Mirror: Enabling Proofs of Data Replication and Retrievability in the Cloud

Frederik Armknecht, *University of Mannheim;* Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame, *NEC Laboratories Europe*

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

Mirror: Enabling Proofs of Data Replication and Retrievability in the Cloud

Frederik Armknecht
University of Mannheim, Germany
`armknecht@uni-mannheim.de`

Ludovic Barman
NEC Laboratories Europe, Germany
`ludovic.barman@neclab.eu`

Jens-Matthias Bohli
NEC Laboratories Europe, Germany
Hochschule Mannheim, Germany
`jens.bohli@neclab.eu`

Ghassan O. Karame
NEC Laboratories Europe, Germany
`ghassan.karame@neclab.eu`

Abstract

Proofs of Retrievability (POR) and Data Possession (PDP) are cryptographic protocols that enable a cloud provider to prove that data is correctly stored in the cloud. PDP have been recently extended to enable users to check in a single protocol that additional file replicas are stored as well. To conduct multi-replica PDP, users are however required to process, construct, and upload their data replicas by themselves. This incurs additional bandwidth overhead on both the service provider and the user and also poses new security risks for the provider. Namely, since uploaded files are typically encrypted, the provider cannot recognize if the uploaded content are indeed replicas. This limits the business models available to the provider, since e.g., reduced costs for storing replicas can be abused by users who upload different files—while claiming that they are replicas.

In this paper, we address this problem and propose a novel solution for proving data replication and retrievability in the cloud, Mirror, which allows to shift the burden of constructing replicas to the cloud provider itself—thus conforming with the current cloud model. We show that Mirror is secure against malicious users and a rational cloud provider. Finally, we implement a prototype based on Mirror, and evaluate its performance in a realistic cloud setting. Our evaluation results show that our proposal incurs tolerable overhead on the users and the cloud provider.

1 Introduction

The cloud promises a cost-effective alternative for small and medium enterprises to downscale/upscale their services without the need for huge upfront investments, e.g., to ensure high service availability.

Currently, most cloud storage services guarantee service and data availability [4, 6] in their Service Level Agreements (SLAs). Availability is typically ensured by means of full replication [4, 23]. Replicas are stored onto different servers—thus ensuring data availability in spite of server failure. Storage services such as Amazon S3 and Google FS provide such resiliency against a maximum of two concurrent failures [30]; here, users are typically charged according to the required redundancy level [4].

Nevertheless, none of today's cloud providers accept any liability for data loss in their SLAs. This makes users reluctant—and rightly so—when using cloud services due to concerns with respect to the integrity of their outsourced data [2, 7, 10]. These concerns have been recently fueled by a number of data loss incidents within large cloud service providers [5, 10]. For instance, Google recently admitted that a small fraction of their customers' data was permanently lost due to lightning strikes which caused temporary electricity outages [10].

To remedy this, the literature features a number of solutions that enable users to remotely verify the availability and integrity of stored data [11, 15, 16, 25, 34]. Examples include Proofs of Retrievability (POR) [25, 34] which provide clients with the assurance that their data is available in its entirety, and Proofs of Data Possession (PDP) [12] which enable a client to verify that its stored data has not undergone any modifications. PDP schemes have been recently extended to verify the replication of files [18, 22, 30]. These extensions can provide guarantees for the users that the storage provider is replicating their data as agreed in the SLA, and that they are indeed getting the value for their money.

Notice, however, that existing solutions require the users themselves to create replicas of their files, appropriately pre-process the replicas (i.e., to create authentication tags for PDP), and finally upload all processed replicas in the cloud. Clearly, this in-

Table 1: Bandwidth cost in different regions as provided by CloudFlare [3]. "% Peered" refers to the percentage of traffic exchanged for free with other providers.

Region	% Peered	Effective price/Mbps/Month
Europe	50%	$5
North America	20%	$8
Asia	55%	$32
Latin America	60%	$32
Australia	50%	$100

curs significant burden on the users. Moreover, this consumes considerable bandwidth from the provider, that might have to scale up its bandwidth to accommodate for such large upload requests. For example, in order to store a 10 GB file together with three replicas, users have to process and upload at least 40 GB of content. Recall that the provider's bandwidth is a scarce resource; most providers, such as AWS and CloudFlare, currently buy bandwidth from a number of so-called Tier 1 providers to ensure global connectivity to their datacenters [3]. For example, CloudFlare pays for maximum utilization (i.e., maximum number of Mbps) used per month. This process is costly (cf. Table 3) and is considerably more expensive than acquiring storage and computing resources [24].

Besides consuming the provider's bandwidth resources, this also limits the business models available to the provider, since e.g., reduced costs for storing replicas can be offered in the case where the replication process does not consume considerable bandwidth resources from the provider (e.g., when the replication is locally performed by the provider). Alternatively, providers can offer reduced costs by offering infrequent/limited access to stored replicas, etc. Amazon S3, for example, charges its users almost 25% of the underlying storage costs for additional replication [1,9]. Users therefore have considerable incentives to abuse this service, and to store their data at reduced costs as if they were replicas. Since the outsourced data is usually encrypted, the provider cannot recognize if the uploaded contents are indeed replicas.

In this paper, we address this problem, and propose a novel solution for proving data replication and retrievability in the cloud, Mirror, which goes beyond existing multi-replica PDP solutions and enables users to efficiently verify the retrievability of all their replicas without requiring them to replicate data by themselves. Notably, in Mirror, users need to process/upload their original files only once irrespective of the replication undergone by their data; here, conforming with the current cloud model [4],

the cloud provider appropriately constructs the replicas given the original user files. Nevertheless, Mirror allows users to efficiently verify the retrievability of all data replicas—including those constructed by the service provider.

To this end, Mirror leverages cryptographic puzzles to impose significant resource constraints—and thus an economic disincentive—on a cloud provider which creates the replicas on demand, i.e., whenever the client initiates the verification protocol. By doing so, Mirror incentivizes a rational cloud provider to correctly store and replicate the clients' data—otherwise the provider risks detection with significant probability.

In summary, we make the following contributions in this work:

- We propose a novel formal model and a security model for proofs of replication and retrievability. Our proposed model, PoR^2, extends the POR model outlined in [34] and addresses security risks that have not been covered so far in existing multi-replica PDP models.
- We describe a concrete PoR^2 scheme, dubbed Mirror that is secure in our enhanced security model. Mirror leverages a tunable replication scheme based on the combination of Linear Feedback Shift Registers (LFSRs) with the RSA-based puzzle by Rivest [33]. By doing so, Mirror shifts the burden of constructing replicas to the cloud provider itself and is therefore likely to be appreciated by cloud providers since it allows them to trade their expensive bandwidth resources with relatively cheaper computing resources.
- We implement and evaluate a prototype based on Mirror in a realistic cloud setting, and we show that our proposal incurs tolerable overhead on both the users and the cloud provider when compared to existing multi-replica PDP schemes.

The remainder of this paper is organized as follows. In Section 2, we introduce a novel model for proofs of replication and retrievability. In Section 3, we propose Mirror, an efficient instantiation of our proposed model, and analyze its security in Section 4. In Section 5, we evaluate a prototype implementation of Mirror in realistic cloud settings and compare its performance to the multi-replica PDP scheme of [18]. In Section 6, we overview related work in the area, and we conclude the paper in Section 7.

2 PoR^2: Proofs of Replication and Retrievability

In this section, we introduce a formal model for proofs of replication and retrievability, PoR^2.

2.1 System Model

We consider a setting where a user $\mathscr{U}$ outsources a file D to a service provider $\mathscr{S}$ who agrees to the following two conditions:

1. Store the file D in its entirety.
2. Additionally store r replicas of D in their entirety.

A PoR^2 protocol aims to ensure to the user that both conditions are kept without the need for users to download the files and the replicas. Hence, our model comprises a further party: the verifier $\mathscr{V}$ who runs the PoR^2 scheme to validate that indeed the data and sufficient copies are stored by $\mathscr{S}$. In a privately-verifiable scheme, the user and the verifier consist of the same entity; these roles might be however different in publicly-verifiable schemes.

As one can see, Condition 1 indirectly implies that a PoR^2 scheme needs to comprise a PDP or POR scheme. Consequently, similar to the POR model, a PoR^2 involves a process for outsourcing the original data, referred to as Store, and an interactive verification protocol Verify.

However, Condition 2 goes beyond common POR/PDP requirements. Hence, one needs an additional (possibly interactive) process denoted by Replicate for generating the replicas and a second process, dubbed CheckReplica, which checks the correctness of the replicas (in case the replicas were created by the user). Moreover, the interactive verification protocol Verify needs to be extended such that it verifies the storage of the original file *and* the copies computed by the service provider. In what follows, we give a formal specification of the procedures and protocols involved in a PoR^2 scheme. Our model adapts and extends the original POR model introduced in [25, 34]. In Section 2.2, we summarize the relation between PoR^2 and previous POR models.

The Store Procedure: This randomized procedure is executed by the user once at the start. Store takes as input the security parameter κ and the file $\widetilde{D}$ to be outsourced, and produces a file tag τ that is required to run the verification procedure. Depending on whether the scheme is private or public verifiable, the verification tag needs to be kept secret or can be made public. The output of Store comprises the file D that the service provider should store. D may

be different from $\widetilde{D}$, e.g., contain some additional information, but $\widetilde{D}$ should be efficiently recoverable from D. Finally, Store outputs public parameters Π which allow the generation of r replicas $D^{(i)}$ of the outsourced file. We assume that the number of copies is (implicitly) given in the copy parameters Π, possibly being specified in the SLA before. Summing up, the formal specification of Store is:

$$(D, \tau, \Pi) \leftarrow \mathsf{Store}(\kappa, \widetilde{D})$$

The Replicate Procedure: The Replicate procedure is a protocol executed between the verifier (who holds the verification tag τ) and the service provider to generate replicas of the original file. To this end, Replicate takes as inputs the copy parameters Π and the outsourced file D, and outputs the r copies $D^{(1)}, \ldots, D^{(r)}$. In addition, the provider gets a *copy tag* τ^* which allows him to validate the correctness of the copies. Formally, we have:

$$\begin{aligned}\mathsf{Replicate}: \quad & [\mathscr{V}: \tau, \Pi; \ \mathscr{S}: D, \Pi] \\ \rightarrow \quad & [\mathscr{V}: \tau; \ \mathscr{S}: D^{(1)}, \ldots, D^{(r)}, \tau^*]\end{aligned}$$

Recall that the verifier $\mathscr{V}$ refers to the party that holds the verification tag and may not necessarily be a third party. This captures *(i)* the case where the user creates the copies on his own at the beginning (as discussed in [18]), and *(ii)* the case where this replication process is completely outsourced to the service provider (or even to a third party).

Observe that the output for the verifier includes again the verification tag. This is the case since we want to capture situations where the verification tag can be changed, depending on the protocol flow of Replicate. To keep the description simple, we denote both values (the verification tag as output of the Store procedure and the potentially updated verification tag after running Replicate) by τ.

The CheckReplica Procedure: The purpose of the CheckReplica procedure, which is used by the service provider, is to validate that the replicas have been correctly *generated*, i.e., are indeed copies of the original file. Notice that CheckReplica is mandatory for a comprehensive model but is not necessary in the case where the service provider replicates the data itself (in this case the service provider can ensure that the replication process is done correctly).

CheckReplica is executed between the verifier and the service provider. The verifier $\mathscr{V}$ takes as input the copy parameters Π and verification tag τ, being his output of the Replicate procedure (see above), while the service provider $\mathscr{S}$ takes as input the uploaded file D, a possible replica D^* (together with a replica

index $i \in \{1, \ldots, r\}$), the copy parameters Π, and the copy tag τ^*. CheckReplica then outputs a binary decision expressing whether the service provider $\mathscr{S}$ believes that D^* is a correct i-th replica of D according to the Replicate procedure and the copy parameters Π.

$$\textsf{CheckReplica} \colon [\mathscr{V} : \tau, \Pi; \ \mathscr{S} : \tau^*, \Pi, D, D^*, i] \to [\mathscr{S} : \textsf{dec}]$$

The Verify Protocol: A verifier $\mathscr{V}$, i.e., the user if the scheme is privately verifiable and possibly a third party if the scheme is publicly verifiable, and the provider $\mathscr{S}$ execute an interactive protocol to convince the verifier that both the outsourced D and the r replicas $D^{(1)}, \ldots, D^{(r)}$ are correctly stored. The input of $\mathscr{V}$ is the tag τ given by Store and the copy parameters Π, while the input of the provider $\mathscr{S}$ is the file D outsourced by the user and the r replicas generated by the Replicate procedure. The output $\textsf{dec} \in \{\textsf{accept}, \textsf{reject}\}$ of the verifier expresses his decision, i.e., whether he accepts or rejects. It holds that:

$$\textsf{Verify} \colon \ [\mathscr{V} : \tau, \Pi; \ \mathscr{S} : D, D^{(1)}, \ldots, D^{(r)}] \longrightarrow [\mathscr{V} : \textsf{dec}]$$

Note that Verify and CheckReplica aim for completely different goals. The CheckReplica procedure allows the service provider $\mathscr{S}$ to check if the replicas have been *correctly generated* and hence protects against a malicious customer who misuses the replicas for storing additional data at lower costs. On the other side, the Verify procedure enables a client or verifier $\mathscr{V}$ to validate that the file and all copies are indeed *stored in their entirety* to provide security against a rational service provider. For instance, CheckReplica can be omitted if the replicas have been generated by the service provider directly while Verify would still be required.

2.2 Relation to Previous Models

Notice that the introduced PoR^2 model covers and extends both proofs of retrievability and proofs of multiple replicas. For example in case that no replicas are created at all, i.e., neither the Replicate nor the CheckReplica procedures are used, the scheme reduces to a standard POR according to the model given in [34]. Observe that in such cases storage allocation is a direct consequence of the incompressibility of the file. Moreover, the multi-replica schemes presented so far (see Section 6 for an overview) can be seen as PoR^2 schemes where the correct replication requirement is simply ignored. In fact, we argue that if existing multi-replica schemes are coupled with a proof that the replicas are honestly generated by the user, then the resulting scheme would be a secure PoR^2 scheme.

2.3 Attacker Model

Similar to existing work in the area [35, 36], we adapt the concept of the *rational attacker model*. Here, rational means that if the provider cannot save any costs by misbehaving, then he is likely to simply behave honestly. In our particular case, the advantage of the adversary clearly depends on the relation between storage costs and other resources (such as computation), and on the availability of these resources to the adversary. In the sequel, we capture such a rational adversary by restricting him to a bounded number of concurrent threads of execution. Given that the provisioning of computational resources would incur additional costs, our restriction is justified by the fact that a rational adversary would only invest in additional computing resources if such a strategy would result in lower total costs (including the underlying costs of storage).

Likewise, we assume that users are interested to misuse the replicas for storing more data than has been agreed upon. Recall that since replicas are typically charged less than original files [1, 9], a rational user may try to encode additional information or other files into the replicas.

2.4 Security Goals and Correctness

In this section, we formalize the security goals of a PoR^2 scheme and define the correctness requirements. Note that we do not consider confidentiality of the file $\widetilde{D}$, since we assume that the user encrypts the file prior to the start the PoR^2 protocol. We start by defining three security notions that a PoR^2 scheme must guarantee:

Extractability: The user can recover the uploaded file D.

Storage Allocation: Provider uses *at least* as much storage as required to store the file and all replicas.

Correct Replication: The files $D^{(i)}$ are *correct* replicas of D.

The extractability notion protects the user against a malicious service provider who does not store the whole file. Similarly, the storage allocation notion aims to protect a user against a service provider who does not commit enough storage to store all replicas. Clearly, the first two conditions together imply that a rational provider $\mathscr{S}$ indeed stores D and the replicas $D^{(1)}, \ldots, D^{(r)}$ and therefore fulfills his part of providing redundancy to protect the data. In contrast to the two previous notions, correct replication aims to

protect the service provider against a malicious user who tries to encode additional data in the replicas. This is an important property, which is not satisfied by existing multi-replica PDP models, but which should cater to any practical deployment of PoR^2. In Section 3, we propose an instantiation of PoR^2 which allows the provider to run Replicate by itself—thus inherently satisfying this property. In the following paragraphs, we provide a formal description of the above defined notions.

Extractability. Extractability guarantees that an honest user is able to recover the data $\widetilde{D}$. Adopting [25, 34], this is formalized as follows. If a service provider is able to convince a honest user with significant probability during the Verify procedure, then there exists an *extractor algorithm* that can interact with the service provider and extract the file. This is captured by a hypothetical game between an adversary and an environment where the latter simulates all honest users and an honest verifier. The adversary is allowed to request the environment to create new honest users (including respective public and private keys), to let them store chosen files, and to run the Verify and Replicate procedures. At the end, the adversary chooses a user $\mathscr{U}$ with the corresponding outsourced file D and outputs a service provider $\mathscr{S}$ who can execute the Verify protocol with $\mathscr{U}$ with respect to the chosen file D. We say that a service provider is ε-*admissible* if the probability that the verifier does not abort is at least ε.

Definition 1 (Extractability) *We say that a* PoR^2 *scheme is ε-extractable if there exists an extraction algorithm such that for any PPT algorithm who plays the aforementioned game and outputs an ε-admissible service provider $\mathscr{S}$, the extraction algorithm recovers D with overwhelming probability.*

In addition, we say that correctness is provided with respect to the extractability if the following holds. If all parties are honest, i.e., the user, the verifier, and the provider, then the verifier accepts the output of the Verify protocol with probability 1. This should hold for any file $\widetilde{D} \in \{0,1\}^*$.

Storage Allocation. Let ST denote the storage of the service provider that has been allocated for storing the file D and the replicas $D^{(1)}, \ldots, D^{(r)}$. We compute the storage allocation by the provider, ρ, as follows:

$$\rho := \frac{|\mathsf{ST}|}{|D| + |D^{(1)}| + \ldots + |D^{(r)}|} \tag{1}$$

Here, we consider the generic case where the sizes of the replicas can be different (e.g., due to different metadata). Moreover, we assume that neither the file nor the replicas can be (further) compressed, e.g., because these have been encrypted first. Since the service provider aims to save storage, it holds in general that $0 \leq \rho \leq 1$. Storage allocation ensures that $\rho \geq \delta$ for a threshold $0 \leq \delta \leq 1$ chosen by the user.

Definition 2 (Binding) *We say that a* PoR^2 *scheme is (δ,ε)-binding if for any rational attacker who plays the aforementioned game, and outputs an ε-admissible service provider $\mathscr{S}$ who invests only a fraction $\rho < \delta$ of memory, it holds that the verifier accepts only with negligible probability (in the security parameter).*

We say that the scheme is even strongly (δ,ε)-binding if it holds for any PPT attacker, i.e., also for non-rational attackers.

We stress that the distinction between binding and strongly binding is necessary in a comprehensive model. For instance, for schemes where the replicas are generated locally by the service provider $\mathscr{S}$ himself, i.e., to save bandwidth, the strongly binding property is impossible to achieve for $\delta > |\mathsf{ST}|/|D|$. The reason is that a non-rational service provider could always store D only and run the Replicate procedure over and over again when needed. On the other hand, if the user is generating and uploading the replicas, strong binding could be achieved when replicas are different encryptions of the original file, e.g., as done in [18]. In Fortress, we aim to outsource the replica generation to the service provider to save bandwidth and hence only aim for the binding property.

Correct Replication. Correct replication means essentially that both, Replicate and CheckReplica, are sound and correct. We detail this below.

We say that Replicate is sound if in the case where the user is involved in the replica generation, the service provider can get assurance that the additionally uploaded data represents indeed correctly built replicas that do not encode, for example, some additional data. That is, Replicate must not be able to encode a significant amount of additional data in the replicas. This is formally covered by the requirement that inputs of the verifier to the replicate procedure Replicate, namely the verification tag τ and the copy parameters Π, have a size that is independent of the file size.

On the other hand, we say that Replicate is correct if replicas represent indeed copies of the uploaded file D. This is formally captured by requiring that

D can be efficiently recovered from any replica $D^{(k)}$. More precisely, we say that Replicate is correct if there exists an efficient algorithm which given τ, Π, and any replica $D^{(k)}$ outputs D.

With respect to CheckReplica, we require that $\mathcal{S}$ only accepts replicas which are valid output of Replicate. Let D and Π be the output of the Store procedure. Let $\mathcal{E}$ be the event that τ^* and $D^{(1)}, \ldots, D^{(r)}$ are the output of a Replicate run. Let dec be the decision of the service provider at the end of the CheckReplica protocol. We say that the scheme is ε^*-correctly building replicas if:

$$\forall i \in \{1, \ldots, r\} : \; Pr[\mathsf{dec} = \mathsf{Accept}|\mathcal{E}] \;=\; 1,$$
$$\max_{i \in \{1, \ldots, r\}} \{Pr[\mathsf{dec} = \mathsf{Accept}|\neg\mathcal{E}]\} \;\leq\; \varepsilon^*.$$

Observe that the first and second condition express the correctness and soundness of CheckReplica, respectively.

3 Mirror: An Efficient PoR^2 Instantiation

3.1 Overview

The goal of Mirror is to provide a verifiable replication mechanism for storage providers. Note that straightforward approaches to construct PoR^2 would either be communication-expensive or would be insecure in the presence of a rational cloud provider.

For instance, the user could create and upload the required t replicas of his files, similar to [18]. Obviously, this alternative incurs considerable bandwidth overhead on the providers and users can abuse the replicas to outsource several, different files in encrypted form. An alternative solution would be to enable the cloud provider to create the replicas (and their tags) on his own given the original files. This would significantly reduce the provider's bandwidth consumption incurred in existing multi-replica schemes at the expense of investing additional (cheaper) computing resources [24]. This alternative might be, however, insecure since it gives considerable advantage for the provider to misbehave, e.g., store only one single replica and construct the replicas on the fly when needed.

To thwart the generic attacks described above, Mirror ensures that a malicious cloud provider can only reply correctly within the verification protocol by investing a minimum amount of resources, i.e., memory and/or time. However, to ensure the binding property (Definition 2), i.e., that the provider invests memory and not time, Mirror allows to scale the computational effort that a dishonest provider would have to invest without increasing the memory effort of an honest provider. This allows to adjust the computational effort of a dishonest provider such that the costs of storing the replicas is cheaper than the costs of computing the response to the challenges on the fly—giving an economic incentive to a rational provider to behave honestly.

This is achieved in Mirror through the use of a tunable puzzle-based replication scheme. Namely, in Mirror, the user has to outsource only his original files and compact puzzles to the cloud provider; the solution of these puzzles will be then combined with the original file in order to construct the r required replicas. Puzzles are constructed such that *(i)* they require noticeable time to be solved by the cloud provider while the user is significantly more efficient by exploiting a trapdoor, *(ii)* storing their solution incurs storage costs that are at least as large as the required storage for replicas, *(iii)* their difficulty can be easily adjusted by the creator to cater for variable strengths (and different cost metrics), and *(iv)* they can be efficiently combined with the original file blocks in order to create r correct replicas of the file preserving the homomorphic properties needed for compact proofs[1].

To this end, Mirror combines the use of the RSA puzzle of Rivest [33] and Linear Feedback Shift Registers (LFSR) (cf. Section 3.3). A crucial aspect here is that the user creates two LFSRs: a short one which is kept secret, and a longer public LFSR. The service provider is only given the public LFSR to generate the exponent values. As we show later, this allows for high degrees of freedom with respect to security and performance of Mirror. In the following, we first explain the deployed main building blocks and give afterwards the full protocol specification.

3.2 Building Blocks

RSA-based Puzzles: Mirror ties each sector with a cryptographic puzzle that is inspired by the RSA puzzle of Rivest [33]. In a nutshell, the puzzle requires the repeated exponentiation of given values $X^a \mod N$ where $N = p \cdot q$ is publicly known RSA modulus and a, p, q remain secret. Without knowing these secrets, this requires to perform modular exponentiation. Modular exponentiation is an inherently sequential process [33]. The running time of the fastest known algorithm for modular exponentiation is linear in the size of the exponent. Although the provider might try to parallelize the computation of

---

[1]This condition restricts our choice of puzzles since e.g., hash-based puzzles cannot be efficiently combined with the authentication tag of each data block/sector.

the puzzle, the parallelization advantage is expected to be negligible [17, 26, 28, 33]. On the other hand, the computation can be efficiently verified by the puzzle generator through the trapdoor offered by Euler's function in $O(\log(N))$ modular multiplications by computing $X^{a'} \bmod N \equiv X^{a' \bmod \phi(N)} \bmod N$.

Observe that this puzzle is likewise multiplicative homomorphic: given a and a', the product of the solutions $X^{a'}$ and $X^{a''}$ represents a solution for $a' + a''$. This preserves the homomorphic properties of the underlying POR and allows for batch verification for all the replicas and hence enables compact proofs.

To further reduce the verification burden on users, Mirror generates the exponents using a Linear Feedback Shift Registers (LFSR) as follows.

Linear Feedback Shift Registers: A *Linear Feedback Shift Register (LFSR)* is a popular building block for designing stream ciphers as it enables the generation of long output streams based on a initial state. In Mirror, LFSRs will be used to generate the exponents for the RSA-based puzzle described above. In what follows, we briefly describe the concept of an LFSR sequence and refer the readers to [29] for further details.

Definition 3 (Linear Feedback Shift Register)
Let $\mathbb{F}$ be some finite field, e.g., $\mathbb{Z}_p$ for some prime p. A Linear Feedback Shift Register (LFSR) of length λ consists of an internal state of length λ and a linear feedback function $F : \mathbb{F}^{\lambda} \to \mathbb{F}$ with $F(x_1, \ldots, x_{\lambda}) = \sum_{i=1}^{\lambda} c_i \cdot x_i$. Given an initial state $(s_1, \ldots, s_{\lambda}) \in \mathbb{F}^{\lambda}$, it defines inductively an LFSR sequence $(s_t)_{t \geq 1}$ by $s_{t+\lambda} = F(s_t, \ldots, s_{t+\lambda-1})$ for $t \geq 1$.

An important and related notion is that of a *feedback polynomial*. Given an LFSR with feedback function $F(x_1, \ldots, x_{\lambda}) = \sum_{i=1}^{\lambda} c_i \cdot x_i$, the feedback polynomial $f(x) \in \mathbb{F}[x]$ is defined as:

$$f(x) = x^{\lambda} - \sum_{i=1}^{\lambda} c_i \cdot x^{i-1}. \qquad (2)$$

It holds that any multiple of a feedback polynomial is again a feedback polynomial. That is, if $f^*(x) = x^{\lambda^*} - \sum_{i=1}^{\lambda^*} c_i^* \cdot x^{i-1}$ is a multiple of f, then it holds that $s_{t+\lambda^*} - \sum_{i=1}^{\lambda^*} c_i^* \cdot s_{t+i-1} = 0$ for each $t \geq 1$. Mirror exploits this feature in order to realize a gap between the puzzle solution created by provider and the verification done by the user.

3.3 Protocol Specification

We now start by detailing the procedures in Mirror.

Specification of the Store Procedure: In the store phase, the user is interested in uploading a file $D \in \{0,1\}^*$. We assume that the file D is encrypted to protect its confidentiality and encoded with an erasure code (as required by the utilized POR in order to provide extractability guarantees) prior to being input to the Store protocol [25, 34]. First, the user generates an RSA modulus $N := p \cdot q$ where p and q are two safe primes[2] whose size is chosen according to the security parameter κ.

Similar to [34], the file is interpreted as n blocks, each is s sectors long. A sector is an element of $\mathbb{Z}_N$ and is denoted by $d_{i,j}$ with $1 \leq i \leq n$, $1 \leq j \leq s$. That is, the overall number of sectors in the file is $n \cdot s$. To ensure unique extractability (see Section 4.1), we require that the bit representation of each sector $d_{i,j}$ contains a characteristic pattern, e.g., a sequence of zero bits. The length of this pattern depends on the file size and should be larger than $\log_2(n \cdot s)$.

Furthermore, the user samples a key k_{prf} per file, where the key length is determined by the security parameter, e.g., $k_{\mathrm{prf}} \in \{0,1\}^{\kappa}$. By invoking k_{prf} as a seed to a pseudo-random function (PRF), the user samples s non-zero elements of $\mathbb{Z}_{\phi(N)}$, i.e., $\varepsilon_1, \ldots, \varepsilon_s \xleftarrow{\mathrm{R}} \mathbb{Z}_{\phi(N)} \setminus \{0\}$. Finally, the user computes σ_i for each i, $1 \leq i \leq n$, as follows:

$$\sigma_i \leftarrow \prod_{j=1}^{s} d_{i,j}^{\varepsilon_j} \quad \in \mathbb{Z}_N. \qquad (3)$$

These values are appended to the original file so that the user uploads $(D, \{\sigma_i\}_{1 \leq i \leq n})$. Unless specified otherwise, we note that all operations are performed in the multiplicative group $\mathbb{Z}_N^*$ of invertible integers modulo N.[3]

Assuming that the user is interested in maintaining r replicas in addition to the original file D at the cloud, the user additionally constructs copy parameters Π which will also be sent to the server. To this end, the user first generates two elements $g, h \in \mathbb{Z}_N^*$ of order p' and q', respectively. Recall that the order of $\mathbb{Z}_N^*$ is $\varphi(N) = (p-1)(q-1) = 4 \cdot p' \cdot q'$. The elements g and h will be made public to the server while their orders are kept secret.

Then, the user proceeds to specify feedback polynomials for two LFSRs, one being defined over $\mathbb{Z}_{p'}$ and the other over $\mathbb{Z}_{q'}$. Both LFSRs need to have a length λ such that $|\mathbb{F}|^{\lambda} > n \cdot s$. Here, for each of the two LFSRs, *two* feedback polynomials are specified: a shorter one which will be kept secret by the user

---

[2]That is, $p - 1 = 2 \cdot p'$ and likewise $q - 1 = 2 \cdot q'$ for two distinct primes p' and q'.

[3]Observe that hitting by coincidence a value outside of $\mathbb{Z}_N^*$ allows to factor N which is considered to be a hard problem.

and a larger one that will be made public to the provider. More precisely, for the LFSR defined over $\mathbb{Z}_{p'}$ the user chooses two polynomials

$$f_a(x) := x^{\lambda} - \sum_{i=1}^{\lambda} \alpha_i \cdot x^{i-1}, \quad f_a^*(x) := x^{\lambda^*} - \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot x^{i-1}$$

such that $f_a^*(x)$ is a multiple of $f_a(x)$ (and hence $\lambda < \lambda^*$). For security reasons, it is necessary to ensure that $\alpha_1^* \geq 2$.

The feedback polynomial $f_a(x)$ with the lower degree will be kept secret while the polynomial $f_a^*(x)$ of the higher degree and the larger coefficients will be given to the provider. $f_a(x)$ will serve as a feedback polynomial to generate for each replica $k \in \{1, \ldots, r\}$ an LFSR sequence $(a_t^{(k)})$. To this end, the user chooses for each k an initial state $(a_1^{(k)}, \ldots, a_{\lambda}^{(k)}) \in \mathbb{Z}_{p'}^{\lambda}$ which defines the full sequence by $a_{t+\lambda+1}^{(k)} = \sum_{i=1}^{\lambda} \alpha_i \cdot a_{t+i}^{(k)}$ for any $t \geq 0$. Observe that due to the fact that $f_a^*(x)$ is a multiple of $f_a(x)$, it likewise holds $a_{t+\lambda^*+1}^{(k)} = \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot a_{t+i}^{(k)}$ for any $t \geq 0$. Finally, the user publishes as part of the copy parameters the values $g^{a_1^{(k)}}, \ldots, g^{a_{\lambda^*}^{(k)}} \in \mathbb{Z}_N$ for each replica and the coefficients $\alpha_1^*, \ldots, \alpha_{\lambda^*}^* \in \mathbb{Z}$. Afterwards, he proceeds analogously over $\mathbb{Z}_{q'}$, i.e., sample coefficients $\beta_i \in \mathbb{Z}_{q'}$, compute feedback functions $f_b(x), f_b^*(x)$, choose an initial state $(b_1^{(k)}, \ldots, b_{\lambda}^{(k)})$ for each replica, and so on.

Summing up, and assuming that the server should construct r replicas, the user sets the file specific verification tag (which are kept secret by the user) to:

$$\begin{aligned} \tau \quad := \quad & \Big(k_{\text{prf}}, p, q, g, h, (a_1^{(k)}, \ldots, a_{\lambda}^{(k)})_{1 \leq k \leq r}, (\alpha_1, \ldots, \alpha_{\lambda}), \\ & (b_1^{(k)}, \ldots, b_{\lambda}^{(k)})_{1 \leq k \leq r}, (\beta_1, \ldots, \beta_{\lambda})\Big). \end{aligned}$$

To enable the server to construct the r replicas, the following copy parameters are given to the server:

$$\begin{aligned} \Pi \quad := \quad & \Big((g^{a_1^{(k)}}, \ldots, g^{a_{\lambda^*}^{(k)}})_{1 \leq k \leq r}, (\alpha_1^*, \ldots, \alpha_{\lambda^*}^*), \\ & (h^{b_1^{(k)}}, \ldots, h^{b_{\lambda^*}^{(k)}})_{1 \leq k \leq r}, (\beta_1^*, \ldots, \beta_{\lambda^*}^*)\Big). \end{aligned}$$

That is, the user sends D, the values $\{\sigma_i\}_{1 \leq i \leq n}$, and Π to the service provider and keeps the verification tag τ secret. Observe that the size of τ and Π are independent of the file size.

Specification of the CheckReplica Procedure: As the replicas are completely generated by the service provider, a CheckReplica procedure is not required in Mirror. However, one could check the validity of the

data replicas by running the Replicate procedure and simply comparing the outputs.

Specification of the Replicate Procedure: Upon reception of D, the values $\{\sigma_i\}_{1 \leq i \leq n}$, and Π, the service provider $\mathscr{S}$ stores D and starts the construction of the r additional replicas $D^{(k)}$ for $1 \leq k \leq r$, of D. Here, each sector $d_{i,j}^{(k)}$ of replica k has the following form:

$$d_{i,j}^{(k)} = d_{i,j} \cdot g_{i,j}^{(k)} \cdot h_{i,j}^{(k)}, \tag{4}$$

We call these values $g_{i,j}^{(k)}$ and $h_{i,j}^{(k)}$ *blinding factors*. Both sets of blinding factors are computed by raising g and h by elements of the LFSR sequences $a_t^{(k)}$ and $b_t^{(k)}$, respectively, but one in the forward and the other in the backward order, namely:

$$g_{i,j}^{(k)} := g^{a_{(i-1) \cdot s + j}^{(k)}}, \quad h_{i,j}^{(k)} := h^{b_{(n \cdot s + 1) - (i-1) \cdot s - j}^{(k)}}. \tag{5}$$

To enable the provider to compute the blinding factors $g^{a_i^{(k)}}$, we make use of the fact that for any $t \geq 0$ it holds $a_{t+\lambda^*+1}^{(k)} = \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot a_{t+i}^{(k)}$ and hence

$$g^{a_{t+\lambda^*+1}^{(k)}} = \prod_{i=1}^{\lambda^*} \left(g^{a_{t+i}^{(k)}}\right)^{\alpha_i^*}. \tag{6}$$

The computation of the blinding factors $h^{b_i^{(k)}}$ works analogously.

In summary, the server constructs replicas $D^{(k)}$ for $k = 1, \ldots, r$ as follows:

$$\begin{pmatrix} d_{1,1} \cdot g^{a_1^{(k)}} \cdot h^{b_{(n-1) \cdot s + s}^{(k)}} & \cdots & d_{1,s} \cdot g^{a_s^{(k)}} \cdot h^{b_{(n-1) \cdot s + 1}^{(k)}} \\ d_{2,1} \cdot g^{a_{s+1}^{(k)}} \cdot h^{b_{(n-2) \cdot s + s}^{(k)}} & \cdots & d_{2,s} \cdot g^{a_{2 \cdot s}^{(k)}} \cdot h^{b_{(n-2) \cdot s + 1}^{(k)}} \\ \vdots & \ddots & \vdots \\ d_{n,1} \cdot g^{a_{(s-1) \cdot s + 1}^{(k)}} \cdot h^{b_s^{(k)}} & \cdots & d_{n,s} \cdot g^{a_{n \cdot s}^{(k)}} \cdot h^{b_1^{(k)}} \end{pmatrix}$$

Specification of the Verify Procedure: The Verify protocol generates at first a random challenge C. It contains a random ℓ-element set of tuples (i, v_i) where $i \in \{1, \ldots, n\}$ denotes a block index, and $v_i \xleftarrow{\text{R}} \mathbb{Z}_N$ is a randomly generated integer. In addition, a nonzero subset $R \subset \{1, \ldots, r\}$ is sampled. The set R will indicate which replicas will be involved in the challenge. Observe that $R = \emptyset$ would mean that simply a proof of retrievability is executed without checking the replicas. The challenge is then the combination of both:

$$C = ((i_c, v_c)_{c=1}^{\ell}, R). \tag{7}$$

Given a challenge C, the server computes the response $\mu = (\mu_1, \ldots, \mu_s) \in \mathbb{Z}_N^s$ as follows:

$$\mu_j := \prod_{c=1}^{\ell} d_{i_c, j}^{v_c}, \quad j = 1, \ldots, s. \tag{8}$$

Observe that μ_j is the product of powers of the original data, that is $d_{i_c,j}{}^{v_c}$. In addition, the file tags are processed in the same manner to obtain:

$$\sigma = \prod_{c=1}^{\ell} \left(\sigma_{i_c} \cdot \prod_{j=1}^{s} \prod_{k \in R} d_{i_c,j}{}^{(k)} \right)^{v_c}. \tag{9}$$

The tuple (μ, σ) marks the response and is sent back to the user who verifies (μ, σ) similar to the private-verifiable POR of [34]. First, he computes:

$$\tilde{\sigma} := \sigma \cdot \prod_{c=1}^{\ell} \left(\prod_{j=1}^{s} \prod_{k \in R} g_{i_c,j}^{(k)} h_{i_c,j}^{(k)} \right)^{-v_c} \tag{10}$$

Observe that this will require the reconstruction of the blinding factors. In Appendix C, we show how to efficiently perform this verification by the user, using the knowledge of the verification tags—in particular the knowledge of the secret shorter LFSR, its initial state, and the order of g and h, respectively.

Next, the user recovers the secret parameters ε_{i_k}, $k = 1, \ldots, \ell$, using the key k_{prf} as a seed for the PRF. Finally, the user verifies that the following holds:

$$\prod_{j=1}^{s} \mu_j{}^{\varepsilon_j + |R|} = \tilde{\sigma}. \tag{11}$$

We now explain why the verification step has to hold if the response has been computed correctly. First, it follows from Equation (4) that $\prod_{c=1}^{\ell} \left(\prod_{j=1}^{s} \prod_{r \in R} d_{i_c,j}^{(r)} \right)^{v_c}$ can be rewritten as the product of $\prod_{c=1}^{\ell} \left(\prod_{j=1}^{s} \prod_{r \in R} d_{i_c,j} \right)^{v_c}$ and $\prod_{c=1}^{\ell} \left(\prod_{j=1}^{s} \prod_{r \in R} g_{i_c,j}^{(r)} h_{i_c,j}^{(r)} \right)^{v_c}$. The second factor is exactly the part that is canceled out in Equation (10) while the first factor can be simplified to $\prod_{c=1}^{\ell} \left(\prod_{j=1}^{s} d_{i_c,j}^{|R|} \right)^{v_c}$.

Given a series of straightforward calculations, one can show that $\tilde{\sigma}$ can be rewritten to $\prod_{j=1}^{s} (\mu_j)^{\varepsilon_j + |R|}$. This proves the correctness of Equation (11)—hence the correctness with respect to extractability.

Moreover, it is easy to see that, since each sector of a replica corresponds to the multiplication of the corresponding sector of the uploaded file D with a blinding factor (that can be reconstructed from Π), the replicas are indeed copies of the original file. This means that Replicate is correct. Summing up, all three correctness requirements explained in Section 2 are fulfilled in Mirror.

4 Security Analysis

We now proceed to prove the security of our scheme according to the definitions in Section 2.4. Recall that the user is not involved in the replica generation and that the size of the parameters involved in creating a replica is independent of the file size. This ensures the correct replication property described in Section 2.4.

It remains to prove that *(i)* if the service provider $\mathscr{S}$ stores at least a fraction δ of all sectors in one replica, then the file can be reconstructed (extractability) and *(ii) if the service provider stores less than a fraction of δ of any replica*, this misbehavior will be detected with overwhelming probability (storage allocation).

4.1 Extractability

In principle, the computations done in the Store and Verify procedures of Mirror can be seen as multiplicative variants of the corresponding mechanisms of the privately-verifiable POR of [34] (see Appendix B for details on the scheme of [34]). In particular, the extractability arguments given in [34] transport directly to Mirror. We assume that an erasure coding is applied to the file to ensure the recovery of file contents from any fraction δ of the file. In particular, we refer to [34] for additional details on the choice of parameters (e.g., for erasure coding) such that retrievability is ensured if a fraction δ of the file is stored.

To show that Mirror enables the reconstruction of the file from sufficiently many correct responses, we point out that given a correct response, the user learns expressions of the form $\mu_j = \prod_{c=1}^{\ell} d_{i_c,j}{}^{v_c}$ for known exponents $v_c \in \mathbb{Z}$ and known indices. Let us assume some arbitrary ordering $\mu^{(1)}, \mu^{(2)}, \ldots$ on these expressions. If sufficiently many responses $\mu^{(i)}$ are known, the user can choose for any (i, j) coefficients $c^{(k)} \in \mathbb{Z}$ such that $\prod_k \left(\mu^{(k)} \right)^{c^{(k)}} = d_{i,j}^u =: \tilde{d}$ for a known value $u \in \mathbb{Z}$.

Recall that the order of any $d_{i,j} \in \mathbb{Z}_N^*$ is a divisor of $2p'q'$. If u is odd, u is co-prime to $p'q'$ with overwhelming probability and the user can simply compute $u^{-1} \bmod p'q'$ and determine $d_{i,j} = \tilde{d}^{u^{-1}}$. On the other hand, if u is even (i.e., $u = 2 \cdot u'$), two cases emerge. If the order of $d_{i,j}$ is a divisor of $p'q'$, the exponent is again co-prime to the order with high probability. In this case, the user computes $u^{-1} \bmod p'q'$ and checks if $\tilde{d}^{u^{-1}}$ contains the characteristic bit pattern (see description of the Store procedure). If this fails, this means that the order of $\tilde{d}$ is even and the user proceeds as follows. Observe that the order of $d_{i,j}^2$ is a divisor of $p'q'$. Thus, the user first computes $(u')^{-1} \bmod p'q'$ and then $\tilde{d}^{(u')^{-1}}$. This yields $d_{i,j}^2$. As the user knows the factorization of N, he can compute all four possible roots of $d_{i,j}^2$

(e.g., using the Chinese Remainder Theorem). Due to the characteristic pattern embedded in $d_{i,j}$ (see the specification of the Store procedure), the user is able to identify the correct $d_{i,j}$.[4]

4.2 Storage Allocation

Observe that Mirror represents in fact a proof of retrievability over the uploaded file *and* all replicas. This means that if a challenge involves sectors that are not stored, Mirror ensures that the service provider fails with high probability unless he is able to correctly reconstruct the missing replicas. In the following, we therefore investigate the effort of a malicious service provider in reconstructing missing sectors. That is, we consider the scenario where the service provider has stored the complete file[5] but only parts of some replicas.

In Mirror, the service provider needs to compute the corresponding blinding factors in order to recompute any missing sectors. As these are products of the form $g_i^{(k)} \cdot h_j^{(k)}$ and since these sequences are independent from each other, the service provider is forced to store values of the sequences $(g_i^{(k)})_i$ and $(h_j^{(k)})_j$ separately. Moreover, since these values are different for the individual replicas, knowing (or reconstructing) a value from one sequence and one replica does not help the service provider in deriving values of other sequences and/or replicas.

A crucial aspect of Mirror is that a cheating service provider should require a significantly higher effort compared to an honest service provider in recomputing missing replicas. Recall that both the user and the provider determine the blinding factors by computing LFSR sequences. One difference though is that the provider has to do his computations on values $g^{a_i^{(k)}}, h^{b_j^{(k)}} \in \mathbb{Z}_N^*$ while the user can efficiently compute on the exponents in $a_i^{(k)} \in \mathbb{Z}_{p'}$ and $b_j^{(k)} \in \mathbb{Z}_{q'}$ directly. Observe that the provider is not able to transfer his computations into $\mathbb{Z}_{p'}$ and $\mathbb{Z}_{q'}$ without eventually factoring $N = p \cdot q$, which is commonly assumed to be a hard problem. A further gap is that the user deploys LFSRs with feedback functions $f_a(x) \in \mathbb{Z}_{p'}[x]$ and $f_b(x) \in \mathbb{Z}_{q'}$ where the provider only knows the feedback functions $f_a^*(x), f_b^*(x) \in \mathbb{Z}[x]$ that are multiples of $f_a(x), f_b(x)$. Given that these functions involve more inputs, and require the construc-

tion of larger coefficients, this would incur additional (significant) computational overhead on the provider compared to the user. For deducing the shorter feedback functions $f_a(x), f_b(x)$, the provider would have to determine $\mathbb{Z}_{p'}$ respectively $\mathbb{Z}_{q'}$—which equally require the knowledge of the factors of N.

It remains to investigate the effort for reconstructing values of the LFSR sequences. Note that the sequences used in the replicas are defined by different independent internal states and that, for each replica, the sequences $(a^{(k)}_i)$ and $(b^{(k)}_j)$ are independent. We can therefore without loss of generality restrict our analysis to one sequence $(g_t)_{t \geq 1}$. For the ease of representation, we omit in the sequel the index (k) and write $g_i = g^{a_i}$ for the ease of representation. We say that $\vec{v} = (v_1, \dots, v_{n \cdot s}) \in \mathbb{Z}^{n \cdot s}$ represents a *valid relation* with respect to $(g_t)_{t \geq 1}$ if:

$$\prod_{i=1}^{n \cdot s} g_i^{v_i} = 1. \tag{12}$$

It follows from known facts about LFSRs that valid relations are the only means for the service provider to compute missing values g_j from known values g_i (see Appendix D for more details).

As the provider is forced to use the feedback function defined by the coefficients $(\alpha_1^*, \dots, \alpha_{\lambda^*}^*)$ (see above), the only valid relations the provider can derive are linear combinations of:

$$\overrightarrow{b_j} := (\underbrace{0 \dots 0}_{j-1}, \alpha_1^*, \dots, \alpha_{\lambda^*}^*, \underbrace{0 \dots 0}_{n \cdot s - (j-1) - \lambda^*}), \tag{13}$$

where $\overrightarrow{b_1}$ corresponds to the given feedback polynomial $f_a^*(x)$ and the others are derived by simple shift of indexes.

Hence, for any valid relation $\vec{v} = (v_i)_i \in \mathbb{V}$, there exist unique coefficients $c_1, \dots, c_{n \cdot s - \lambda + 1} \in \mathbb{Z}$ such that $\vec{v} = \sum_i c_i \cdot \overrightarrow{b_i}$. Let $i_{\min}$ be the smallest index with $c_{i_{\min}} \neq 0$. Then, it holds that the first $v_j = 0$ for $j < i_{\min} - 1$ and that $v_{i_{\min}} = c_{i_{\min}} \cdot \alpha_1^*$ (as the other vectors with index $i > i_{\min}$) are zero at index $i_{\min}$. Hence, it holds that $\max_i\{\lceil \log_2(v_i) \rceil\} \geq \lceil \log_2(\alpha_1^*) \rceil$.

This shows that the effort of *executing* a valid relation (cf. Equation (12)) involves at least one exponentiation with an exponent of size $\lceil \log_2(\alpha_1^*) \rceil$.[6] The effort to compute one exponentiation for an exponent of bitsize k is $3/2 \cdot k \cdot T_{\mathrm{mult}}$, where T_{mult} stands for the time it takes the resource-constrained rational attacker (see Section 2.3) to multiply two values modulo N [27].

---

[4]Observe that in principle it may happen with some probability that more than one root exhibits this pattern. This is the reason why a padding length $\geq \log_2(n \cdot s)$ is proposed such that the expected number of incorrectly reconstructed sectors in the file is less than 1.

[5]Recall that this property is validated by the proof of retrievability already.

[6]One may combine several exponentiations to reduce the overall number of exponentiations but one cannot reduce the effort to compute at least once the exponentiation with the highest exponent.

Thus, a pessimistic lower bound for reconstructing a missing value g_i is $3/2 \cdot \lceil \log_2(\alpha_1^*) \rceil \cdot T_{\text{mult}}$. Observe that we ignore here additional efforts such as finding appropriate valid relations (cf. Equation (12)), etc.

Assume now that the service provider stores less than a fraction δ of all sectors of a given replica where δ refers to the threshold chosen by the user (see also Definition 2). Thus, for any value g_i contained in the challenge, the probability that g_i has to be re-computed is at least $1 - \delta$. Due to the fact that this holds for the values h_j as well and that a challenge requests $\ell \cdot s$ sectors, the expected number of values that need to be recomputed is $2\ell \cdot s \cdot (1 - \delta)$. To achieve the binding property with respect to a rational attacker, one has to ensure that the time effort for recomputing these values incurs costs that exceed the costs for storing these values. This implies a time threshold T_{thr} which marks the minimum computational effort this should take. Given such a threshold T_{thr}, we get the following inequality:

$$\lceil \log_2(\alpha_1^*) \rceil \geq \frac{T_{\text{thr}}}{3\ell \cdot s \cdot (1 - \delta) \cdot T_{\text{mult}}}. \qquad (14)$$

That is, if the parameters are chosen as displayed in (14), a dishonest provider would bear on average higher costs than an honest provider. Here, one can use the common cut-and-choose approach, by posing a number of challenges where the number is linear in the security parameter κ, to ensure that the overall probability to circumvent these computations is negligible in κ. This proves the binding property (cf. Definition 2) with respect to the class of PPT service providers that can execute a bounded number of threads in parallel only.

Notice that Mirror can easily cope with *(i)* different attacker strengths, and *(ii)* variable cost metrics, as follows:

Length of LFSR: One option would be to increase λ^*, i.e., the length of the LFSR communicated to the provider, and hence the number of values $g^{a_i^{(k)}}$ and $h^{b_j^{(k)}}$ the provider has to use for generating the replicas. In the extreme case, λ^* could be made equal to half of the total number of sectors $n \cdot s$—which would result into a scheme whose bandwidth consumption is comparable to [18].

Bitlength of coefficients: Another alternative would be to keep λ^* short, but to increase the bitlengths of the coefficients α_i^* and β_j^*. This would preserve the small bandwidth consumption of Mirror but would increase the time effort to run Replicate. This option will also not affect the latency borne by users in verifying the provider's response.

Hybrid approach: Clearly, one can also aim for a hybrid scheme, by increasing both the public LFSR length λ^* and the coefficients α_i^* and β_j^*.

In Section 5, we investigate reasonable choices of α_1^*, ρ, and T_{thr} to satisfy Equation 14. Of course, the same considerations can also be applied with respect to β_1^* which we omit for space reasons.

5 Implementation & Evaluation

In this section, we evaluate an implementation of Mirror within a realistic cloud setting and we compare the performance of Mirror to the MR-PDP solution of [18].

5.1 Implementation Setup

We implemented a prototype of Mirror in Scala. For a baseline comparison, we also implemented the multi-replica PDP protocol[7] of [18], which we denote by MR-PDP in the sequel (see Appendix A for a description of the MR-PDP of [18]). In our implementation, we relied on SHA-256, and the Scala built-in random number generator.

We deployed our implementation on a private network consisting of two 24-core Intel Xeon E5-2640 with 32GB of RAM. The storage server was running on one 24-core Xeon E5-2640 machine, whereas the clients and auditors were co-located on the second 24-core Xeon E5-2640 machine.

To emulate a realistic Wide Area Network (WAN), the communication between various machines was bridged using a 100 Mbps switch. All traffic exchanged on the networking interfaces of our machines was shaped with NetEm [31] to fit a Pareto distribution with a mean of 20 ms and a variance of 4 ms—thus emulating the packet delay variance specific to WANs [19].

In our setup, each client invokes an operation in a closed loop, i.e., a client may have at most one pending operation.

When implementing Mirror, we spawned multiple threads on the client machine, each thread corresponding to a unique worker handling a request of a client. Each data point in our plots is averaged over 10 independent measurements; where appropriate, we include the corresponding 95% confidence intervals.

---

[7]We acknowledge that PDP offers weaker guarantees than POR. However, to the best of our knowledge, no prior proposals for multi-replica-POR exist. MR-PDP thus offers one-of-the-few reasonable benchmarks for Mirror.
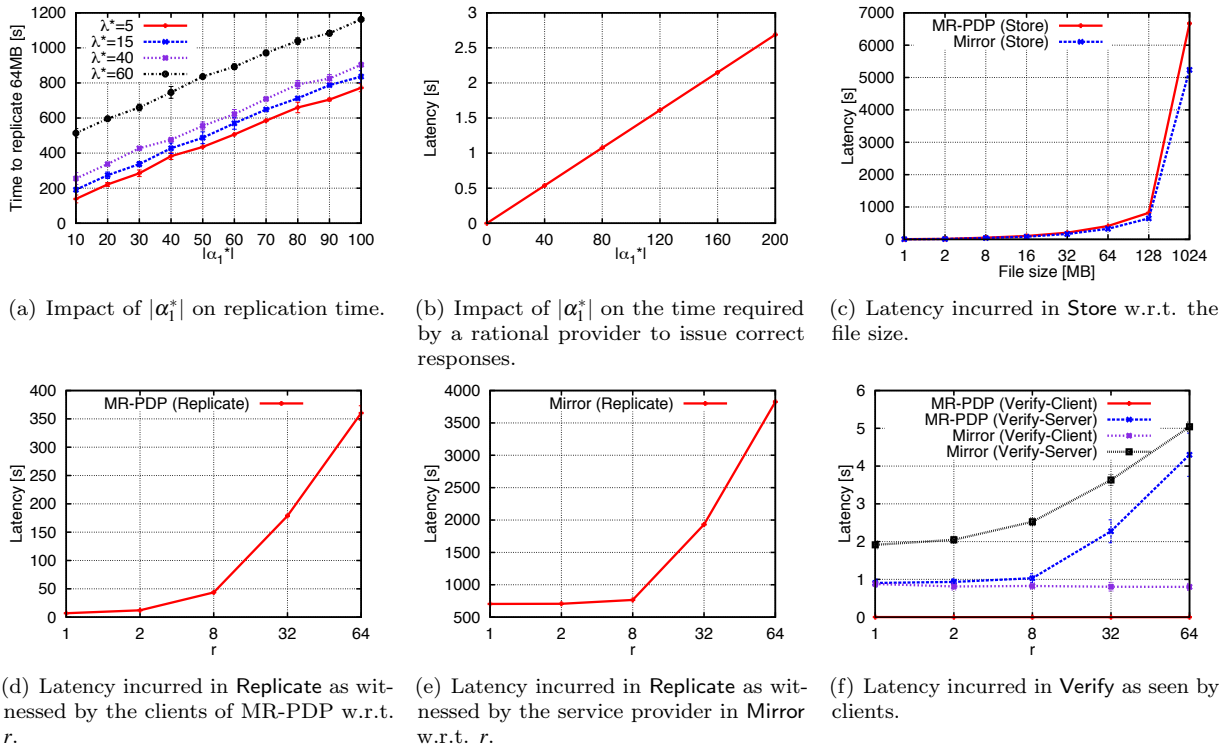
(a) Impact of $|\alpha_1^*|$ on replication time.

(b) Impact of $|\alpha_1^*|$ on the time required by a rational provider to issue correct responses.

(c) Latency incurred in Store w.r.t. the file size.

(d) Latency incurred in Replicate as witnessed by the clients of MR-PDP w.r.t. r.

(e) Latency incurred in Replicate as witnessed by the service provider in Mirror w.r.t. r.

(f) Latency incurred in Verify as seen by clients.

Figure 1: Performance evaluation of Mirror in comparison to the MR-PDP scheme of [18]. Each data point in our plots is averaged over 10 independent runs; where appropriate, we also show the corresponding 95% confidence intervals.

Parameter	Default Value
File size	64 MB
$|p|$	1024 bits
$|q|$	1024 bits
RSA modulus size	2048 bit
Number of challenges ℓ	40 challenges
Length of secret LFSR λ	2
Length of public LFSR λ^*	15
Fraction of stored sectors δ	0.9
Number of replicas r	2

Table 2: Default parameters used in the evaluation.

Table 2 summarizes the default parameters assumed in our setup.

5.2 Evaluation Results

Before evaluating the performance of Mirror, we start by analyzing the impact of the block size on the latency incurred in the verification of Mirror and in MR-PDP. Our results (Figure 4 in Appendix E) show that modest block sizes of 8 KB yield the most balanced performance, on average, across the investigated schemes. In the rest of our evaluation, we therefore set the block size to 8 KB.

Impact of the bitsize of α_1^*: In our implementation, our choice of parameters was mainly governed by the need to establish a tradeoff between the replication performance and the resource penalty incurred on a dishonest provider. To this end, we choose a small value for the public LFSR length λ^*, i.e., the LFSR length communicated to $\mathscr{S}$, and small coefficients α_i^* and β_j^* (these coefficients were set to 1 bit for $i, j > 1$). Recall that using smaller coefficients allows for faster exponentiations and hence a decreased replication effort.

However, as shown in Equation 14, the bitsize of α_1^* (which we shortly denote by $|\alpha_1^*|$ in the following) plays a paramount role in the security of Mirror. Note that the same analysis applies to β_1^*—which we do not further consider to keep the discussion short. Clearly, $|\alpha_1^*|$ (and λ^*) also impacts the file replication time at the service provider. In Figure 1(a), we evaluate the impact of $|\alpha_1^*|$ on the replication time, and on the time invested by a rational provider (who does not replicate) to answer every client challenge in Mirror. Our results indicate that the file replication time grows linearly with $|\alpha_1^*|$. Moreover, the higher λ^* is, the longer it takes to replicate a given file. On the other hand, as shown in Figure 1(b), $|\alpha_1^*|$ consid-

$|\alpha_1^*|$	Estimated EC2 costs per challenge (USD)
40	0.000058
70	0.00011
80	0.00013
120	0.00019

Table 3: Costs borne by a rational provider who computes the responses to a challenge of size $l = 40$ on the fly. We assume two replicas of size 64 MB, and estimate costs for a compute-optimized (extra large) instance from Amazon EC2 (at 0.42 USD per hour).

erably affects the time incurred on a rational provider which did not correctly replicate (some) user files. The larger $|\alpha_1^*|$ is, the longer it takes a misbehaving provider to reply to the user challenges, and thus the bigger are the amount of computational resources that the provider needs to invest in. Here, we assume the lower bound on the effort of a misbehaving provider (i.e., which only stores a fraction δ of the sectors per replica) given by Equation 14.

Setting $|\alpha_1^*|$: Following this analysis, suitable choices for α_1^* need to be large enough such that the costs borne by a rational provider who computes the responses on the fly are higher than those borne by an honest provider who correctly stores the replicas. In Table 3, we display the corresponding costs borne by a rational provider who computes the responses on the fly to a single challenge, assuming $l = 40$, and $r = 2$ replicas of size 64 MB. Here, we estimate the computation costs as follows: we interpolate the time required by a rational provider in answering challenges from Figure 1(c). We then estimate the corresponding computation costs assuming a compute-optimized (extra large) instance from Amazon EC2 (at 0.42 USD per hour), which offers comparable computing power than that used in our implementation setup.

For comparison purposes, notice that the cost of storing two 64 MB replicas per day (based on Amazon S3 pricing [9]) is approximately 0.00011 USD. *This shows that when instantiating Mirror with parameters $|\alpha_1^*| = 70$, the provider should not gain any (rational) advantage in misbehaving, if the user issues at least one PoR^2 challenge of $l = 40$ randomly selected blocks per day.* Clearly, users can increase the number of challenges that they issue accordingly to ensure that the costs borne by a rational provider are even more pronounced, e.g., to account for possible fluctuations in costs.

Following this analysis, we assume that $|\alpha_1^*| = 70$ throughout the rest of the evaluation. As shown in

Figure 1(a), this parameter choice results in reasonable replication times. e.g., when $\lambda^* = 5$ or $\lambda^* = 15$. Observe that, in this case, users can *detect/suspect* misbehavior by observing the cloud's response time. As shown in Figure 1(f), the typical response time of an honest service provider is less than 2 seconds when $r = 2$. An additional 0.9 seconds of delay (i.e., totalling 2.9 seconds) in responding to a challenge can be then detected by users.

Store performance: In Figure 1(c), we evaluate the latency incurred in Store with respect to the file size. Our findings suggest that the Store procedure of Mirror is considerably faster than that of MR-PDP. This is the case since the tag creation in Mirror requires fewer exponentiations per block (cf. Appendix A). For instance, the Store procedures is almost 20% faster than that of MR-PDP for files of 64MB in size.

Replicate performance: Figure 1(d) depicts the latency incurred on the clients of MR-PDP in the replicate procedure Replicate with respect to the number of replicas. Recall that, in MR-PDP, clients have to process and upload all replicas by themselves to the cloud. Clearly, the latency of Replicate increases with the number of replicas stored. Given our multi-threaded implementation, notice that the replication process can be performed in parallel. Here, as the number of concurrent replication requests increases, the threads in our thread pool are exhausted and the system saturates—which explains the sharp increase in the latency witnessed by clients who issue more than 8 concurrent replication requests. Notice that users of Mirror do not bear any overhead due to replication since this process is performed by the service provider.

In Figure 1(e), we show the latency incurred on the service provider in Mirror with respect to the number of replicas r. Since Mirror relies on puzzles, the replication process consumes considerable resources from the service provider. However, we point out that is a one-time effort per file, and can be performed offline (i.e., the provider can replicate files using "offline" resources in the back-end). For example, the creation of 8 additional 64 MB file replicas incurs a latency of almost 765 seconds. As mentioned earlier, Mirror trades this additional computational burden with bandwidth. Namely, users of Mirror only have to upload the file once, irrespective of the number of replicas desired. This, in turn, reduces the download bandwidth of providers and, as a consequence, the costs of offering the service.

In Figure 2, we estimate the costs of the additional computations incurred in Mirror for a 64 MB file,
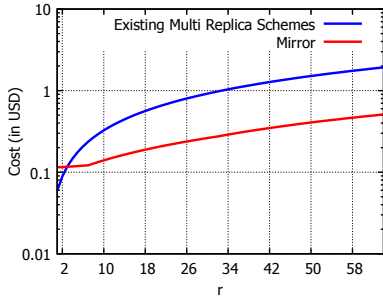
Figure 2: Replication costs for a 64 MB file (in USD) incurred Mirror vs. existing multi-replica schemes. We assume that the provider provisions a large general instance from Amazon EC2 at 0.441 USD per hour). We assume the replication time given by our experiments in Figure 1(e) and we estimate bandwidth costs by adapting the findings of [3] (cf. Table 3).

compared to those incurred by existing multi-replica schemes which require users to upload all the replicas. To estimate computing costs, we rely on the AWS pricing model [8]; we assume that the provider provisions a multi-core (compute-optimized) extra large instance from Amazon EC2 (at 0.441 USD per hour). We rely on our findings in Figure 1(e) to estimate the computing time for replication. We estimate bandwidth costs by adapting the findings of [3] (i.e., by assuming $5 per Mbps per month cf. Table 3). Our estimates suggest that Mirror considerably reduces the costs borne on the provider by trading bandwidth costs with the relatively cheaper computing costs. We expect that the cost savings of Mirror will be more significant for larger files, and/or additional replicas.

Verify **performance:** In Figure 1(f), we evaluate the latency witnessed by the users and service provider in the Verify procedure of Mirror and MR-PDP, respectively. Our findings show that the verification overhead witnessed by the service provider in Mirror is almost twice that of MR-PDP. Moreover, users of Mirror require almost 1 second to verify the response issued by the provider. Notice that the majority of this overhead is spent while computing/verifying the response to the issued challenge. This discrepancy mainly originates from the fact that the challenge-response in Mirror involves all the 32 sectors of each block in order to ensure the extractability of all replicas[8]. We contrast this to MR-PDP where each block comprises a single sector—which only ensures data/replica possession but does not provide

---

[8]We point out that this is not particular to Mirror and applies to all schemes which ensure retrievability (e.g., [34]).
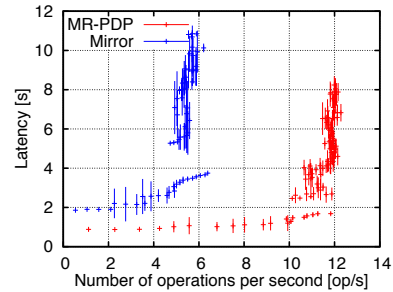

Figure 3: Latency vs. throughput comparison between MR-PDP and Mirror.

extractability guarantees.

In Figure 3, we evaluate the peak throughput exhibited by the service provider in the Verify procedure. Here, we require that the service provider handles verification requests back to back; we then gradually increase the number of requests in the system (until the throughput is saturated) and measure the associated latency. Our results confirm our previous analysis and show that Mirror attains a maximum throughput of 6 verification operations per second; on the other hand, the service provider in MR-PDP can handle almost 12 operations per second. As mentioned earlier, this discrepancy is mainly due to the fact that the MR-PDP blocks only comprise a single sector, whereas each block in Mirror comprises 32 sectors. However, we argue that the overhead introduced by our scheme compared to MR-PDP can be easily tolerated by clients; for instance, for 64 MB files, our proposal only incurs an additional latency overhead of 800 ms on the clients when compared to MR-PDP.

6   Related Work

Curtmola *et al.* propose in [18] a multi-replica PDP, which extends the basic PDP scheme in [12] and enables a user to verify that a file is replicated at least across *t* replicas by the cloud. In [16], Bowers *et al.* propose a scheme that enables a user to verify if his data is stored (redundantly) at multiple servers by measuring the time taken for a server to respond to a read request for a set of data blocks. In [13, 14], Barsoum and Hasan propose a multi-replica dynamic data possession scheme which allows users to verify multiple replicas, and to selectively update/insert their data blocks. This scheme builds upon the BLS-based SW scheme of [34]. In [22], the authors extend the dynamic PDP scheme of [21] to transparently support replication in distributed cloud storage systems. All existing schemes however share a common system model, where the user constructs and uploads the

replicas onto the cloud. On the other hand, Mirror conforms with the existing cloud model by allowing users need to process/upload their original files only once irrespective of the replication performed by the cloud provider.

Proofs of location (PoL) [32, 36] aim at proving the geographic position of data, e.g., if it is stored on servers within a certain country. In [36], Watson *et al.* provide a formal definition for PoL schemes by combining the use of geolocation techniques together with the SW POR schemes [34]. In [36], the authors assume a similar system model to Mirror, where the user uploads his files to the service provider only once. The latter then re-codes the tags of the file, and replicates content across different geo-located servers. Users can then execute individual PORs with each server to ensure that their data is stored in its entirety at the desired geographical location. We contrast this to our solution, where the user has to invoke a single Mirror instance to efficiently verify the integrity of all stored replicas.

Proofs of space [20] ensure that a prover can only respond correctly if he invests at least a certain amount of space or time per execution. However, this notion is not applicable to our scenario where we need to ensure that a *minimum* amount of space has been invested by the prover. Moreover, the instantiation in [20] does not support batch-verification which is essential in Mirror to conduct POR on several replicas in a single protocol run.

7   Conclusion

In this paper, we proposed a novel solution, Mirror, which enables users to efficiently verify the retrievability of their data replicas in the cloud. Unlike existing schemes, the cloud provider replicates the data by itself in Mirror; by doing so, Mirror trades expensive bandwidth resources with cheaper computing resources—a move which is likely to be welcomed by providers and customers since it promises better service while lowering costs.

Consequently, we see Mirror as one of the few economically-viable and workable solutions that enable the realization of verifiable replicated cloud storage.

8   Acknowledgements

References

[1] Amazon S3 Introduces Cross-Region Replication.

[2] Cloud Computing: Cloud Security Concerns. `http://technet.microsoft.com/en-us/magazine/hh536219.aspx`.

[3] The Relative Cost of Bandwidth Around the World.

[4] Amazon S3 Service Level Agreement, 2009. `http://aws.amazon.com/s3-sla/`.

[5] Are We Safeguarding Social Data?, 2009. MIT Technology Review, `http://www.technologyreview.com/view/412041/are-we-safeguarding-social-data/`.

[6] Microsoft Corporation. Windows Azure Pricing and Service Agreement, 2009.

[7] Protect data stored and shared in public cloud storage. `http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell_Data_Protection_Cloud_Edition_Data_Sheet.pdf`, 2013.

[8] Amazon EC2 Pricing, 2015. `https://aws.amazon.com/ec2/pricing/`.

[9] Amazon S3 Pricing, 2015. `http://aws.amazon.com/s3/pricing/?nc2=h_ls`.

[10] Google loses data after lightning strikes. `http://money.cnn.com/2015/08/19/technology/google-data-loss-lightning/`, 2015.

[11] 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*, CCS '14, pages 831–843, New York, NY, USA, 2014. ACM.

[12] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security*, pages 598–609, 2007.

[13] Ayad F. Barsoum and M. Anwar Hasan. Integrity verification of multiple data copies over untrusted cloud servers. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 829–834, 2012.

[14] Ayad F. Barsoum and M. Anwar Hasan. Provable multicopy dynamic data possession in cloud computing systems. *IEEE Transactions on Information Forensics and Security*, 10(3):485–497, 2015.

[15] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security*, pages 187–198, 2009.

[16] 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 *ACM Conference on Computer and Communications Security*, pages 501–514, 2011.

[17] Jin-yi Cai, Richard J. Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *Proceedings of the Eigth Annual Structure in Complexity Theory Conference, San Diego, CA, USA, May 18-21, 1993*, pages 2–11, 1993.

[18] Reza Curtmola, Osama Khan, Randal C. Burns, and Giuseppe Ateniese. MR-PDP: Multiple-Replica Provable Data Possession. In *ICDCS*, pages 411–420, 2008.

[19] 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*, CCS '13, pages 285–298, New York, NY, USA, 2013. ACM.

[20] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605. Springer, 2015.

[21] C. Christopher Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security*, pages 213–222, 2009.

[22] Mohammad Etemad and Alptekin Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 1–18, Berlin, Heidelberg, 2013. Springer-Verlag.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[24] Jim Gray. Distributed computing economics. *Queue*, 6(3):63–68, May 2008.

[25] Ari Juels and Burton S. Kaliski Jr. PORs: Proofs Of Retrievability for Large Files. In *ACM Conference on Computer and Communications Security*, pages 584–597, 2007.

[26] Ghassan Karame and Srdjan Capkun. Low-cost client puzzles based on modular exponentiation. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, pages 679–697, 2010.

[27] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[28] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[29] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, USA, 1986.

[30] Yadi Ma, Thyaga Nandagopal, Krishna P. N. Puttaswamy, and Suman Banerjee. An ensemble of replication and erasure codes for cloud file systems. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pages 1276–1284, 2013.

[31] NetEm. NetEm, the Linux Foundation. Website, 2009. Available online at http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[32] Zachary N. J. Peterson, Mark Gondree, and Robert Beverly. A position paper on data sovereignty: The importance of geolocating data in the cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[33] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[34] Hovav Shacham and Brent Waters. Compact Proofs of Retrievability. In *ASIACRYPT*, pages 90–107, 2008.

[35] 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 Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 265–280. ACM, 2012.

[36] Gaven J. Watson, Reihaneh Safavi-Naini, Mohsen Alimomeni, Michael E. Locasto, and Shivaramakrishnan Narayan. LoSt: location based storage. In Ting Yu, Srdjan Capkun, and Seny Kamara, editors, *CCSW*, pages 59–70. ACM, 2012.

A MR-PDP

In what follows, we briefly describe the multi-replica provable data possession scheme by Curtmola *et al.* [18]. Here, the user first splits the file *D* into

n blocks $d_1, \ldots, d_n \in \mathbb{Z}_N$. Let $p = 2p' + 1, q = 2q' + 1$ be safe primes, and $N = pq$ an RSA modulus; moreover, let g be a generator of the quadratic residues of $\mathbb{Z}_N^*$, and e, d a pair of integers such that $e \cdot d = 1 \mod p'q'$. The user creates authentication tags for each block $i \in [1, n]$ by computing $T_i \leftarrow (h(v||i)g^{d_i})^d \mod N$, where $h : \{0, 1\}^* \rightarrow \mathbb{Z}_N$ is a hash function and $v \in \mathbb{Z}_N$.

Subsequently, each replica is created by the user as follows: $d_i^{(k)} \leftarrow d_i + \mathsf{PRF}(k||i)$ where PRF denotes a pseudorandom function. The user sends to the service provider the tags $\{T_i\}$, the original file blocks $\{d_i\}$, and the replica blocks $d_i^{(k)}$.

At the verification stage, the user selects a replica k and creates a (pseudo-)random challenge set $I = \{(k_1, i_1), \ldots, (k_\ell, i_\ell)\}$ where k_j denotes the replica number and i_j the block index. In addition, the user picks $s \in \mathbb{Z}_N^*$, and computes $g_s = g^s \mod N$. The challenge query then comprises the set I and the value g_s which are both sent to the service provider who stores replica k. The service provider then computes the response (T, σ) as follows and sends it back to the verifier:

$$T \leftarrow \prod_{i \in I} T_i, \quad \sigma \leftarrow g_s^{\sum_{1 \leq j \leq \ell} d_{i_j}^{(k_j)}}$$

Finally, the user checks whether:

$$\sigma \stackrel{?}{=} \left(\frac{T^e}{\prod h(v||i)} g^{\sum_{1 \leq j \leq \ell} \mathsf{PRF}(k_j||i_j)} \right)^s$$

B POR Schemes of Shacham and Waters

In what follows, we briefly describe the private POR scheme by Shacham and Waters [34]. This scheme leverages a pseudo-random function PRF. Here, the user first applies an erasure code to the file and then splits it into n blocks $d_1, \ldots, d_n \in \mathbb{Z}_p$, where p is a large prime. The user then chooses a random $\alpha \in_R \mathbb{Z}_p$ and creates for each block an authentication value as follows:

$$\sigma_i = \mathsf{PRF}_{key}(i) + \alpha \cdot d_i \in \mathbb{Z}_p. \tag{15}$$

The blocks $\{d_i\}$ and their authentication values $\{\sigma_i\}$ are all stored at the service provider in D^*.

At the POR verification stage, the verifier (here, the user) chooses a random challenge set $I \subset \{1, \ldots, n\}$ of size ℓ, and ℓ random coefficients $v_i \in_R \mathbb{Z}_p$. The challenge query then is the set $Q := \{(i, v_i)\}_{i \in I}$ which is sent to the prover (here, service provider). The

prover computes the response (σ, μ) as follows and sends it back to the verifier:

$$\sigma \leftarrow \sum_{(i, v_i) \in Q} v_i \sigma_i, \quad \mu \leftarrow \sum_{(i, v_i) \in Q} v_i d_i.$$

Finally, the verifier checks the correctness of the response:

$$\sigma \stackrel{?}{=} \alpha \mu + \sum_{(i, v_i) \in Q} v_i \cdot \mathsf{PRF}(i).$$

C Improving User Verification in Mirror

In what follows, we describe a number of optimizations that we adopted in our implementation in order to reduce the effort in verifying the service provider's response.

Using either g or h: Recall that the service provider's response involves powers of g and of h which have order p' and q', respectively. One technique that allows to reduce the effort on the user's side is to rely on either g or h. That is, at the beginning of the verification step, the user randomly decides whether only g or only h shall be taken into account. For example, let us assume that the choice falls on g. Then, the user proceeds as follows:

1. The user computes:

$$\tilde{\sigma} := \sigma^{q'} \cdot \prod_{c=1}^{\ell} \left(\prod_{j=1}^{s} \prod_{k \in R} g_{i_c, j}^k \right)^{-(q' \cdot v_c)}. \tag{16}$$

Here, we exploit the fact that $(h^e)^{q'} = 1$ for any e.

2. The user checks if:

$$\left(\prod_{j=1}^{s} \mu_j^{\varepsilon_j + |R|} \right)^{q'} = \tilde{\sigma}. \tag{17}$$

This approach incurs two additional exponentiations but completely eliminates the need to compute the expressions for the values h.

Representing LFSRs by Pre-computed Matrices: According to our experiments, suitable parameter choices are to choose the length λ of the secret LFSR quite small, e.g., equal to 2, while the block size s is comparatively large. This motivates the following optimization.

Let $A_t^{(k)} := (a_t^{(k)}, \ldots, a_{t+\lambda-1}^{(k)})$ for any $t \geq 1$ and any $k \in \{1, \ldots, r\}$. That is, $A_1^{(k)}$ denotes the initial state of the k-th LFSR while $A_t^{(k)}$ denotes the state after $t - 1$

clocks. Recall that we consider r LFSR sequences which are all generated by the same feedback function. Namely, it holds for any $t \geq 1$ and any $k \in \{1, \ldots, r\}$ that $a_{t+\lambda}^{(k)} = \sum_{i=1}^{\lambda} \alpha_i \cdot a_{t+i-1}^{(k)}$. Due to the linearity of the feedback function, there exists a $\lambda \times \lambda$ matrix M, called the companion matrix, for which it holds that:

$$M^t \cdot A_1^{(k)} = A_{t+1}^{(k)}, \quad \forall t \geq 0. \tag{18}$$

Recall that we aim to compute for each $i \in I$ the value

$$\sum_{k \in R} \left(a_{\pi(i_c,1)}^{(k)} + a_{\pi(i_c,1)+1}^{(k)} + \ldots + a_{\pi(i_c,1)+s-1}^{(k)} \right) \tag{19}$$

where $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is a mapping such that $g_{i,j}^{(k)} = g_{\pi(i,j)}^{(k)} = g^{a_t^{(k)}}$ and to raise g by the resulting value. The idea is now to combine as many computations as possible to reduce the overall effort. To this end, the goal is to sum-up for each $k \in R$ and each $i \in I$ the following internal states:

$$A_{\pi(i,1)}^{(k)} = (a_{\pi(i,1)}^{(k)}, \ldots, a_{\pi(i,1)+\lambda-1}^{(k)})$$
$$\vdots$$
$$A_{\pi(i,1)+\lfloor s/\lambda \rfloor \cdot \lambda}^{(k)} = (a_{\pi(i,1)+\lfloor s/\lambda \rfloor \cdot \lambda}^{(k)}, \ldots, a_{\pi(i,1)+\lfloor s/\lambda \rfloor \cdot \lambda-1}^{(k)})$$

This can be accomplished by computing:

$$\left(\sum_{i \in I} M^{\pi(i,1)} \right) \cdot \left(\sum_{j=0}^{\lfloor s/\lambda \rfloor} M^{j \cdot \lambda} \right) \cdot \left(\sum_{k \in R} A_1^{(k)} \right). \tag{20}$$

Observe that $\sum_{j=0}^{\lfloor s/\lambda \rfloor} M^{j \cdot \lambda}$ is independent of the current challenge and can be precomputed. Moreover, due to the fact that we aim for a small LFSR length, e.g., $\lambda = 2$, the user may consider to precompute the value in the last bracket for any choice of R, yielding an additional storage effort of $\lambda \cdot (2^r - 1)$ sectors. In such case, the computation would boil down to the effort of computing the first bracket only. We note that if λ does not divide s, then the user has to compute in addition:

$$\left(\sum_{i \in I} M^{\pi(i,1)} \right) \cdot M^{\lceil s/\lambda \rceil \cdot \lambda} \cdot \left(\sum_{k \in R} A_1^{(k)} \right), \tag{21}$$

and to add the sum of the first $s \mod \lambda$ entries to the value computed above. Also here, similar precomputations can be done to accelerate this step.

D Valid Relations

We now explain why $\vec{v} = (v_1, \ldots, v_{n \cdot s}) \in \mathbb{Z}^{n \cdot s}$ such that

$$\prod_{i=1}^{n \cdot s} g_i^{v_i} = 1, \tag{22}$$

represents the only type of equations that allows the provider to compute missing values g_j from known values g_i.

To see why, recall that $g_j = g^{a_j}$ where $(a_j)_j$ represents an LFSR-sequence. More precisely, this sequence is defined by the feedback polynomial and the initial state, i.e., the first λ entries. Without knowing these entries, it is (information-theoretically) impossible to determine a_j for larger indices. Also, any element in $(a_j)_j$ is a linear combination of the initial state values. Let us denote this combination by L_1. This can be relaxed as follows: the knowledge of any λ elements $a_{j_1}, \ldots, a_{j_\lambda}$ of the sequence $(a_j)_j$ allows almost always to compute another element by an appropriate linear combination (say L_2) of $a_{j_1}, \ldots, a_{j_\lambda}$. Notice that the coefficients of L_2 have to be a linear combination of the coefficients (or shifted versions) of L_1 (this is in inherent property of LFSRs). This is exactly the definition of "valid relations" given in Equation (12).

E Impact of the Block Size on the Performance of MR-PDP and Mirror
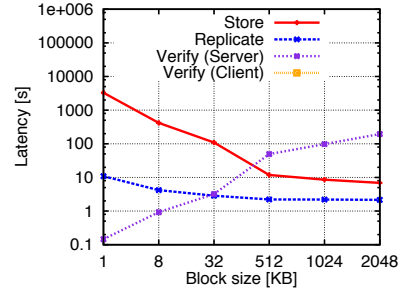
(a) Impact of block size on the performance of MR-PDP [18].
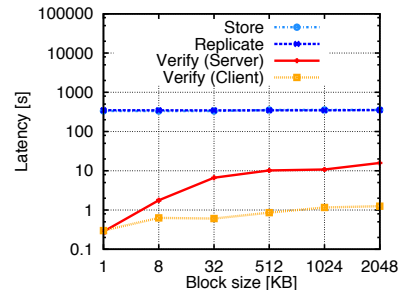
(b) Impact of block size on the performance of Mirror.

Figure 4: Impact of the block size on the performance of MR-PDP [18] and Mirror.