PORTOS: Proof of Data Reliability for Real-World Distributed Outsourced Storage

Dimitrios Vasilopoulos, Melek Önen and Refik Molva
EURECOM, Sophia Antipolis, France

Keywords: Secure Cloud Storage, Proofs of Reliability, Reliable Storage, Verifiable Storage.

Abstract: Proofs of data reliability are cryptographic protocols that provide assurance to a user that a cloud storage system correctly stores her data and has provisioned sufficient redundancy to be able to guarantee reliable storage service. In this paper, we consider distributed cloud storage systems that make use of erasure codes to guarantee data reliability. We propose a novel proof of data reliability scheme, named PORTOS, that on the one hand guarantees the retrieval of the outsourced data in their entirety through the use of proofs of data possession and on the other hand ensures the actual storage of redundancy. PORTOS makes sure that redundancy is stored at rest and not computed on-the-fly (whenever requested) thanks to the use of time-lock puzzles. Furthermore, PORTOS delegates the burden of generating the redundancy to the cloud. The repair operations are also taken care of by the cloud. Hence, PORTOS is compatible with the current cloud computing model where the cloud autonomously performs all maintenance operations without any interaction with the user. The security of the solution is proved in the face of a rational adversary whereby the cheating cloud provider tries to gain storage savings without increasing its total operational cost.

1 INTRODUCTION

Distributed storage systems guarantee data reliability by redundantly storing data on multiple storage nodes. Depending on the redundancy mechanism, each storage node may store either a full copy of the data (replication) or a fragment of an encoded version of it (erasure codes). In the context of a trusted environment such as an on-premise storage system, this type of setting is sufficient to guarantee data reliability against accidental data loss caused by random hardware failures or software bugs.

With the prevalence of cloud computing, outsourcing data storage has become a lucrative option for many enterprises as they can meet their rapidly changing storage requirements without having to make tremendous upfront investments for infrastructure and maintenance. However, in the cloud setting, outsourcing raises new security concerns with respect to misbehaving or malicious cloud providers. An economically motivated cloud storage provider could opt to “take shortcuts” when it comes to data maintenance in order to maximize its returns and thus jeopardizing the reliability of user data. In addition, cloud storage providers currently do not assume any liability in case data is lost or damaged. Hence, cloud users may be reluctant to adopt cloud storage services for applications that are otherwise perfectly suited for the cloud such as archival storage or backups. As a result, technical solutions that aim at allowing users to verify the long-term integrity and reliability of their data, would be beneficial both to customers and providers of cloud storage services.

Focusing on the long term integrity requirement, the literature features a number of solutions for verifiable outsourced storage. Notably, proofs of retrievability (PoR) (Juels and Kaliski, 2007; Shacham, H. and Waters, B., 2008) and data possession (PDP) (Ateniese et al., 2007) are cryptographic mechanisms that enable a cloud user to efficiently verify that her data is correctly stored by an untrusted cloud storage provider.

PoRs and PDPs have been extended in order to ensure data reliability over time introducing the notion of proofs of data reliability (Curtmola et al., 2008; Bowers et al., 2009; Armknecht et al., 2016; Vasilopoulos et al., 2018). In addition to the integrity of a user’s outsourced data, a proof of data reliability provides the user with the assurance that the cloud storage provider has provisioned sufficient redundancy to be able to guarantee reliable storage service. In a straightforward approach, the cloud user...
locally generates the necessary redundancy and subsequently stores the data together with the redundancy on multiple storage nodes. Previous work has established that, in the case of erasure code-based storage systems, the relation between the data and redundancy symbols should stay hidden from the cloud storage provider, otherwise, the latter could store a portion of the encoded data only and compute any missing symbols upon request. Similarly, in the case of replication-based storage systems, each storage node should store a different replica; otherwise, the cloud storage provider could simply store a single replica. As a result, most proof of data reliability schemes (Curtmola et al., 2008; Chen and Curtmola, 2017; Bowers et al., 2009; Chen et al., 2015) require some interaction with the user to repair damaged data. Hence these schemes are at odds with automatic maintenance that is a key feature of cloud storage systems.

In this paper, we propose PORTOS, a proof of data reliability scheme tailored to distributed cloud storage systems that makes use of erasure codes to guarantee data reliability. PORTOS achieves the following properties against a rational cloud storage provider:

- **Data reliability against t storage node failures:** PORTOS uses a systematic linear erasure code to add redundancy to the outsourced data which thereafter stores across multiple storage nodes. The system can tolerate the failure of up to t storage nodes, and successfully reconstruct the original data using the contents of the surviving ones.

- **Data and redundancy integrity:** PORTOS leverages a PDP scheme to verify the integrity of the outsourced data and it further takes advantage of the homomorphic properties of the PDP tags, in order to verify the integrity of the redundancy. Moreover, thanks to the combination of the PDP scheme with erasure codes, PORTOS can provide a cloud user with the assurance that she can recover her data in their entirety.

- **Automatic data maintenance by the cloud storage provider:** In PORTOS, the cloud storage provider has the means to generate the required redundancy, detect storage node failures and repair corrupted data entirely on its own without any interaction with the user conforming to the current cloud model. This setting, however, allows a malicious cloud storage provider to delete a portion of the encoded data and compute any missing symbols upon request. To defend against such an attack, PORTOS relies on time-lock puzzles, and masks the data, making the symbol regeneration process time-consuming. In this way, a rational cloud is provided with a strong incentive to conform to the proof of data reliability protocol.

- **Real-world cloud storage architecture:** PORTOS conforms to the current model of erasure-code-based distributed storage systems. Moreover, it does not make any assumption regarding the system’s underlying technology as opposed to prior proof of data reliability schemes that also allow for automatic data maintenance by the cloud storage provider.

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

- We propose a new formal definition for proofs of data reliability, which is more generic than the definitions presented in prior work (Armknecht et al., 2016).

- We present a novel proof of data reliability scheme, named PORTOS, that on the one hand guarantees that the outsourced data can be retrieved in their entirety through the use of proofs of data possession and on the other hand ensures the actual storage of redundancy. PORTOS makes sure that redundancy is stored at rest and not computed on-the-fly (whenever requested) thanks to the use of time-lock puzzles. Furthermore, PORTOS delegates the burden of generating the redundancy, as well as the repair operations, to the cloud storage provider.

- We show that PORTOS is secure against a rational adversary, and we further evaluate the impact of two types of attacks, considering in both cases the most favorable scenario for the adversary. Finally, we evaluate the performance of PORTOS.

The remaining of this paper is organized as follows: In Section 2, we give an overview of prior work in the field. We describe the formal definition, adversary model, and security requirements of a proof of data reliability scheme in Section 3. In Section 4, we introduce PORTOS, a novel data reliability scheme and we analyze its security in Section 5. Finally, in Section 6 we analyze the performance of PORTOS.

## 2 PRIOR WORK

**Replication-based Proofs of Data Reliability.** The authors in (Curtmola et al., 2008) propose a proof of data possession scheme (PDP) which extends the PDP scheme in (Ateniese et al., 2007) and enables the client to verify that the cloud provider stores at least t replicas of her data. In (Chen and Curtmola, 2013; Chen and Curtmola, 2017), the replica differentiation
mechanism in (Curtmola et al., 2008) is replaced by a tunable masking mechanism, which allows to shift the bulk of repair operations to the cloud storage provider with the user acting as a repair coordinator. In (Leontiadis and Curtmola, 2018), the scheme in (Chen and Curtmola, 2017) is extended in order to construct a proof of data reliability protocol that allows for cross-user file-level deduplication. The authors in (Barsoum and Hasan, 2012; Barsoum and Hasan, 2015) propose a multi-replica dynamic PDP scheme that enables clients to update/insert selected data blocks and to verify multiple replicas of their outsourced files. The scheme in (Etemad and Küpcü, 2013), extends the dynamic PDP scheme in (Erway et al., 2009) in order to transparently support replication in distributed cloud storage systems. In (Armknecht et al., 2016), the authors propose a multi-replica PoR scheme that delegates the replica construction to the cloud storage provider. The scheme uses tunable puzzles as part of its replication mechanism in order to force the cloud storage provider to store the replicas at rest.

**Erasure-code-based Proofs of Data Reliability.**

The authors in (Bowers et al., 2009) propose HAIL, which provides a high availability and integrity layer for cloud storage. HAIL uses erasure codes in order to guarantee data retrievability and reliability among distributed storage servers, and enables a user to detect and repair data corruption. The work in (Chen et al., 2015) redesigns parts of (Bowers et al., 2009) in order to achieve a more efficient repair phase that shifts the bulk computations to the cloud side. In (Chen et al., 2010), the authors present a remote data checking scheme for network-coding-based distributed storage systems that minimizes the communication overhead of the repair component compared to erasure-coding-based approaches. The work in (Le and Markopoulou, 2012) extends the scheme in (Chen et al., 2010) by improving the repair mechanism in order to reduce the computation cost for the client, and introducing a third party auditor. Based on the introduction of this new entity, the authors in (Thao and Omote, 2016) design a network-coding-based PoR scheme in which the repair mechanism is executed between the cloud provider and the third party auditor without any interaction with the client. The authors in (Bowers et al., 2011) propose RAFT, an erasure-code-based protocol that can be seen as a proof of fault tolerance. RAFT relies on technical characteristics of rotational hard drives in order to construct a time-based challenge, that enables a client to verify that her encoded data is stored at multiple storage nodes within the same data center. In (Vasilopoulos et al., 2018), the authors propose POROS, a proof of data reliability scheme for erasure-code-based cloud storage systems which combines PDP with time constrained operations in order to force the cloud storage provider to store the redundancy at rest. POROS dictates that all the redundancy is stored permuted on a single storage node and leverages the technical characteristics of rotational hard drives to set a time threshold for the generation of the proof. Both schemes enable the outsourcing of the data encoding to the cloud storage provider as well as automatic maintenance operations without any interaction with the client.

Most of the proof of data reliability schemes presented above share a common system model, where the user generates the required redundancy locally, before uploading it together with the data to the cloud storage provider. Furthermore, when corruption is detected, the cloud cannot repair it autonomously, because either it expects some input from another entity or all computations are performed by the client. The solution in (Armknecht et al., 2016) outsources to the cloud the redundancy generation, however, it relies on replication to provide data reliability and hence is not directly comparable to our proof of data reliability scheme.

PORTOS is directly comparable with the work in (Bowers et al., 2011) and in (Vasilopoulos et al., 2018), in the sense that it also uses erasure codes to guarantee data reliability and it delegates the burden of generating the redundancy, as well as the repair operations, to the cloud storage provider. Nonetheless, in contrast to these schemes, our solution does not make any assumption regarding the underlying storage technology. Namely, both schemes rely on technical characteristics of rotational hard drives in order to set a time-threshold for the cloud servers to respond to a read request for a set of data blocks. Furthermore, unlike in PORTOS, the challenge verification in (Bowers et al., 2011) requires that a copy of the encoded data is stored locally at the user. Lastly, the proof of data reliability scheme in (Vasilopoulos et al., 2018) requires that all redundancy symbols are stored on a single storage node without fragmentation. On the contrary, PORTOS does not deviate from the standard model of erasure-code based distributed storage systems.
3 PROOFS OF DATA RELIABILITY

3.1 Environment

We consider a setting where a user U produces a data object D from a file D and subsequently outsources D to an untrusted cloud storage provider C who commits to store D in its entirety across a set of n storage nodes \{S(j)\}_{1 \leq j \leq n} with reliability guarantee t: storage service guarantee against t storage node failures. We define a proof of data reliability scheme as a protocol executed between the cloud storage provider C with its storage nodes \{S(j)\}_{1 \leq j \leq n} on the one hand and a verifier V on the other hand. The aim of such a protocol is to enable V to check and validate (i) the integrity of D and, (ii) whether the reliability guarantee t is satisfied. In order not to cancel out the storage and performance advantages of the cloud, all this verification should be performed without V downloading the entire content associated to D from \{S(j)\}_{1 \leq j \leq n}. We consider two different settings for a proof of data reliability scheme: a private one where U is the verifier and a public one where V can be any third party except for C.

Formal Definition. A proof of data reliability scheme comprises seven polynomial time algorithms:

- **Setup** (\{1^n, t\} \rightarrow (\{S(j)\}_{1 \leq j \leq n}, \text{param}_{\text{system}})): This algorithm takes as input the security parameter \lambda and the reliability parameter t, and returns the set of storage nodes \{S(j)\}_{1 \leq j \leq n}, the system parameters \text{param}_{\text{system}}, and the specification of the redundancy mechanism.

- **Store** (\{1^n, D\} \rightarrow (K_U, D, \text{param}_g)): This randomized algorithm invoked by the user U takes as input the security parameter \lambda and the to-be-outsourced file D, and outputs the user key K_U, the verifiable data object D, which also includes a unique identifier fid, and optionally, a set of data object parameters \text{param}_g. In a private proof of data reliability scheme K_U := sk is the user’s secret key, whereas in a public one K_U := (sk, pk) is the user’s private/public key pair.

- **GenR** (D, \text{param}_{\text{system}}, \text{param}_g) \rightarrow (\hat{D}) : This algorithm takes as input the verifiable data object D, the system parameters \text{param}_{\text{system}}, and optionally, the data object parameters \text{param}_g, and outputs the data object \hat{D}. Algorithm GenR may be invoked either by the user U, when the user generates the redundancy on her own; or by C, when the redundancy computation is entirely outsourced to the cloud storage provider. Depending on the redundancy mechanism, \hat{D} may comprise multiple copies of D or an encoded version of it. Additionally, algorithm GenR generates the necessary integrity values that will further help for the integrity verification of D’s redundancy.

- **Chall** (K_V, \text{param}_{\text{system}}) \rightarrow (chal): This stateful and probabilistic algorithm invoked by the verifier V takes as input the verifier key K_V and the system parameters \text{param}_{\text{system}}, and outputs a challenge chal. In a private proof of data reliability scheme the verifier key is the user’s secret key (K_V := sk), whereas in a public one the verifier key is the user’s public key (K_V := pk).

- **Prove** (chal, \hat{D}) \rightarrow (proof): This algorithm invoked by C takes as input the challenge chal and the data object \hat{D}, and returns C’s response proof of data reliability.

- **Verify** (K_V, chal, proof, \text{param}_g) \rightarrow (dec): This deterministic algorithm invoked by V takes as input C’s proof corresponding to a challenge chal, the verifier key K_V, and optionally, the data object parameters \text{param}_g, and outputs a decision dec \in \{accept, reject\} indicating a successful or failed verification of the proof, respectively.

- **Repair** (\hat{D}, \text{param}_{\text{system}}, \text{param}_g) \rightarrow (\tilde{D}) : This algorithm takes as input a corrupted data object *\hat{D} together with its parameters \text{param}_g and the system parameters \text{param}_{\text{system}}, and either reconstructs \tilde{D} or outputs a failure symbol \perp. Algorithm Repair may be invoked either by U or C depending on the proof of data reliability scheme.

3.2 Adversary Model

Similar to (Armknecht et al., 2016; Vasilopoulos et al., 2018), we consider an adversary model whereby the cloud storage provider C is rational, in the sense that C decides to cheat only if it achieves some cost savings. For a proof of data reliability scheme that deals with the storage of data and its redundancy, a rational adversary would try to save some storage space without increasing its overall operational cost. The overall operational cost is restricted to the maximum number n of storage nodes \{S(j)\}_{1 \leq j \leq n} whereby each of them has a bounded capacity of storage and computational resources. More specifically, assume that for some proof of data reliability scheme there exists an attack which allows C to produce a
valid proof of data reliability while not fulfilling the reliability guarantee $t$. If in order to mount this attack, $C$ has to provision either more storage resources or excessive computational resources compared to the resources required when it implements the protocol in a correct manner, then a rational $C$ will choose not to launch his attack.

3.3 Security Requirements

A proof of data reliability scheme oughts to fulfill the following requirements.

**Req 0:** **Correctness.** A proof of data reliability scheme should be correct: if both the cloud storage provider $C$ and the verifier $V$ are honest, then on input $\text{chall}$ sent by the verifier $V$, using algorithm $\text{Chall}$, algorithm $\text{Prove}$ (invoked by $C$) generates a Proof of Data Reliability proof such that algorithm $\text{Verify}$ yields accept with probability 1. Put differently, an honest $C$ should always be able to pass the verification of proof of data reliability.

**Req 1:** **Extractability.** It is essential for any proof of data reliability scheme to ensure that an honest user $U$ can recover her file $D$ with high probability. This guarantee is formalized using the notion of extractability introduced in (Juels and Kaliski, 2007; Shacham, H. and Waters, B., 2008). The environment simulates all honest users and verifiers, and it further provides the $A$ with oracles for the algorithms Setup, Store, Chall, and Verify. $A$ interacts with the environment and requests $\text{O}_{\text{store}}$ to compute the tuple $(\text{sk}_D, \text{param}_D)$ for several chosen files $D$ and for different honest users. Thereafter, $A$ invokes a series of proof of data reliability executions by interacting with $\text{O}_{\text{chall}}$ and $\text{O}_{\text{verify}}$. Finally, $A$ picks a user $U$ and a tuple $(\text{sk}_D, \text{param}_D)$, corresponding to some file $D$ and simulates a cheating cloud storage provider $C$. Let $C$ succeed in making algorithm $\text{Verify}$ yield $\text{dec} := \text{accept}$ in an non-negligible $\epsilon$ fraction of proof of data reliability executions. We say that the proof of data reliability scheme meets the extractability guarantee, if there exists an extractor algorithm such that given sufficient interactions with $C$, it recovers $D$.

**Req 2:** **Soundness of Redundancy Generation.** In addition to the extractability guarantee a proof of data reliability scheme should ensure the soundness of the redundancy generation mechanism. This entails that, in the face of data corruption, the original file $D$ can be effectively reconstructed using the generated redundancy. Hence, in proof of data reliability schemes wherein algorithm $\text{GenR}$ is implemented by the cloud storage provider $C$, it is crucial to ensure that the latter performs this operation in a correct manner. Namely, an encoded data object should either consist of actual codeword symbols or all replicas should be genuine copies of the data object. In other words, the only way $C$ can produce a valid proof of data reliability is by correctly generating the redundancy.

**Req 3:** **Storage Allocation Commitment.** A crucial aspect of a Proof of Data Reliability scheme, is forcing a cloud storage provider $C$ to store at rest the outsourced data object $D$ together with the relevant redundancy. This requirement is formalized similarly to the storage allocation guarantee introduced in (Armknecht et al., 2016). A cheating cloud storage provider $C'$ that participates in the above mentioned extractability game (see Req 1), and dedicates only a fraction of the storage space required for storing both $D$ and its redundancy in their entirety, cannot convince the verifier $V$ to accept its proof with overwhelming probability.

4 PORTOS

In this section, we present PORTOS: a proof of data reliability scheme. PORTOS is based on the use of erasure codes to offer reliable storage with automatic maintenance. A user $U$ sends a data object $D$ to a cloud storage provider $C$, which in turn encodes $D$ using a systematic linear $(k, n)$-MDS code (Xing and Ling, 2003), and thereupon stores it across a set of $n$ storage nodes $\{S^{[j]}\}_{1 \leq j \leq n}$ with reliability guarantee against $t$ storage node failures. Moreover, in PORTOS, $C$ has the means to generate the required redundancy, detect storage node failures, and repair corrupted data, entirely on its own, without any interaction with $U$.

Such setting, however, presents a cheating $C$ with the opportunity to misbehave, for instance, by not fulfilling the storage allocation commitment requirement and computing the redundancy symbols on-thefly upon request. To prevent such behavior, PORTOS leverages time-lock puzzles in order to augment the resources (storage and computational) a cheating $C$ has to provision in order to produce a valid proof of
data reliability. Nonetheless, this mechanism does not incur any additional storage or computational cost to an honest $C$ that generates the same proof. Moreover, the puzzle difficulty can be adapted to $C$’s computational capacity as it evolves over time. Therefore, a rational $C$ is provided with a strong incentive to conform to the proof of data reliability protocol rather than attempt to misbehave. Figure 1 depicts the steps of PORTOS outsourcing process.

4.1 Building Blocks

**MDS Codes.** Maximum distance separable (MDS) codes (Xing and Ling, 2003; Suh and Ramchandran, 2011) are a class of linear block erasure codes used in reliable storage systems that achieve the highest error-correcting capabilities for the amount of storage space dedicated to redundancy. A $(k, n)$-MDS code encodes a data segment of size $k$ symbols into a codeword comprising $n$ code symbols. The input data symbols and the corresponding code symbols are elements of a finite field $\mathbb{F}_q$, where $q$ is a large prime and $k \leq n \leq q$. In the event of data corruption, the original data segment can be reconstructed from any set of $k$ code symbols. Furthermore, up to $n - k + 1$ corrupted symbols can be repaired. The new code symbols can either be identical to the lost ones, in which case we have exact repair, or can be functionally equivalent, in which case we have functional repair where the original code properties are preserved.

A systematic linear MDS-code has a generator matrix of the form $G = [I_k \mid P]$ and a parity check matrix of the form $H = [-P^\top \mid I_{n-k}]$. Hence, in a systematic code, the code symbols of a codeword include the data symbols of the original segment. Reed-Solomon codes (Xing and Ling, 2003) are a typical example of MDS codes, their generator matrix $G$ can be easily defined for any given values of $(k, n)$, and are used by a number of storage systems (Blaum et al., 1994).

**Linearly–homomorphic Tags.** PORTOS’s integrity guarantee derives from the use of linearly-homomorphic tags proposed, for the design of the private PoR scheme in (Shacham, H. and Waters, B., 2008).

This scheme consists of the following algorithms:

**SW.Store** $(1^\lambda, D) \rightarrow (\text{fid}, \text{sk}, D)$: This randomized algorithm invoked by the user $U$, first picks a pseudo-random function $\text{PRF} : \{0, 1\}^\lambda \times \{0, 1\}^* \rightarrow \mathbb{Z}_q$, together with its pseudo-randomly generated key $\text{k}_{\text{prf}} \in \{0, 1\}^\lambda$ and a non-zero element $\alpha \leftarrow \mathbb{Z}_q$, and subsequently computes a homomorphic authentication tag for each symbol $d_i$ of the file $D$ as follows:

$$\sigma_i = \alpha d_i + \text{PRF}(\text{k}_{\text{prf}}, i) \in \mathbb{Z}_q, \quad \text{for } 1 \leq i \leq n.$$
Algorithm SW.Store then picks a unique identifier $\text{fid}$, and terminates its execution by outsourcing to the cloud storage provider $C$ the authenticated data object:

$$\mathcal{D} := \{ \text{fid}; \{ d_i \}_{1 \leq i \leq n}, \{ \sigma_i \}_{1 \leq i \leq n} \}. $$

$\text{SW.Chall}(\text{fid}, \text{sk}) \rightarrow (\text{chal})$: This algorithm invoked by $U$ picks $l$ random elements $v_i \in \mathbb{Z}_q$ and $l$ random symbol indices $i_s$, and sends to $C$ the challenge

$$\text{chal} := \{ (i_s, v_i) \}_{1 \leq i \leq l}. $$

$\text{SW.Prove}(\mathcal{D}, \text{chal}) \rightarrow (\text{proof})$: Upon receiving the challenge $\text{chal}$, $C$ invokes this algorithm which computes the proof $\text{proof} = (\mu, \tau)$ as follows:

$$\mu = \sum_{(i, v_i) \in \text{chal}} v_i d_i, \quad \tau = \sum_{(i, v_i) \in \text{chal}} v_i \sigma_i. $$

$\text{SW.Verify}(\text{sk}, \text{proof}, \text{chal}) \rightarrow (\text{dec})$: This algorithm invoked by $U$, verifies that the following equation holds:

$$\tau = \alpha \mu + \sum_{(i, v_i) \in \text{chal}} v_i \text{PRF}(k_{\text{prf}, i_s}). $$

If proof is well formed, algorithm $\text{SW.Verify}$ outputs $\text{dec} := \text{accept}$; otherwise it returns $\text{dec} := \text{reject}$. Thanks to the unforgability of homomorphic tags, a malicious $C$ cannot corrupt outsourced data whilst eluding detection.

### Time-lock Puzzles

A time-lock puzzle is a cryptographic function that requires the execution of a predetermined number of sequential exponentiation computations before yielding its output. The RSA-based puzzle of (Rivest et al., 1996) requires the repeated squaring of a given value $\beta$ modulo $N$, where $N := p'q'$ is a publicly known RSA modulus, $p'$ and $q'$ are two safe primes\(^1\) that remain secret, and $\mathcal{T}$ is the number of squarings required to solve the puzzle, which can be adapted to the solver’s capacity of squarings modulo $N$ per second. Thereby, $\mathcal{T}$ defines the puzzle’s difficulty. Without the knowledge of the secret factors $p'$ and $q'$, there is no faster way of solving the puzzle than to begin with the value $\beta$ and perform $\mathcal{T}$ squarings sequentially. On the contrary, an entity that knows $p'$ and $q'$, can efficiently solve the puzzle by first computing the value $e := 2^\mathcal{T}(\mod \phi(N))$ and subsequently computing $\beta^e(\mod N)$.\(^2\)

PORTOS leverages the cryptographic puzzle of (Rivest et al., 1996) to build a mechanism that enables a user $U$ to increase the computational load of a misbehaving cloud storage provider $C$. To this end, $C$ is required to generate a set of pseudo-random values, called masking coefficients, which are combined with the symbols of the encoded data object $\mathcal{D}$. $C$ is expected to store at rest the masked data. More specifically, in the context of algorithm Store, $U$ outputs two functions: the function $\text{maskGen}$ which is sent to $C$ together with $\mathcal{D}$ and is used by algorithms $\text{GenR}$ and $\text{Repair}$; and the function $\text{maskGenFast}$ which is used by $U$ within the scope of algorithm $\text{Verify}$.

$\text{maskGen}((i, j), \text{param}_m) \rightarrow m_{ij}^{(j)}$: This function takes as input the indices $(i, j)$, and the tuple $\text{param}_m := (N, \mathcal{T}, \text{PRG}_{\text{mask}}, \eta_m)$ comprising the RSA modulus $N := p'q'$, the squaring coefficient $\mathcal{T}$, a pseudo-random generator $\text{PRG}_{\text{mask}} : \mathbb{Z}_N \times \{0, 1\}^* \rightarrow \mathbb{Z}_N^2$, and a seed $\eta_m \in \mathbb{Z}_N$.

Function $\text{maskGen}$ computes the masking coefficient $m_{ij}^{(j)}$ as follows:

$$m_{ij}^{(j)} := \left( \text{PRG}_{\text{mask}}(\eta_m, i \parallel j) \right)^\mathcal{T} (\mod N).$$

$\text{maskGenFast}((i, j), (p', q'), \text{param}_m) \rightarrow m_{ij}^{(j)}$: In addition to $(i, j)$ and $\text{param}_m := (N, \mathcal{T}, \text{PRG}_{\text{mask}}, \eta_m)$, this function takes as input the secret factors $(p', q')$. Knowing $p'$ and $q'$, function $\text{maskGenFast}$ efficiently computes the masking coefficient $m_{ij}^{(j)}$ by first computing the value $e$:

$$\phi(N) := (p' - 1)(q' - 1), \quad e := 2^\mathcal{T}(\mod \phi(N)), \quad m_{ij}^{(j)} := \left( \text{PRG}_{\text{mask}}(\eta_m, i \parallel j) \right)^e (\mod N).$$

The puzzle difficulty can be adapted to the computational capacity of $C$ as it evolves over time such that the evaluation of the function $\text{maskGen}$ requires a noticeable amount of time to yield $m_{ij}^{(j)}$. Furthermore, the masking coefficients are at least as large as the respective symbols of $\mathcal{D}$, hence storing the coefficients, as a method to deviate from our data reliability protocol, would demand additional storage resources which is at odds with $C$’s primary objective.

### 4.2 Protocol Specification

We now describe in detail the algorithms of PORTOS.

$\text{Setup}(1^t, T) \rightarrow (\{ S_{ij}^{(j)} \}_{1 \leq i \leq n}, \text{system})$: Algorithm $\text{Setup}$ first picks a prime number $q$, whose size

\(^1\)such that 2 is guaranteed to have a large order modulo $\phi(N)$ where $\phi(N) = (p' - 1)(q' - 1)$

\(^2\)such that its output is guaranteed to have a large order modulo $N$. 179
is chosen according to the security parameter \(\lambda\). Afterwards, given the reliability parameter \(t\), algorithm Setup yields the generator matrix \(G = [I_k | P]\) of a systematic linear \((k, n)\)-MDS code in \(\mathbb{Z}_q\), for \(k < n < q\) and \(t \leq n - k + 1\). In addition, algorithm Setup chooses \(n\) storage nodes \(\{S(j)\}_{1 \leq j \leq n}\) that are going to store the encoded data: the first \(k\) of them are data nodes that will hold the actual data symbols, whereas the rest \(n - k\) are considered as redundancy nodes.

Algorithm Setup then terminates its execution by returning the storage nodes \(\{S(j)\}_{1 \leq j \leq n}\) and the system parameters \(\text{param}_\text{system} := (k, n, G, q)\).

**U.Store** \((i^*, D, \text{param}_\text{system}) \rightarrow (sk, D, \text{param}_\text{g})\): On input security parameter \(\lambda\), file \(D \in \{0, 1\}^*\) and, system parameters \(\text{param}_\text{system}\), this randomized algorithm first splits \(D\) into \(s\) segments, each composed of \(k\) data symbols. Hence \(D\) comprises \(s \cdot k\) symbols in total. A data symbol is an element of \(\mathbb{Z}_q\) and is denoted by \(d_i^{(j)}\) for \(1 \leq i \leq s\) and \(1 \leq j \leq k\).

Algorithm U.Store also picks a pseudo-random function \(\text{PRF} : \{0, 1\}^l \times \{0, 1\}^* \rightarrow \mathbb{Z}_q\), together with its pseudo-randomly generated key \(k_{\text{prt}} \in \{0, 1\}^{l}\), and a non-zero element \(\alpha \leftarrow \mathbb{Z}_q\). Hereafter, U.Store computes for each data symbol a linearly-homomorphic MAC as follows:

\[
\sigma_i^{(j)} = ad_i^{(j)} + \text{PRF}(k_{\text{prt}}, i \parallel j) \in \mathbb{Z}_q.
\]

In addition, algorithm U.Store produces a time-lock puzzle by generating an RSA modulus \(N := p'q\), where \(p'\) and \(q\) are two randomly-chosen safe primes of size \(\lambda\) bits each, and specifies the puzzle difficulty coefficient \(T\), and the time threshold \(T_{\text{thr}}\). Thereafter, algorithm U.Store picks a pseudo-random generator \(\text{PRG}_{\text{mask}} : \mathbb{Z}_N \times \{0, 1\}^* \rightarrow \mathbb{Z}_N\), together with a random seed \(\eta_m \in \mathbb{Z}_N\), and constructs the functions maskGen and maskGenFast as described in Section 4.1

\[
\text{maskGen}((i, j), \text{param}_\text{g}) \rightarrow m_i^{(j)},
\]

\[
\text{maskGenFast}((i, j), (p', q'), \text{param}_\text{g}) \rightarrow m_i^{(j)}.
\]

Finally, algorithm U.Store picks a pseudo-random generator \(\text{PRG}_{\text{chal}} : \{0, 1\}^* \rightarrow [1, s]^3\) and a unique identifier fid.

Algorithm U.Store then terminates its execution by returning the user key

\[
\text{sk} := (\text{fid}, (\alpha, k_{\text{prt}}), (p', q', \text{maskGenFast})),
\]

such that its output is guaranteed to have a large order modulo \(N\).

---

### Table 1: Notation used in the description of PORTOS.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D)</td>
<td>File to be-outsourced</td>
</tr>
<tr>
<td>(\mathcal{S})</td>
<td>Outsourced data object ((D) consists of data and PDP tags)</td>
</tr>
<tr>
<td>(\mathcal{S})</td>
<td>Encoded and masked data object</td>
</tr>
<tr>
<td>(S)</td>
<td>Storage node</td>
</tr>
<tr>
<td>(G)</td>
<td>Generator matrix of the ((k, n))-MDS code</td>
</tr>
<tr>
<td>(t), (k_{\text{prt}})</td>
<td>Secret key used by the linearly homomorphic tags</td>
</tr>
<tr>
<td>(j)</td>
<td>Storage node index, (1 \leq j \leq n), ((\text{there are } n \text{ of these in total)})</td>
</tr>
<tr>
<td>(i)</td>
<td>Data segment index, (1 \leq i \leq k) ((D) consists of (s) segments)</td>
</tr>
<tr>
<td>(d_i^{(j)})</td>
<td>Data symbol, (1 \leq j \leq k) and (1 \leq i \leq s)</td>
</tr>
<tr>
<td>(r_i^{(j)})</td>
<td>Data symbol tag, (1 \leq j \leq k) and (1 \leq i \leq s)</td>
</tr>
<tr>
<td>(\eta_i^{(j)})</td>
<td>Redundancy symbol, (k + 1 \leq j \leq n) and (1 \leq i \leq s)</td>
</tr>
<tr>
<td>(\nu_i^{(j)})</td>
<td>Redundancy symbol tag, (k + 1 \leq j \leq n) and (1 \leq i \leq s)</td>
</tr>
<tr>
<td>(\psi_i^{(j)})</td>
<td>Redundancy symbol, (1 \leq j \leq n) and (1 \leq i \leq s)</td>
</tr>
<tr>
<td>(\eta_m)</td>
<td>Random seed used to generate (m_i^{(j)})</td>
</tr>
<tr>
<td>(p', q')</td>
<td>Primes for RSA modulus (N := p'q) of the time-lock puzzle</td>
</tr>
<tr>
<td>(T)</td>
<td>Time-lock puzzle’s difficulty coefficient</td>
</tr>
<tr>
<td>(t)</td>
<td>Size of the challenge</td>
</tr>
<tr>
<td>(T_{\text{thr}})</td>
<td>Time threshold for the proof generation</td>
</tr>
<tr>
<td>(l)</td>
<td>Indices of challenged symbols, (1 \leq j \leq n) and (1 \leq i \leq l)</td>
</tr>
<tr>
<td>(\chi_i^{(j)})</td>
<td>Random seed used to generate (\chi_i^{(j)} \in \mathbb{Z}_N)</td>
</tr>
<tr>
<td>(\chi_i^{(j)})</td>
<td>Challenge coefficients, (1 \leq i \leq s)</td>
</tr>
<tr>
<td>(\rho_i^{(j)})</td>
<td>Aggregated data/redundancy symbols, (1 \leq j \leq n)</td>
</tr>
<tr>
<td>(\tau_i^{(j)})</td>
<td>Aggregated data/redundancy tags, (1 \leq j \leq n)</td>
</tr>
<tr>
<td>(\mathcal{I}_f)</td>
<td>Set of failed storage nodes</td>
</tr>
<tr>
<td>(\mathcal{I}_f)</td>
<td>Set of surviving storage nodes</td>
</tr>
</tbody>
</table>

the to-be-outsourced data object together with the integrity tags

\[
\mathcal{D} := \{\text{fid} : (\sigma_i^{(j)})_{1 \leq i \leq s} ; (\psi_i^{(j)})_{1 \leq i \leq s}\},
\]

and the data object parameters

\[
\text{param}_\text{g} := \left(\text{PRG}_{\text{chal}}, \text{maskGen}, \right. \text{param}_\text{m} \left. : (N, T, \eta_m, \text{PRG}_{\text{mask}})\right).
\]

**C.GenR** \((D, \text{param}_\text{system}, \text{param}_\text{g}) \rightarrow (\mathcal{D})\): Upon reception of data object \(D\), algorithm C.GenR starts computing the redundancy symbols \(\{r_i^{(j)}\}_{k+1 \leq j \leq n}\) by multiplying each segment \(d_i^{(j)} := (d_i^{(1)}, \ldots, d_i^{(k)})\) with the generator matrix \(G = [I_k | P]\):

\[
d_i^{(j)} | [I_k | P] = (d_i^{(1)}, \ldots, d_i^{(k)}, [k+1, \ldots, n]).
\]

Similarly, algorithm C.GenR multiplies the vector of linearly-homomorphic tags \(\sigma_i := (\sigma_i^{(1)}, \ldots, \sigma_i^{(k)})\) with \(G\):

\[
\sigma_i | [I_k | P] = (\sigma_i^{(1)}, \ldots, \sigma_i^{(k)}, [\psi_i^{(k+1)}, \ldots, \psi_i^{(n)})).
\]

One can easily show that \(\{\psi_i^{(j)}\}_{k+1 \leq j \leq n}\) are the linearly-homomorphic authenticators of \(\{r_i^{(j)}\}_{k+1 \leq j \leq n}\).
Thereafter, algorithm C.GenR generates the masking coefficients using the function maskGen:

\[
\{m_1^{(j)}\}_{1 \leq j \leq q} := \text{maskGen}(i, j, \text{param}_m) \pmod q,
\]

and then, masks all data and redundancy symbols as follows:

\[
\begin{align*}
\{d_1^{(j)}\}_{1 \leq j \leq q} & \leftarrow \{d_1^{(j)} + m_1^{(j)}\}_{1 \leq j \leq q}, \\
\{\tilde{d}_1^{(j)}\}_{1 \leq j \leq q} & \leftarrow \{\tilde{d}_1^{(j)} + m_1^{(j)}\}_{1 \leq j \leq q}.
\end{align*}
\]

At this point, algorithm C.GenR deletes all masking coefficients \(m_1^{(j)}\) and terminates its execution by returning the encoded data object

\[
\tilde{\mathcal{D}} := \left\{ \text{fid}, \left( \{d_1^{(j)}\}_{1 \leq j \leq q}, \{\tilde{d}_1^{(j)}\}_{1 \leq j \leq q} \right) \right\},
\]

and by storing the data symbols \(d_1^{(j)}\) together with \(\sigma_1^{(j)}\) and the redundancy symbols \(\tilde{d}_1^{(j)}\) together with \(\psi_1^{(j)}\) at the corresponding storage nodes.

Algorithm C.Prove terminates its execution by returning the set of tuples:

\[
\text{proof} := \left\{ (\tilde{d}_1^{(j)}, \tilde{\tau}_1^{(j)}) \right\}_{1 \leq j \leq n^*}.
\]

U.Verify \((\text{sk}, \text{chal}, \text{proof}, \text{maskGenFast}, \text{param}_D) \rightarrow \text{(dec)}\): On input of user key \(\text{sk} := (\alpha, k_{\text{prf}}, p', q')\), challenge \(\text{chal} := (\text{fid}, \left( (\eta_1^{(j)})_{j=1}^n, (\nu_1^{(j)})_{j=1}^n \right))\), proof \(\text{proof} := \left\{ (\tilde{d}_1^{(j)}, \tilde{\tau}_1^{(j)}) \right\}_{1 \leq j \leq n^*}\), function maskGenFast, and data object parameters \(\text{param}_D := (\text{PRG}_\text{chal}, \text{maskGen}, \text{param}_m)\), this algorithm first checks if the response time of all storage nodes \(\{S_1^{(j)}\}_{1 \leq j \leq n}\) is shorter than the time threshold \(T_{\text{th}}\). If not algorithm U.Verify terminates its execution and checks that all tuples \((\tilde{d}_1^{(j)}, \tilde{\tau}_1^{(j)})\) in proof are well formed as follows:

- It first derives the indices of the requested symbols and their respective tags

\[
\begin{align*}
(i_c^{(j)})_{1 \leq j \leq n} & := \text{PRG}_\text{chal}(\eta_1^{(j)}), \quad \text{for } 1 \leq j \leq n,
\end{align*}
\]

- and it generates the corresponding masking coefficients

\[
\begin{align*}
m_1^{(j)} & := \text{maskGenFast}((i_c^{(j)}, j), (p', q'), \text{param}_D),
\end{align*}
\]

Subsequently, it computes

\[
\begin{align*}
\tilde{\tau}_1^{(j)} & := \tilde{\tau}_1^{(j)} + \alpha \cdot \sum_{c=1}^l \nu_c m_1^{(j)},
\end{align*}
\]

and then it verifies that the following equations hold

\[
\begin{align*}
\tilde{\tau}_1^{(j)} & \equiv 
\begin{cases} 
\sum_{c=1}^l \nu_c \text{PRF}(k_{\text{prf}}, i_c^{(j)} || j), & \text{if } 1 \leq j \leq k, \\
\sum_{c=1}^l \nu_c \text{prf}(j) \cdot G^{(j)} \cdot i_c^{(j)}, & \text{if } k + 1 \leq j \leq n,
\end{cases}
\end{align*}
\]

where \(G^{(j)}\) denotes the \(j\)th column of generator matrix \(G\), and \(\text{prf}^{(j)} := (\text{PRF}(k_{\text{prf}}, i_c^{(j)} || 1), \ldots, \text{PRF}(k_{\text{prf}}, i_c^{(j)} || k))\) is the vector of PRFs for segment \(i_c^{(j)}\).

If the responses from all storage nodes \(\{S_1^{(j)}\}_{1 \leq j \leq n}\) are well formed, algorithm U.Verify outputs \(\text{dec} := \text{accept}\); otherwise it returns \(\text{dec} := \text{reject}\).
Finally, by plugging Equations 1 and 3 we get

\[ \tilde{\tau}(j) = \alpha \tilde{\mu}(j) + \sum_{c=1}^{l} v_c \text{PRF}(k_{prf}, i_c(j) \parallel j). \quad \square \]

As regards to the second part of Equation 6 that defines the verification of the proofs \( \{ (\tilde{\mu}(j), \tilde{\tau}(j)) \}_{k+1 \leq j \leq n} \) generated by the redundancy storage nodes \( \{ S(j) \}_{k+1 \leq j \leq n} \), we observe that for all \( c \in [1, l] \) it holds that redundancy symbols \( r_c(j) = d_{\tilde{\mu}(j)} \cdot G_c(j) \) and tags \( \psi_{\tilde{\tau}(j)} = \sigma_{\tilde{\mu}(j)} \cdot G_c(j) \), whereby \( G_c(j) \) is the \( c \)th column of generator matrix \( G \), \( d_{\tilde{\mu}(j)} := (d_{\tilde{\mu}(j)}^{(1)}, \ldots, d_{\tilde{\mu}(j)}^{(k)}) \) is the vector of data symbols for segment \( i_c \), and \( \sigma_{\tilde{\mu}(j)} := (\sigma_{\tilde{\mu}(j)}^{(1)}, \ldots, \sigma_{\tilde{\mu}(j)}^{(k)}) \) is the corresponding vector of linearly homomorphic tags. Hence, by Equation 7 the following equality always holds:

\[
\psi_{\tilde{\tau}(j)} = (\alpha d_{\tilde{\mu}(j)} + \text{PRF}(k_{prf}, i_c(j) \parallel j)) \cdot G_c(j) = \alpha \tilde{\mu}(j) + \text{PRF}(k_{prf}, i_c(j) \parallel j),
\]

where \( \text{PRF}(k_{prf}, \cdot) \) is the vector of PRFs for segment \( i_c \). Thereby, given the same straightforward calculations as in the case of data storage nodes \( \{ S(j) \}_{1 \leq j \leq k} \), we derive the following equality:

\[ \tilde{\tau}(j) = \alpha \tilde{\mu}(j) + \sum_{c=1}^{l} v_c \text{PRF}(k_{prf}, i_c(j) \parallel j). \quad \square \]

**Req 1 : Extractability.** We now show that PORTOS ensures, with high probability, the recovery of an outsourced file \( D \). To begin with, we observe that algorithms \( C.Prove \) and \( U.Verify \) can be seen as a distributed version of the algorithms SW.Prove and SW.Verify of the private PoR scheme in (Shacham, H., and Waters, B., 2008) executed across all storage nodes \( \{ S(j) \}_{1 \leq j \leq n} \). More precisely, we assume that the MDS-code parameters \( (k, n) \) outputted by algorithm **Setup** , fulfill the retrievability requirements stated in (Shacham, H. and Waters, B., 2008), in addition to the reliability guarantee \( t \).

We argue that given a sufficient number of interactions with a cheating cloud storage provider \( C \), the user \( U \) eventually gathers linear combinations of at least \( k \leq p \leq n \) code symbols for each segment of data object \( D \). These linear combinations are of the form

\[
\tilde{\mu}(j) - \begin{cases} 
\sum_{c=1}^{l} v_c d_{\tilde{\mu}(j)}^{(c)}, & \text{for } 1 \leq j \leq k \\
\sum_{c=1}^{l} v_c h_{\tilde{\mu}(j)}^{(c)}, & \text{for } k+1 \leq j \leq n,
\end{cases}
\]
Table 2: Evaluation of the response time and the effort required by a storage node $S$ to generate its response. The cheating cloud storage provider $C'$ tries to deviate from the correct protocol execution in two ways: (i) by storing the data object $D$ encoded but unmasked; and (ii) by not storing the data object $\tilde{D}$ in its entirety. RTT is the round trip time between the user $U$ and $C'$; $\Pi$ is the number of computations $S$ can perform in parallel; and $T_{puzzle} := T' - T_{resp}$ is the time required by $S$ to generate one masking coefficient.

<table>
<thead>
<tr>
<th>Scenario:</th>
<th>Response generation complexity for one $S$</th>
<th>$S$’s Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honest $C$:</td>
<td>$2l \text{ mult } + 2(l + 1) \text{ add}$</td>
<td>$T_{resp} = \text{RTT} + \frac{m}{\Pi} T_{mult} + \frac{3l - 1}{\Pi} T_{add}$</td>
</tr>
<tr>
<td>$C'$ stores $D$ unmasked:</td>
<td>$l'T' \text{ exp } + 2l \text{ mult } + (3l - 2) \text{ add}$</td>
<td>$T'<em>{resp} = \text{RTT} + \frac{m}{\Pi} T</em>{puzzle} + \frac{3l - 1}{\Pi} T_{add}$</td>
</tr>
<tr>
<td>$C'$ deletes up to $s(n - k + 1)$ symbols from $\tilde{D}$:</td>
<td>$S$ with missing symbol: $T' \text{ exp } + 2l \text{ mult } + (2l + k - 1) \text{ add}$</td>
<td>$T'<em>{resp} = \text{RTT} \text{ mult } + \frac{3l - 1}{\Pi} T</em>{add}$</td>
</tr>
<tr>
<td></td>
<td>$k \text{ is participating in symbol generation: } T' \text{ exp } + (2l + 1) \text{ mult } + (2l - 1) \text{ add}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remaining $S$: $2l \text{ mult } + 2(l - 1) \text{ add}$</td>
<td></td>
</tr>
</tbody>
</table>

for known coefficients $(\nu_i)_{i=1}^l$, and known indices $\delta_i^{(j)}$ and $\eta_i^{(j)}$. Furthermore, $U$ can efficiently derive the unmasked expressions

$$
\mu^{(j)} \leftarrow \sum_{i=1}^l \nu_i \delta_i^{(j)}, \text{ for } 1 \leq j \leq k
$$

and

$$
\tau^{(j)} \leftarrow \sum_{i=1}^l \nu_i \eta_i^{(j)}, \text{ for } k + 1 \leq j \leq n
$$

by computing the masking coefficients $\{m_i^{(j)}\}_{i \geq j \geq l}$ using the function $\text{maskGenFast}$, and subtracting from $\bar{\mu}^{(j)}$ the corresponding linear combination $\sum_{i=1}^l \nu_i m_i^{(j)}$.

Hereby, the extractability arguments given in (Shacham, H. and Waters, B., 2008) can be applied to the aggregated output of algorithms $C$.Prove and $U$.Verify. More precisely, given that $C'$ succeeds in making algorithm $U$.Verify yield desc := accept in an $\varepsilon$ fraction of the interactions, and the indices $\hat{i}_i^{(j)}$ of the challenge a chosen at random, then user $U$ has at its disposal at least $\rho - \varepsilon > k$ correct code symbols for each segment of data object $D$. Therefore, user $U$ is able to reconstruct the data object $\tilde{D}$ using the parity check matrix $H = [-P^T | I_{n-k}]$. □

**Req 2 : Soundness of Redundancy Generation.** In PORTOS, the cloud storage provider $C$ is the party that generates the redundancy of the outsourced data object $D$. Namely, the redundancy symbols and $\left\{\xi_i^{(j)}\right\}_{i \geq j \geq l}$ and their tags $\left\{\psi_i^{(j)}\right\}_{i \geq j \geq l}$ are computed by applying a linear combination over the original data symbols $\left\{d_i^{(j)}\right\}_{i \geq j \geq l}$ and their tags $\left\{\sigma_i^{(j)}\right\}_{i \geq j \geq l}$, respectively. Hence, following Theorem 4.1 in (Shacham, H. and Waters, B., 2008), if the pseudo-random function $\text{PRF}$ is secure, then no cheating cloud storage provider $C'$ will cause a verifier $V$ to accept in a proof of data reliability instance, except by responding with values

$$
\bar{\mu}^{(j)} \leftarrow \sum_{i=1}^l \nu_i \delta_i^{(j)}, \text{ for } k + 1 \leq j \leq n
$$

and

$$
\tau^{(j)} \leftarrow \sum_{i=1}^l \nu_i \eta_i^{(j)}, \text{ for } k + 1 \leq j \leq n
$$

that are computed correctly; i.e., by computing the pair $(\bar{\mu}, \tau)$ using values $\bar{\delta}_i^{(j)}$ and $\bar{\eta}_i^{(j)}$ which are the output of algorithm $C$.GenR. □

**Req 3 : Storage Allocation Commitment.** We now show that a rational cheating cloud storage provider $C'$ cannot produce a valid proof of data reliability as long as the time threshold $T_{thr}$ is tuned properly.

In essence, PORTOS consists of parallel proof of data possession challenges over all storage nodes $\left\{S_i^{(j)}\right\}_{i \geq j \geq l}$: a challenge for each symbol of the codeword. It follows that when a proof of data reliability challenge contains symbols which are not stored at rest, the relevant storage nodes cannot generate their part of the proof unless $C'$ is able to generate the missing symbols. Hereafter, we analyze the effort that $C'$ has to put in order to output a valid proof of data reliability in comparison to the effort an honest cloud storage provider $C$ has to put in order to output the same proof. Given that the computational effort required by $C$ and $C'$ can be translated into their response time $T_{resp}$ and $T'_{resp}$, we can determine the lower and upper bounds for the time threshold $T_{thr}$.

A fundamental design feature of PORTOS is that $C'$ has to compute a masking coefficient for each symbol of the encoded data object $\tilde{D}$. We observe that the
masking coefficients have the same size as $\mathcal{D}$’s symbols. Hence, assuming that $\mathcal{D}$ cannot be compressed more (e.g., because it has been encrypted by the user), a strategy whereby $C'$ is storing the masking coefficients would effectively double the required storage space. Moreover, a strategy whereby $C'$ does not store the content of up to $n - k + 1$ storage nodes and yet it stores the corresponding masking coefficients, would increase $C'$’s operational cost without yielding any storage savings. Given that strategies that rely on storing the masking coefficients do not yield any gains in terms of either storage savings or overall operational cost, $C'$ is left with two reasonable ways to deviate from the correct protocol execution:

(i) The first one is to store the data object $\mathcal{D}$ encoded but unmasked. Although this approach does not offer any storage savings, it significantly reduces the complexity of storing and maintaining $\mathcal{D}$ at the cost of a more expensive proof generation. More specifically, in order to compute a PORTOS proof, $C'$ has to generate $2l$ masking coefficients $\{m^{(i)}_{\pi(i)}\}_{1 \leq i \leq \Pi}$.

(ii) The second way $C'$ may misbehave, is by not storing the data object $\mathcal{D}$ in its entirety and hence generating the missing symbols involved in a PORTOS challenge on-the-fly. In particular, $C'$ can drop up to $s(n - k + 1)$ symbols of $\mathcal{D}$ either by not provisioning up to $n - k + 1$ storage nodes; or by uniformly dropping symbols from all $n$ storage nodes $\{S^{(j)}\}_{1 \leq j \leq n}$, ensuring that it preserves at least $k$ symbols for each data segment.

In order to determine the lower bound for the time threshold $T_{\text{thr}}$, we evaluate the response time $T_{\text{resp}}$ of an honest cloud storage provider $C$. Additionally, for each type of $C'$’s malicious behavior we evaluate its response time, and determine the upper bound for the time threshold $T_{\text{thr}}$ as $T_{\text{resp}} = \min(T_{\text{resp}}', T_{\text{resp}}'')$, where $T_{\text{resp}}'$ and $T_{\text{resp}}''$ denote $C'$’s response time when it opts to keep $\mathcal{D}$ unmasked and delete symbols of $\mathcal{D}$, respectively. Concerning the evaluation of $T_{\text{resp}}''$, we consider the most favorable scenario for $C'$ where it has to generate only one missing symbol for a PORTOS challenge. Table 2 presents the effort required by a storage node $S$ in order to output its response, for each of the scenarios described above, together with the corresponding response time.

For the purposes of our analysis, we assume that all storage nodes $\{S^{(j)}\}_{1 \leq j \leq n}$ have a bounded capacity of $\Pi$ concurrent threads of execution, that computations—exponentiations, multiplications, additions, etc.—require a minimum execution time, and that $T_{\text{add}} \ll T_{\text{exp}}$ and $T_{\text{add}} \ll T_{\text{mult}}$. Furthermore, we assume that $\{S^{(j)}\}_{1 \leq j \leq n}$ are connected with premium network connections (low latency and high bandwidth), and hence the communication among them has negligible impact on $C'$’s response time. As given in Table 2, Req 3 is met as long as the time threshold $T_{\text{thr}}$ is tuned such that it fulfills the following relations:

\[ T_{\text{thr}} > \frac{\text{RTT}_{\max}}{\Pi} + \frac{2l}{\Pi} T_{\text{mult}} \text{ (Lower bound)}, \]

\[ T_{\text{thr}} < \frac{\text{RTT}_{\max}}{\Pi} + T_{\text{puzzle}} + \frac{2l + 1}{\Pi} T_{\text{mult}} \text{ (Upper bound)}, \]

where $\text{RTT}_{\max}$ is the worst-case RTT and $T_{\text{puzzle}} := T_{\text{exp}}$. $T_{\text{exp}}$ is the time required by $S$ to evaluate the function $\text{maskGen}$. By carefully setting the puzzle difficulty coefficient $T'$, we can guarantee that $T_{\text{resp}} < T_{\text{resp}}'$ and $\text{RTT}_{\max} \ll T_{\text{resp}}' - T_{\text{resp}}$, and hence make our proof of data reliability scheme robust against network jitter. Finally, notice that PORTOS can adapt to $C'$’s computational capacity as it evolves over time by tuning $T'$ accordingly.

## 6 PERFORMANCE ANALYSIS

Table 3 summarizes the performance of PORTOS in terms of computation, communication, and storage costs. The computational effort required to verify the response of a redundancy node $\{S^{(j)}\}_{k+1 \leq j \leq n}$ is much higher than the effort required to verify the response of a data node $\{S^{(j)}\}_{1 \leq j \leq k}$. Namely, in addition to the $2nl$ exponentiations required by the function $\text{maskGenFast}$ to generate the masking coefficients of the $nl$ involved symbols, algorithm $U.\text{Verify}$ evaluates $kl$ PRFs and $k + 2(l + 1)$ multiplications in

---

### Table 3: PORTOS computation, communication, and storage costs.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>PORTOS with Symbol Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.Store complexity:</td>
<td>$sk \text{ PRF} + sk \text{ mult} + sk \text{ add}$</td>
</tr>
<tr>
<td>C.Prove complexity:</td>
<td>$n \text{ PRG}_{\text{ch}} + 2nl \text{ mult} + 2(nl + 1) \text{ add}$</td>
</tr>
<tr>
<td>U.Verify complexity:</td>
<td>$n \text{ PRG}_{\text{ch}} + 2nl \text{ exp} + kl(n - k + 1) \text{ PRF}$ $+ 2nl + 1 + kl(n - k) \text{ mult} + (n - k)(kl + k + 2)$ add</td>
</tr>
<tr>
<td>Storage cost:</td>
<td>$2 \times \text{ the size of } \mathcal{D}$</td>
</tr>
<tr>
<td>Bandwidth:</td>
<td>$2n \text{ symbols}$</td>
</tr>
</tbody>
</table>
order to verify the response of each redundancy node, compared to the $l$ PRFs and $2(l + 1)$ multiplications it has to compute for the response of each data node. We conclude that compared to existing erasure-code based proof of data reliability schemes, namely (Bowers et al., 2009; Chen et al., 2015; Vasilopoulos et al., 2018), we achieve comparable computational gain while enabling data repair at the cloud side. Nevertheless, we observe that storage and bandwidth costs remain important. In order to improve the performance of our scheme and reduce these costs, we propose a new version of PORTOS which namely implements the storage efficient variant of the linearly homomorphic tags introduced in (Shacham and Waters, B., 2008). More specifically, instead of generating one tag per symbol, the new algorithm U.Store computes a linearly homomorphic tag for a data segment, comprising $k$ symbols. Due to space constraints, we omit the description of this new solution which will be included in an extended technical report, that will become available after the review process.

7 CONCLUSION

In this paper, we proposed PORTOS, a novel proof of data reliability solution for erasure-code-based distributed cloud storage systems. PORTOS enables users to verify the retrievability of their data, as well as the integrity of its respective redundancy. Moreover, in PORTOS the cloud storage provider generates the required redundancy and performs data repair operations without any interaction with the user, thus conforming to the current cloud model. Thanks to the combination of PDP with time-lock puzzles, PORTOS provides a rational cloud storage provider with a strong incentive to provision sufficient redundancy, which is stored at rest, guaranteeing this way a reliable storage service.

REFERENCES


