallenge	$l \cdot (\mathrm{BS} + \mathrm{TS})$	$l \cdot (s+1)$	$l \cdot (s+1)$	0	$l \cdot (s+1)$
UserVerify	0	$s \cdot l \cdot w \cdot +2s + l$	$2 \cdot l \cdot s + 2s + l$	$w \cdot l \cdot s$	$2 \cdot l \cdot s + 3 \cdot s + l$

4.4.3 Discussion

Advantages. The very interesting idea of this pre-computation puzzle is the ability to tune the replication time as wanted; here, Λ only has impact for the replication time, and can be (almost) freely choosed. This was not the case in the two previous version, where the user was using a trapdoor to lighten the computation imposed to the server : tuning the problem on the client side inevitably altered the problem on the server side.

Disadvantages. The verification time for the user is now really significant, especially for an operation that needs to be run often. In addition, the latency of UserVerify is not anymore constant with respect to the file size; the latency now increases with the file size, even though the algorithm itself does not depend on it and the IO accesses, l and s are constant. What varies is the range of the puzzles p^{E^x}, since x is distributed in $[0, ns\lambda]$ which depends on the file size because of n. With n increasing, so is, on average, x and the length of the exponent of p, which takes more time.

4.4.4 Security

Unfortunately, this scheme is much less secure than the previous one.

Size of Λ. The novelty of using one pre-computation as a main puzzle for replication comes with a downside : the malicious service provider might try to store the pre-computation instead of the replica, as it allows to replicate quickly the queried sectors and answer correctly and in time to the client. Hence, it is necessary to tune the parameters such that storing the pre-computation is not an option for the rational malicious service provider. In this scheme, this is ensured by having Λ bigger than 100%, thus making the pre-computation storage requirement *bigger* than the actual replica.

Size of w. Following this argument, it is necessary to have $w > 1$; indeed, with $w = 1$, one sector corresponds to exactly one exponential of the form $p^{E^x} \mod n$; since both have the same size, it would be possible for the server to store with no overhead the p^{E^x} instead of the $m_{i,j}^{(r)}$. With that, the service provider is able to answer quickly and correctly the challenges, with a data redundancy less than desired since the file cannot be reconstructed from the sequence of p^{E^x}.

Puzzle hardness. The main problem that arise with this specific computation is that the malicious service provider can cheat : an efficient strategy would be to compute, and store, the values $p^{E^k}, p^{E^{2k}}, p^{E^{3k}}, \cdots$. With $k > 1$, the storage required is less than the whole pre-computation; Given any file size, since k is arbitrarily fixed, the sequence can also be *less than the size of the replica*. To answer correctly a challenge involving the blinding factor p^{E^x}, the malicious service provider would locate the closest smaller power $p^{E^{nk}}$ stored, and compute

the difference $x - nk$ (at most $k - 1$) exponentials in $\mathbb{Z}_N$. This is clearly a good and efficient strategy that is hard to detect.

Extractability and Storage allocation. Considering the original file F, the extractability translates directly from [29], as usual. For the replicas $F^{(1)}, \cdots, F^{(R)}$, since the server has a way of cheating and making the verifier accept with good probabilty *without* storing the replicas, but by storing parts of the precomputation (which are useless to reconstruct the original file), this scheme does not satisfies both criteria.

Correct replication. This critera does not apply in this situation; the server being responsible for creating the replicas, the malicious user cannot try to encode additional information, and the *correct replication* criteria is trivially respected.

Chapter 5

Mirror

We now present MIRROR, which was selected as the best alternative to the schemes presented in chapter 4.

This fourth scheme provides a new degree of freedom for tuning the puzzle time by using LSFR and time-lock puzzles (both introduced in Section 3.4).

5.1 Protocol

Store. The user first divides the file into blocks and sectors of fixed size, and picks s secrets, $\epsilon_1 \cdots \epsilon_s$ using a keyed PRF. He computes the tags as follow, for $i \in [1, n]$:

$$\sigma_i \leftarrow \prod_{j=1}^{s} m_{i,j}^{\epsilon_j} \mod N$$

He then sends the file F along with the tags σ_i to the service provider, and may delete F.

Generating the replication parameters. In this scheme, the puzzle is based on RSA time lock puzzles (see 3.4.1 for more details), and on two tuples of FSR (see 3.4.2 for more details). Recall that a FSR is defined by a feedback function F, and an initial state S. In what follow, we explain the procedure for one tuple of FSR, the procedure for the second being identical; then, we will explain the use of the two tuples.

The client will generate some FSRs for the server, one per replica. We will exploit the asymmetry presented in 3.4.1 : for each of the server FSR of length λ^*, the client will have a matching FSR of length λ outputting the same sequence, with $\lambda < \lambda^*$. To achieve this, the client proceeds as follow :

1. Define f as p'. Let the field $\mathbb{F}$ be $\mathbb{Z}_f$.
2. The clients finds an element g of order f in $\mathbb{Z}_N^*$.
3. Then, he chooses two feedback polynomial $f(x), f^*(x)$ over $\mathbb{F}$:

$$f(x) = x^\lambda + \sum_{i=1}^{\lambda} \alpha_i \cdot x^{i-1}$$

$$f^*(x) = x^\lambda + \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot x^{i-1}$$

such that $f^*(x)$ is a multiple of $f(x)$. Here, the user has a number of choice selecting the parameters that we will discuss later. For security reasons, it is necessary to ensure that $\alpha_1^* \geq 2$.

4. He then picks at random an initial state $S_0 = (a_1, \cdots, a_\lambda) \in \mathbb{F}^\lambda$. The *small LFSR* is defined by $f(x)$ and S_0, and its output sequence is defined by $a_{t+\lambda+1} = \sum_{i=1}^{\lambda} \alpha_i \cdot a_{t+i}$ for $t \geq 0$.

5. Since $f^*(x)$ is a multiple of $f(x)$, it also holds that $a_{t+\lambda^*+1} = \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot a_{t+i}$ for $t \geq 0$. Hence, the *big LFSR* is defined by $f^*(x)$ and $S_0^* = (a_1, \cdots a_\lambda, a_{\lambda+1}, \cdots, a_{\lambda^*})$. His output sequence is identical to the small LFSR, only it requires more computation per output element.

6. To further increase the asymmetry between the client and the server, we combine those LFSR with time-lock puzzles. On the user side, we simply define the final sequence $\mathcal{U}_{\text{seq}}$ as $g^{a_0}, g^{a_1}, \cdots$ and so on, each exponentiation being done modulo N as usual.

7. For the server, the user will first compute the following initial state $\widetilde{S_0^*} = (g^{a_1}, \cdots, g^{a_{\lambda^*}})$, each exponentiation being done modulo N as usual. For convenience, we define

$$(\widetilde{a}_1, \cdots, \widetilde{a}_{\lambda^*}) := (g^{a_1}, \cdots, g^{a_{\lambda^*}})$$

8. Then, he discloses the coefficients $(\alpha_1^*, \cdots, \alpha_{\lambda^*}^*)$ along with the initial state $(\widetilde{a}_1, \cdots, \widetilde{a}_{\lambda^*})$ to the server. The server will use a FSR that is *not linear* : instead, he will compute the element $\widetilde{a}_{t+\lambda^*+1} = \prod_{i=1}^{\lambda^*} \widetilde{a}_i^{\alpha_i^*} \mod N$.

9. We define the final server sequence $\mathcal{S}_{\text{seq}}$ as $\widetilde{a}_0, \widetilde{a}_1, \cdots$ and so on. Notice that the user sequence $\mathcal{U}_{\text{seq}}$ and the server sequence $\mathcal{S}_{\text{seq}}$ are identical.

10. To sum up, the replication parameters for one replica, regarding the field $\mathbb{F}$, is given by, for the user :

$$(g, (\alpha_1, \cdots, \alpha_\lambda), (a_1, \cdots, a_\lambda))$$

And for the server :

$$((\alpha_1^*, \cdots, \alpha_{\lambda^*}^*, (\widetilde{a}_1, \cdots, \widetilde{a}_{\lambda^*})))$$

11. For another replica u, the user keeps the same polynomials $f(x)$ and $f^*(x)$, but selects a new initial state for the small LFSR $S_0^{(r)} = \left(a_1^{(u)}, \cdots, a_\lambda^{(u)}\right)$, and recomputes the initial state $\widetilde{S_0^*}^{(u)} = \left(\widetilde{a}_1^{(u)}, \cdots, \widetilde{a}_{\lambda^*}^{(u)}\right)$ used by the server.

12. Hence, for R replicas, with respect to $\mathbb{F}$, the replicas parameters are defined as follow : for the client,

$$\left(g, (\alpha_1, \cdots, \alpha_\lambda), \left(a_1^{(1)}, \cdots, a_\lambda^{(1)}\right) \cdots \left(a_1^{(R)}, \cdots, a_\lambda^{(R)}\right)\right)$$

And for the server :

$$\left((\alpha_1^*, \cdots, \alpha_{\lambda^*}^*), \left(\widetilde{a}_1^{(1)}, \cdots, \widetilde{a}_{\lambda^*}^{(1)}\right) \cdots \left(\widetilde{a}_1^{(R)}, \cdots, \widetilde{a}_{\lambda^*}^{(R)}\right)\right)$$

The procedure presented above yield, given $f = p'$, a couple of FSR (one for the client, one for the server) per replica. We repeat the procedure with $f = q'$, and in this case we name h the element of order q', call b_i every state element previously called a_i, and β_i every coefficient previously called α_i.

To conclude, we run the aforementioned procedure twice, which yield the following parameters for the client :

$$\left(g, h, (\alpha_1 \cdots \alpha_\lambda), (a_1^{(k)} \cdots a_\lambda^{(k)})_{1 \leq k \leq R}, (\beta_1 \cdots \beta_\lambda), (b_1^{(k)} \cdots b_\lambda^{(k)})_{1 \leq k \leq R}\right)$$

And for the server :

$$\Pi := \left((\alpha_1^* \cdots \alpha_{\lambda^*}^*), (\widetilde{a}_1^{(k)} \cdots \widetilde{a}_{\lambda^*}^{(k)})_{1 \leq k \leq R}, (\beta_1^* \cdots \beta_{\lambda^*}^*), (\widetilde{b}_1^{(k)} \cdots \widetilde{b}_{\lambda^*}^{(k)})_{1 \leq k \leq R}\right)$$

Replicate. Upon reception of Π, the service provider will start computing a number of R of replicas. For each replica u, the server computes the new sectors $m_{i,j}^{(u)}$ as

$$m_{i,j}^{(u)} \leftarrow m_{i,j} \cdot \widetilde{a}_{s \cdot i + j}^{(u)} \cdot \widetilde{b}_{n \cdot s - (s \cdot i + j)}^{(u)}$$

That is, the original sectors $m_{i,j}$ is multiplied by the server's FSR first (in $\mathbb{Z}_{p'}$) output sequence in the forward order, and to the server's second FSR (in $\mathbb{Z}_{q'}$) output sequence in the backward order.

Since it is a priori not possible to produce efficiently the backward sequence of a FSR (it would require the existence and knowledge of a feedback function $S_t = F(S_{t+1})$), the replication is essentially done in two steps :

- First, in the forward direction, with i and j increasing, we compute $m_{i,j}^{(u)} \leftarrow m_{i,j} \cdot \widetilde{a}_{s \cdot i + j}^{(u)}$
- Then, in the backward direction, with i and j decreasing, we compute $m_{i,j}^{(u)} \leftarrow m_{i,j}^{(u)} \cdot \widetilde{b}_{s \cdot i + j}^{(u)}$

Hence, we do two passes over the replica's sectors.

Challenge. As usual, the clients sends two seeds : $\text{seed}_i, \text{seed}_v$, that will generate numbers in the range $[1, \text{NumberOfBlocks}]$ and $[1, \text{RSAModulus}]$, respectively. In addition, he selects a set of replicas to challenge, and sends their indices to the server.

AnswerChallenge. First, the server computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. Then, he computes the following values σ and μ_j for each sector $j \in [1, s]$.

$$\sigma \leftarrow \prod_{(i, v_i) \in Q} \left(\sigma_i \cdot \prod_{j=1}^{s} \prod_{r \in k} m_{i,j}^{(r)}\right)^{v_i} \qquad \text{and} \qquad \mu_j \leftarrow \sum_{(i, v_i) \in Q} {m_{i,j}}^{v_i}$$

He transmits the s values of μ_j and the value of σ to the client.

UserVerify. The client also computes the sequences $i_1 \cdots i_l$ and $v_1 \cdots v_l$ using the PRF. He computes the Left Hand Side as follow :

$$\text{LHS} \leftarrow \prod_{(i,v_i)\in Q} \left(\sigma_i \cdot \prod_{j=1}^{s} \prod_{r\in k} g^{a_{i,j}^{(r)}} \cdot h^{b_{i,j}^{(r)}} \right)^{-v_i}$$

As usual, the LHS can be computed much more easily by grouping the g and h, and computing the elements linearly in the exponent.

For the Right Hand Side, he proceeds as follow :

$$\text{RHS} \leftarrow \prod_{j=1} \mu_j^{\epsilon_j + \|k\|}$$

He accepts if the equality holds, ie. if LHS = RHS, and if the response time of the server if below a threshold T_{thresh} (to be defined later on).22

Chapter 6

Security analysis

We now perform a security analysis on MIRROR; this is adapted from [5]. Similar to what was done in chapter 4, we discuss *(1)* the puzzle hardness, *(2)* the extractability property, *(3)* the storage allocation property, and *(4)* the correct replication property.

Puzzle hardness. In this scheme, the puzzles mainly relies on the server FSR; in particular, the puzzle time is exactly the time to recompute an element from the output sequence. First, let us focus on one replica. Recall that we denote the output sequence of the first server FSR by $\widetilde{a}_1, \widetilde{a}_2, \cdots$, and by $\widetilde{b}_1, \widetilde{b}_2, \cdots$ the output of the second server FSR.

Generally speaking, the element-wise sum of two LFSR[1] sequence; one could consider the sequence $(\widetilde{a}_i \cdot \widetilde{b}_{n \cdot s - i})$ as a LFSR. However, both sequences runs in different direction. Since we require both α_1^* and β_1^* to be ≥ 2, and the server does not know the factorization of N, he cannot inverse nor α_1^* nor β_1^* which would be necessary to compute a sequence in the opposite direction (i.e. computing α_i^* from the state $\alpha_{i+1}^*, \cdots, \alpha_{i+\lambda^*+1}^*$). Hence, we need to consider both sequence separately.

The idea is to give a lower bound on the time to compute one value $\widetilde{a}_i$ given some state (we do not specify the indexes in the sequence for this state on purpose, as the cloud provider might know or have computed arbitrary values except $\widetilde{a}_i$). Generally speaking, the server can find some v_i such that the relation $\prod_{i=1}^{n \cdot s} \widetilde{a}_i^{v_i} = 1 \mod N$ holds, which allows him to compute some unknown output element $\widetilde{a}_j$ from known values $\widetilde{a}_i$. By introducing a new cryptographic problem that reduces to the Decisional Diffie-Hellman Problem, the authors of [5] show that finding such a valid relation is the only way to compute an unknown output element $\widetilde{a}_j$ from known values $\widetilde{a}_i$, and that the effort to *compute* the relation (all the efforts to *find* the relation are ignored) is lower bounded by the effort to compute once the exponentiation with the biggest exponent.

Assume W.L.O.G that the biggest exponent is α_1^*; Define by T_{Mult} the average time to perform a multiplication in $\mathbb{Z}_N^*$. The pessimistic lower-bound to compute the unknown value $\widetilde{a}_i$ is $3/2 \cdot \text{ceil}\left(\log_2(\alpha_1^*)\right) \cdot T_{\text{Mult}}$, that is, the time to perform an single exponentiation with an exponent of ceil $\left(\log_2(\alpha_1^*)\right)$ bits.

As a conclusion, we can lower bound the time needed by a malicious adversary to cheat the puzzle given the bit length of the bigger coefficient in the server LFSR. Since this bit length

[1] We sometimes call "LFSR" the server FSR; it *is* a LFSR in the exponent of g, and the server could use a real LFSR — and not an exponential FSR — if he had knowledge of g.

is freely fixed, we can make it as costly as wanted for the malicious server. Of course, this is reflected in the replication time for the honest server.

Extractability. The argument given in the multiplicative schemes applies directly here, since only the puzzle changed.

Storage allocation. This argument given in the additive scheme almost directly translate here. Following the argument done for the puzzle hardness, the time to recompute a missing sector is lower bounded, and we can freely modify this lower bound.

Hence, we tune the lower bound on the time to recompute one value from the FSR output sequence, i.e. we tune Δ_{sector} such that :

$$\Delta_{\text{sector}} \gg \frac{T_{\text{thresh}}}{(1-\rho) \cdot l \cdot s}$$

For a given T_{thresh} defined given the standard response time from the server, and a fixed ρ defined by the erasure-coding applied on the file, and with a number of challenges that is linear given a security parameters κ, we can achieve that the probability that the verifier accepts given the incorrectly stored fraction of file $1-\rho$ is negligible in κ.

Correct replication. This critera does not apply in this situation; the server being responsible for creating the replicas, the malicious user cannot try to encode additional information, and the *correct replication* criteria is trivially respected.

Chapter 7

Performance analysis

7.1 Implementation setting

We implemented a prototype of Mirror in Scala [27]; in addition, we implemented the multi-replica PDP scheme from Curtmola et al. [15]. We will use the MR-PDP scheme as a comparison for our results. In our implementation, we relied on SHA256 for the default hash function, and on the JVM build-in random number generator.

As before, our implementation was deployed on a private network consisting of two machines directly connected to each other; the description of the machines is given in Table 4.1. The speed of the link between the machines was set to 100Mbps, and to emulate a realistic Wide Area Network simulation, we used the tool Netem [25] to shape the traffic following a Pareto distribution with a mean of 20ms, and a variance of 4ms as done in [16].

In our setup, one servers spawns a certain number of client, while the other one acts as a server and listen for client requests. Each client is created with a list of tasks, and will execute those tasks sequentially. We evaluate the latency (and throughput) incurred for various operations, and compare Mirror to MR-PDP, by varying the number of challenges, the block size, the number of replicas, and the file size. Each data point in the plots is averaged over 10 independent measurements; where appropriate, we include the corresponding 95% confidence intervals.

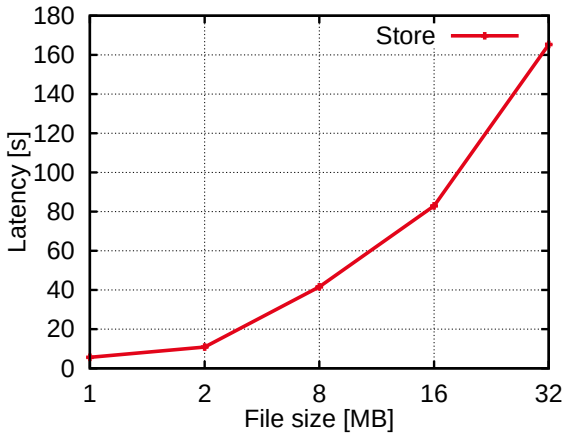
Table 7.1 summarizes the default parameters assumed in our setup.

Table 7.1 Default parameters used in the evaluation.

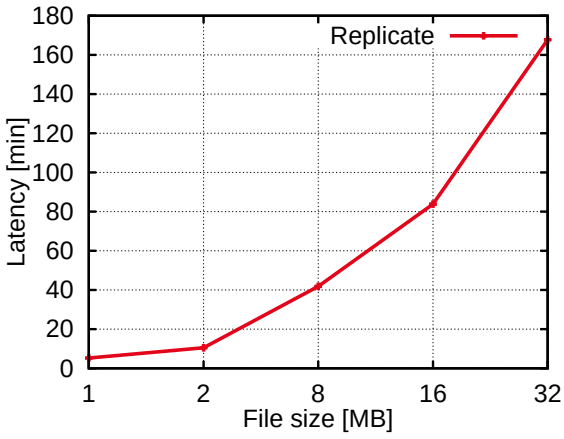
Parameter	**(also called)**	**Value**
File size		64 MB
Block size		8192 KB
Sector size		256 KB
RSA Modulus Length		2048 bits
PRF KeyLength		80 bits
Number of Challenged Blocks	l	40
Length of Secret LFSR	λ	2
Length of Public LFSR	λ^*	15
Fraction of stored sectors	ρ	0.9
Threshold T	T	0.8 s
Number of replicas	r	2

7.2 Performance

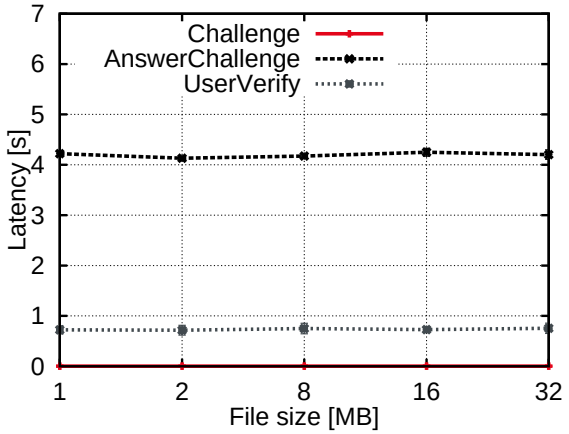
As usual, we start by presenting the latency of the operations, with respect to the file size.



(a) Store



(b) Replicate



(c) Challenge, AnswerChallenge, UserVerify

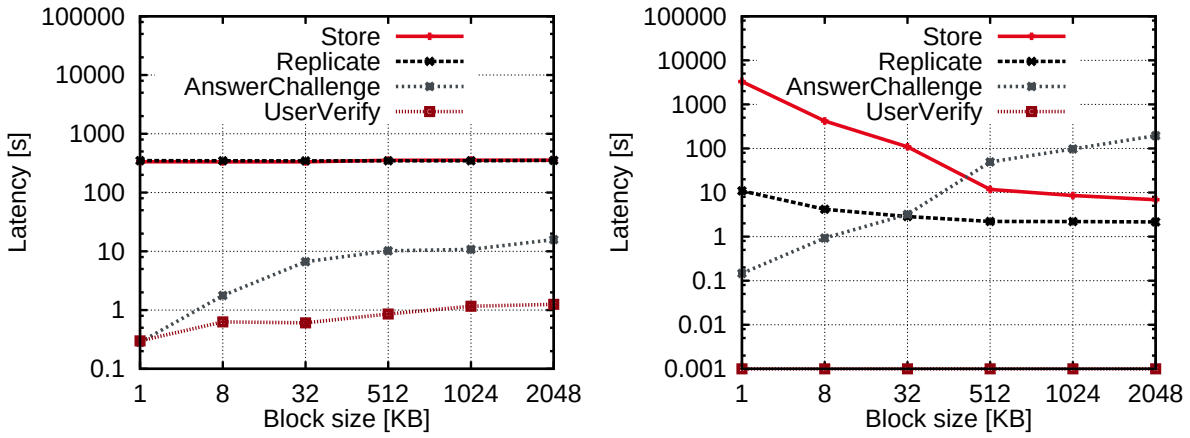
Figure 7.1: Performance of the LFSR-based scheme

Table 7.2 LFSR-based scheme — Performance analysis

	IO [Byte]	**Mod Pow**	**Mult**	**Add**	**Mod**
Store	$n \cdot (\mathrm{BS} + \mathrm{TS})$	$n \cdot s$	$n \cdot s$	0	$n \cdot s$
Replicate	$4 \cdot n \cdot \mathrm{BS}$	$2 \cdot n \cdot s \cdot \lambda^*$	$2 \cdot n \cdot s \cdot \lambda^*$	0	$4 \cdot n \cdot s$
Challenge	0	0	0	0	0
AnswerChall.	$l \cdot (r \cdot \mathrm{BS} + \mathrm{TS})$	$l \cdot (s+1)$	$l \cdot (s \cdot (1+r)+1)$	0	$l \cdot (s \cdot (1+r)+1)$
UserVerify	0	$s+2$	$l+s+1$	$l \cdot s \cdot r + l + s$	$s+1$

7.3 Evaluation results

Block size. Before evaluating the performance of Mirror, we analyze the impact of the block size on the different procedures, for MR-PDP and our scheme. Figure 7.2 shows that a block size of 8 KB yields balanced performance; this is also the default size assumed in MR-PDP [15].



(a) Mirror

(b) MR-PDP

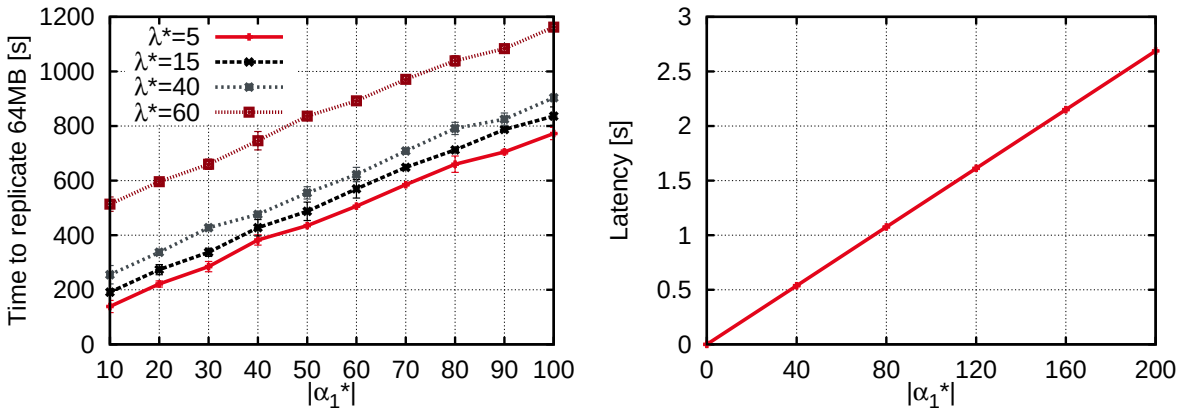
Figure 7.2: Impact of the block size

Impact of the bit length of α_1^*. Recall that we expressed the following relation between the threshold T_{thresh}, the multiplication time in $\mathbb{Z}_N$ T_{mult}, and the percentage of missing sectors ρ:

$$\text{ceil}\left(\log_2(\alpha_1^*)\right) \geq \frac{T_{\text{thresh}}}{3 \cdot l \cdot s \cdot (1-\rho) \cdot T_{\text{mult}}}$$

As explained previously, the bit length of α_1^* plays a major role in the security of Mirror; in addition, both α_1^* and λ^* impact the file replication time. Therefore, we analyze the impact of those parameters on both the replication time, and on the time to detect misbehavior. To maximize the performances of Mirror, we set the size of the remaining coefficients $\alpha_2^*, \cdots, \alpha_{\lambda^*}^*$ to 1 bit, i.e. either 0 or 1.

Figure 7.3a shows that the replication time grows linearly with the bit length of α_1^*; in addition, the bigger λ^*, the bigger it takes to replicate. In addition, the bit length of α_1^* considerably affects the time to detect a provider who did not replicate correctly some sectors, as shown in figure 7.3b, where we plot the lower bound previously discussed.



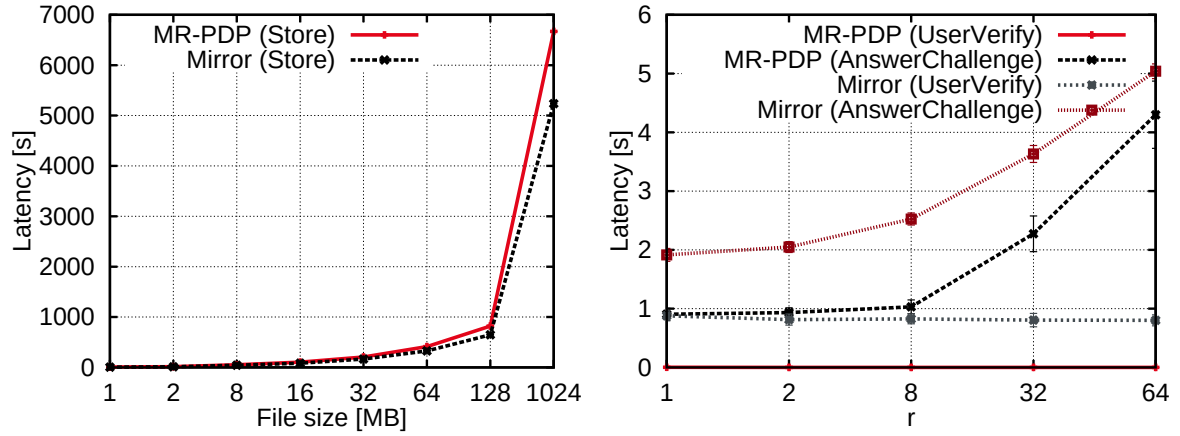
(a) Impact of $\|\alpha_1^*\|$ and λ^* on Replicate

(b) Impact of $\|\alpha_1^*\|$ for the malicious server

Figure 7.3: Impact of replication parameters on Mirror

We select 70 for the bit length of α_1^*, and 15 for λ^*, which gives reasonable replication time while allowing users to detect misbehavior.

Numerical example. For $\rho = 0.9$, $\|\alpha_1^*\| = 70$, $\lambda^* = 15$. and $r = 10$, and given $l = 40$ challenges, a misbehaving service provider would need an additional 880ms in order to provide a correct response. Since the typical response time of a honest cloud provider is around 2 seconds, an increase of 880ms (which is a very pessimistic lower bound on the real additional time needed) is easily detectable by users.



(a) Store

(b) UserVerify

Figure 7.4: Performance comparison of Store and UserVerify

Store. Figure 7.4a shows that Mirror is faster than MR-PDP for running Store; it is due to the fact that the tag creation takes less exponentiation in Mirror than in MR-PDP. For instance, running Store on a 64MB file is about 20% faster in Mirror than in MR-PDP.

Verify. Figure 7.4b shows the time to perform the two procedures AnswerChallenge and UserVerify; we omit Challenge, as its latency is low and constant (around 2ms). Our finding shows that the time to answer a challenge in Mirror is almost twice as much as in MR-PDP. In addition, where the UserVerify in MR-PDP is almost negligible, in Mirror it takes around 1 second; both are constant for a fixed l.

This difference is mainly explained by the fact that Mirror processes the data per sectors, whereas MR-PDP processes it by blocks; when challenged, Mirror has to process the s sectors in order to ensure the extractability property. In contrast, in MR-PDP each block contains only one sector, which provides data possession but not extractability. We argue that the overhead introduced is easily tolerable for the clients, as running UserVerify takes less than a second for a 64MB file.

We further explore the difference in throughput between MR-PDP and Mirror in AnswerChallenge in figure 7.5. For identical implementation settings, the maximum number of clients served per second is around 12 for MR-PDP, but only around 6 for Mirror. Again, we argue that it is mainly due to having a POR, which forces a processing based on sectors.

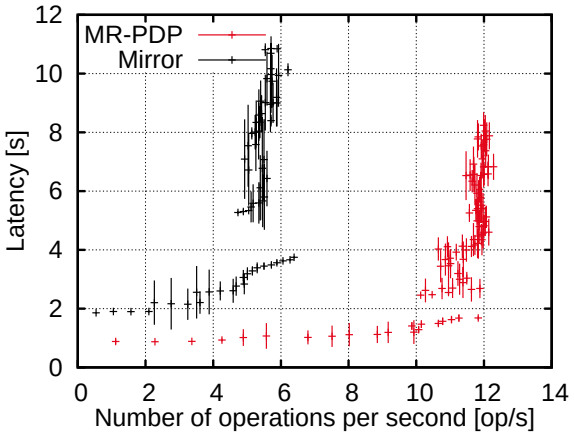


Figure 7.5: Performance of AnswerChallenge

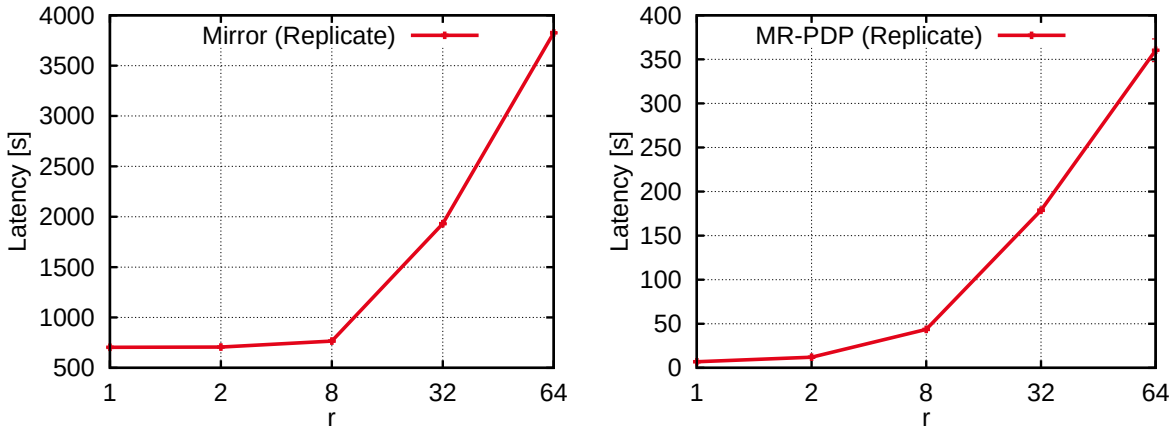
Replicate. Finally, for our choice of parameters, we show in figure 7.6 the latency of the Replicate operation in Mirror and MR-PDP.

Clearly, the latency increase with the number of replicas. Since we use a multi-threaded implementation, replicating can be done in parallel; as the number of parallel tasks increase, the threads in our thread pool are exhausted, and the system saturates, increasing sharply the latency.

Recall that the parallel processing — on a multicore infrastructure — of the replicas is fully exploited in Mirror; in contrast, in MR-PDP, the users have to create themselves the replicas, and also needs to send the R replicas over the network, process that cannot be parallelized.

Figure 7.6a shows the latency incurred for Replicate in Mirror. Since our scheme relies on cryptographic puzzles, the replication processes uses considerably more ressources than

MR-PDP; we argue that this is an effort done once per file, and it can be performed *offline* — i.e., the cloud provider can process the replicas using offline resources in the back-end.



(a) Mirror (b) MR-PDP

Figure 7.6: Performance of the Replicate procedure

7.4 Storage overhead

The storage overhead on the client is of n tags, that is $n \cdot \|N\|$, since each tag is between 0 and $N-1$ inclusive. On the server, the overhead is the same, as he needs to store the tags as well; replicas are not counted, as their would be part of the agreement between the user and the server. For the standard RSA modulus size of 2048 bits, the storage for both client and server is of $n \cdot 256$B. For a 64MB and a block size of 8KB, the tags represent a modest 2MB.

7.5 Communication

In terms of communications, the users send their original data once, and send replications parameters Π. The size of Π is of $2 \cdot \lambda^* \cdot (R+1) \cdot \|N\|$, that is $\lambda^* \cdot (R+1) \cdot 512$B. For a 11 replicas and $\lambda^* = 15$, the amount of information sent is 165KB. Notice that we assumed all replication parameters to be distributed in $\mathbb{Z}_N$; in fact, we know that instead α_1^* and β_1^* which have a long bitlength, all other $\alpha_2^* \cdots \alpha_{\lambda^*}^*$ have a bitlength of 1; however, we did not explore compression options here, and present the trivial way of exchanging Π.

Concerning the VERIFY protocol, the communications is of two seeds, plus $s+1$ values as an answer; in total, we estimate the communication for one challenge to be close to $(s+3) \cdot 256$B.

Chapter 8

Conclusion

In this work, we proposed four novel solutions that allow users to efficiently verify the retrievability and the replication of outsourced data. Unlike existing schemes, our solutions make the service provider replicate the data, which lessen the burden on the users, and allows the service provider to use a new business model with cheaper costs for replicated data while preventing potential abuse of the system by users.

We implemented those solutions and benchmarked them in realistic settings; we then discussed their strength and weaknesses. The selected solution, Mirror, incurs tolerable overhead for both the client and the server, and is secure in the chosen model.

Some open questions remains : the construction of a multi-replica *dynamic* scheme, i.e. that allows file to be updated, similar to the work of Barsoum et al., and Etemad et al. [8, 17], and/or the construction of a multi-replica scheme that allows the verifier to be *outsourced*, such as the work by Armknecht et al. [6].

Bibliography

[1] Amazon. Announcing amazon s3 reduced redundancy storage, 2010. URL: https://aws.amazon.com/about-aws/whats-new/2010/05/19/announcing-amazon-s3-reduced-redundancy-storage/.

[2] Amazon. Amazon s3 introduces cross-region replication, 2015. URL: http://aws.amazon.com/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/.

[3] Amazon. Amazon web services : Regions and endpoints, 2015. URL: http://docs.aws.amazon.com/general/latest/gr/rande.html.

[4] Amazon. Amazon s3 pricing, 2105. URL: http://aws.amazon.com/s3/pricing/.

[5] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O Karame. Nec technical report. 2015.

[6] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O Karame, Zongren Liu, and Christian A Reuter. Outsourced proofs of retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 831–843. ACM, 2014.

[7] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. Acm, 2007.

[8] A Barsoum and M Hasan. Provable multi-copy dynamic data possession in cloud computing systems. 2015.

[9] Ayad F Barsoum and M Anwar Hasan. Provable possession and replication of data over cloud servers. *Centre For Applied Cryptographic Research (CACR), University of Waterloo, Report*, 32:2010, 2010.

[10] Karyn Benson, Rafael Dowsley, and Hovav Shacham. Do you know where your cloud files are? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 73–82. ACM, 2011.

[11] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *Advances in Cryptology—ASIACRYPT 2001*, pages 514–532. Springer, 2001.

[12] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54. ACM, 2009.

[13] Kevin D Bowers, Marten van Dijk, Ari Juels, Alina Oprea, and Ronald L Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 501–514. ACM, 2011.

[14] Jin-Yi Cai, Richard J Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *Structure in Complexity Theory Conference, 1993., Proceedings of the Eighth Annual*, pages 2–11. IEEE, 1993.

[15] Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. Mr-pdp: Multiple-replica provable data possession. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pages 411–420. IEEE, 2008.

[16] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 285–298. ACM, 2013.

[17] Mohammad Etemad and Alptekin Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *Applied Cryptography and Network Security*, pages 1–18. Springer, 2013.

[18] Forbes. Roundup of cloud computing forecasts and market estimates, 2015, 2015. URL: `http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015/`.

[19] Forbes. Roundup of cloud computing forecasts and market estimates, 2015, 2015. URL: `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html`.

[20] Décio Luiz Gazzoni Filho and Paulo Sérgio Licciardi Messeder Barreto. Demonstrating data possession and uncheatable data transfer. *IACR Cryptology ePrint Archive*, 2006:150, 2006.

[21] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.

[22] Ghassan O Karame and Srdjan Čapkun. Low-cost client puzzles based on modular exponentiation. In *Computer Security–ESORICS 2010*, pages 679–697. Springer, 2010.

[23] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge university press, 1994.

[24] Andrew Mirsky. Liability for data loss in the cloud: Why no one accepts liability? why carve it out?, 2013. URL: `http://mirskylegal.com/2013/03/liability-for-data-loss-in-the-cloud-why-noone-accepts-liability-why-carve-it-out/`.

[25] The Linux Foundation NetEm. Netem, 2009. URL: `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[26] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[27] Scala. The scala programming language, 2015. URL: http://www.scala-lang.org/.

[28] Francesc Sebé, Josep Domingo-Ferrer, Antoni Martinez-Balleste, Yves Deswarte, and J-J Quisquater. Efficient remote data possession checking in critical information infrastructures. *Knowledge and Data Engineering, IEEE Transactions on*, 20(8):1034–1038, 2008.

[29] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Advances in Cryptology-ASIACRYPT 2008*, pages 90–107. Springer, 2008.

[30] Statista. Number of registered dropbox users from april 2011 to june 2015, 2015. URL: http://www.statista.com/statistics/261820/number-of-registered-dropbox-users/.

[31] Marten Van Dijk, Ari Juels, Alina Oprea, Ronald L Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 265–280. ACM, 2012.

[32] Gaven J Watson, Reihaneh Safavi-Naini, Mohsen Alimomeni, Michael E Locasto, and Shivaramakrishnan Narayan. Lost: location based storage. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 59–70. ACM, 2012.

[33] Ke Zeng. Publicly verifiable remote data integrity. In *Information and Communications Security*, pages 419–434. Springer, 2008.

Appendices

Table 1 Additive scheme

Client		Server
Let k be a keyed PRF, with k_{PRF} as user key.		
$p \leftarrow 2p' + 1$, p' prime		
$q \leftarrow 2q' + 1$, q' prime		
$N \leftarrow pq$		
Pre-share a collection of p_r and E_r		
Pick g a generator of QR_N		
STORE (File F, k_{PRF}):		
Pick $\alpha_1 \cdots \alpha_s$ from $\mathbb{Z}_{\phi(N)}$ using the PRF		
Pick $\text{secret}_1 \cdots \text{secret}_n$ from $\mathbb{Z}_N$ using the PRF		
For $i \in [1, n]$, $\sigma_i \leftarrow \text{secret}_i + \sum_{j=1}^{s} \alpha_j m_{i,j}$		
$\tau_U \leftarrow (k_{\text{PRF}}, N)$	$\xrightarrow{\{m_{i,i}\}, \{\sigma_i\}}$	Stores $\{m_{i,i}\}, \{\sigma_i\}$
Keep $\tau_U, \{\sigma_i\}$, may delete $\{m_{i,i}\}$		
		REPLICATE (r) :
		For all $i \in [1, n]$, $j \in [1, s]$:
		$m_{i,j}^{(r)} \leftarrow m_{i,j} + p_r^{E_r^{(j-1)s+i}}$
CHALLENGE:		
Choose l indices $i_1 \cdots i_l$ in $[1, n]$		
Pick $v_{i_1} \cdots v_{i_l}$ at random in $\mathbb{Z}_p$		
$Q = \{(i_k, v_{i_k})\} \; \forall \; k \in [1, l]$	$\xrightarrow{Q}$	$\sigma \leftarrow \sum_{(i,v_i) \in Q} \sigma_i \cdot v_i$
		$\forall j \in [1, s]$,
		$\mu_j \leftarrow \sum_{(i,v_i) \in Q} v_i \cdot m_{i,j}^{(r)}$
	$\xleftarrow{\sigma, \{\mu_j\}}$	
Computes $\alpha_1 \cdots \alpha_n, \text{secret}_1 \cdots \text{secret}_n$ from the PRF		
$\text{LHS} = \sum_{j=1}^{s} \alpha_j \cdot \mu_j^* + \sum_{(i,v_i) \in Q} v_j \cdot \text{secret}_i$		
$\text{RHS} = \sigma + \sum_{j=1}^{s} \alpha_j \cdot P_j$, where		
$P_j := \sum_{(i,v_i) \in Q} v_i \cdot p_r^{E_r^{(i-1) \cdot s + j \mod \phi(N)}} \mod (N)$		
Accepts if LHS = RHS		

Table 2 Multiplicative scheme

Client		Server
Let k be a keyed PRF, with k_{PRF} as user key.		
$p \leftarrow 2p' + 1$, p' prime		
$q \leftarrow 2q' + 1$, q' prime		
$N \leftarrow pq$		
Pre-share a collection of p_r and E_r		
Pick g a generator of QR_N		
STORE (File F, k_{PRF}):		
Pick $\alpha_1 \cdots \alpha_s$ from $\mathbb{Z}_{\phi(N)}$ using the PRF		
Pick $\text{secret}_1 \cdots \text{secret}_n$ from $\mathbb{Z}_N$ using the PRF		
For $i \in [1, n]$, set $\sigma_i \leftarrow \text{secret}_i \prod_{j=1}^{s} m_{i,j}^{\alpha_j}$		
$\tau_U \leftarrow (k_{\text{PRF}}, N)$	$\xrightarrow{\{m_{i,i}\}, \{\sigma_i\}}$	Stores $\{m_{i,i}\}, \{\sigma_i\}$
Keep $\tau_U, \{\sigma_i\}$, may delete $\{m_{i,i}\}$		
		REPLICATE (r) :
		For all $i \in [1, n]$, $j \in [1, s]$:
		$m_{i,j}^{(r)} \leftarrow m_{i,j} \cdot p_{j,r}^{E_r^{(j-1)s+i}}$
VERIFY:		
Picks $\text{seed}_i, \text{seed}_v$ at random in Z_n	$\xrightarrow{\text{seed}_i, \text{seed}_v}$	Using the PRF, computes :
$i_1 \cdots i_l$ using seed_i		$i_1 \cdots i_l$ using seed_i
$v_1 \cdots v_l$ using seed_v		$v_1 \cdots v_l$ using seed_v
Let $Q = \{(i_k, v_k)\}\ \forall k \in [1, l]$		Let $Q = \{(i_k, v_k)\}\ \forall k \in [1, l]$
		Computes :
		$\sigma \leftarrow \prod_{(i,v_i) \in Q} \sigma_i^{v_i}$
		$\forall j \in [1, s]$, computes
		$\mu_j \leftarrow \prod_{(i,v_i) \in Q} \left(m_{i,j}^{(r)}\right)^{v_i}$
	$\xleftarrow{\sigma, \{\mu_j\}}$	
Computes $\alpha_1 \cdots \alpha_n, \text{secret}_1 \cdots \text{secret}_n$ from the PRF		
$\text{LHS} \leftarrow \prod_{j=1}^{s} \mu_j^{\alpha_j} \prod_{(i,v_i) \in Q} \text{secret}_i^{v_j}$		
$\text{RHS} \leftarrow \sigma \prod_{j=1}^{s} P_j^{\alpha_j}$, where		
$P_j := \prod_{(i,v_i) \in Q} p_j^{v_k E^{(i-1) \cdot s + j} \mod \phi(N)} \mod (N)$		
Accepts if LHS = RHS		

Table 3 Scheme with precomputation

Client		Server
Let k be a keyed PRF, with k_{PRF} as user key.		
$p \leftarrow 2p' + 1$, p' prime		
$q \leftarrow 2q' + 1$, q' prime		
$N \leftarrow pq$		
Pick g a generator of QR_N		
Let $\Omega : (\{0,1\}^*, \{0,1\}^*) \rightarrow [0, \Lambda]^w$ a PRF		
λ, w, Ω are fixed and shared		
VERIFY (File F, k_{PRF}):		
Pick $\alpha_1 \cdots \alpha_s$ from $\mathbb{Z}_{\phi(N)}$ using the PRF		
Pick $\text{secret}_1 \cdots \text{secret}_n$ from $\mathbb{Z}_N$ using the PRF		
For $i \in [1, n]$, $\sigma_i \leftarrow \text{secret}_i \prod_{j=1}^{s} m_{i,j}^{\alpha_j}$		
$\tau_U \leftarrow (k_{\text{PRF}}, N)$	$\xrightarrow{\{m_{i,i}\}, \{\sigma_i\}}$	Stores $\{m_{i,i}\}, \{\sigma_i\}$
Keep $\tau_U, \{\sigma_i\}$, may delete $\{m_{i,i}\}$		
		REPLICATE (r) :
		Precomputation:
		Set $\Lambda \leftarrow ns\lambda$
		$i \in [0, \Lambda]$:
		$\text{Powers}_{p_r}(i) \leftarrow p_r^{E^i}$
		Actual Replication:
		For all $i \in [1, n]$, $j \in [1, s]$:
		$m_{i,j}^{(r)} \leftarrow m_{i,j} \cdot \prod_{(i_k \in \Omega(i,j))} \text{Powers}_{p_r}(i_k)$
VERIFY:		
Picks $\text{seed}_i, \text{seed}_v$ at random in Z_n	$\xrightarrow{\text{seed}_i, \text{seed}_v}$	Using the PRF, computes :
$i_1 \cdots i_l$ using seed_i		$i_1 \cdots i_l$ using seed_i
$v_1 \cdots v_l$ using seed_v		$v_1 \cdots v_l$ using seed_v
Let $Q = \{(i_k, v_k)\}$ $\forall k \in [1, l]$		Let $Q = \{(i_k, v_k)\}$ $\forall k \in [1, l]$
		Computes :
		$\sigma \leftarrow \prod_{(i,v_i) \in Q} \sigma_i^{v_i}$
		$\forall j \in [1, s]$, computes
		$\mu_j \leftarrow \prod_{(i,v_i) \in Q} \left(m_{i,j}^{(r)}\right)^{v_i}$
	$\xleftarrow{\sigma, \{\mu_j\}}$	
Computes $\alpha_1 \cdots \alpha_n, \text{secret}_1 \cdots \text{secret}_n$ from the PRF		
$\text{LHS} \leftarrow \prod_{j=1}^{s} {\mu_j}^{\alpha_j} \prod_{(i,v_i) \in Q} \text{secret}_i^{v_j}$		
$\text{RHS} \leftarrow \sigma \prod_{j=1}^{s} \prod_{(i,v_i) \in Q} {p_r}^{v_i \alpha_j \sum_{(i,v_i) \in Q} E^i \mod \phi(n)}$		
Accepts if LHS = RHS		

Table 4 LFSR-based scheme

Client		Server
Let k be a keyed PRF, with k_{PRF} as user key.		
$p \leftarrow 2p' + 1$, p' prime		
$q \leftarrow 2q' + 1$, q' prime		
$N \leftarrow pq$		
Pre-share the replication parameters Π		
STORE (File F, k_{PRF}):		
Pick $\epsilon_1 \cdots \epsilon_s$ from $\mathbb{Z}_{\phi(N)}$ using the PRF		
For $i \in [1, n]$, set $\sigma_i \leftarrow \prod_{j=1}^{s} m_{i,j}^{\epsilon_j}$		
$\tau_U \leftarrow (k_{\text{PRF}}, N)$	$\xrightarrow{\{m_{i,i}\}, \{\sigma_i\}}$	Stores $\{m_{i,i}\}, \{\sigma_i\}$
Keep $\tau_U, \{\sigma_i\}$, may delete $\{m_{i,i}\}$		
		REPLICATE (r):
		Forward direction:
		For all $i \in [1, n]$, $j \in [1, s]$:
		set $m_{i,j}^{(r)} \leftarrow m_{i,j} \cdot \widetilde{a}_{(i-1)\cdot s+j}$
		Backward direction:
		For all i from n to 1, j from s to 1 :
		set $m_{i,j}^{(r)} \leftarrow m_{i,j}^{(r)} \cdot \widetilde{b}_{n\cdot s-((i-1)\cdot s+j)}$
VERIFY:		
Picks $\text{seed}_i, \text{seed}_v$ at random in Z_n	$\xrightarrow{\text{seed}_i, \text{seed}_v}$	Using the PRF, computes :
$i_1 \cdots i_l$ using seed_i		$i_1 \cdots i_l$ using seed_i
$v_1 \cdots v_l$ using seed_v		$v_1 \cdots v_l$ using seed_v
Let $Q = \{(i_k, v_k)\}\ \forall k \in [1, l]$		Let $Q = \{(i_k, v_k)\}\ \forall k \in [1, l]$
		Computes :
		$\sigma \leftarrow \prod_{(i,v_i)\in Q} \left(\sigma_i \cdot \prod_{j=1}^{s} \prod_{r\in[1,R]} m_{i,j}^{(r)}\right)^{v_i}$
		$\forall j \in [1, s]$, computes
		$\mu_j \leftarrow \prod_{(i,v_i)\in Q} \left(m_{i,j}^{(r)}\right)^{v_i}$
	$\xleftarrow{\sigma, \{\mu_j\}}$	
Computes $\epsilon_1 \cdots \epsilon_n$ from the PRF		
LHS $\leftarrow \prod_{(i,v_i)\in Q} \left(\sigma_i \cdot \prod_{j=1}^{s} \prod_{r\in k} g^{a_{i,j}^{(r)}} \cdot h^{b_{i,j}^{(r)}}\right)^{-v_i}$		
RHS $\leftarrow \prod_{j=1} \mu_j^{\epsilon_j + \|k\|}$		
Accepts if LHS = RHS		