Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE ARTICLE

Proofs of Data Reliability: Verification of Reliable Data Storage with Automatic Maintenance[†]

Dimitrios Vasilopoulos*[1] | Melek Önen[1] | Refik Molva[1] | Kaoutar Elkhiyaoui**[2]

[1]EURECOM, Sophia Antipolis, France
[2]IBM Research Zurich, Zurich, Switzerland

Correspondence
*Dimitrios Vasilopoulos.
Email: dimitrios.vasilopoulos@eurecom.fr

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 two novel Proof of Data Reliability schemes, namely POROS and 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. As a result, POROS and PORTOS are compatible with the current cloud computing model where the cloud autonomously performs all maintenance operations without any interaction with the user. Moreover, POROS and PORTOS delegate the burden of generating the redundancy to the cloud as well. The security of both solutions 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.

KEYWORDS:
Secure cloud storage, proofs of reliability, reliable storage, verifiable storage.

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 preliminary version of this work was published in SCC '18[1] and SECRYPT '19[2].
**This work was done while the author was affiliated with EURECOM.

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[3,4] (PoR) and Proofs Data Possession[5] (PDP) are cryptographic mechanisms that enable a cloud user to efficiently verify that her data is correctly stored by an untrusted cloud storage provider. Yet, PoR and PDP schemes are of limited value when it comes to the audit of reliable data storage mechanisms: indeed, a successful PoR/PDP verification does not indicate whether a cloud provider has in place reliable data storage mechanisms, whereas an unsuccessful one attests the irreversible damage of the outsourced data. Detecting that an outsourced file is corrupted is of little help for a client because the latter is no longer retrievable.

Furthermore, in the adversarial setting of outsourced storage, there seems to be an inherent conflict between the customers' requirement for verifying reliable data storage mechanisms and a key feature of modern storage systems, namely, *automatic maintenance*. On the one hand, automatic maintenance based on either replication or erasure codes requires the storage of redundant information along with the data object and, on the other hand, to guarantee the storage of redundancy the cloud storage provider should not have access to the content of the data since the redundancy information is a function of the original data itself. Hence, the root cause of the conflict between verification of reliable data storage and automatic maintenance stems from the fact that the redundancy is a function of the original data object itself. A malicious storage provider can exploit this property in order to save storage space by keeping only the outsourced data, without the required redundancy, and leverage the data repair mechanism when needed in order to positively meet the verification of reliable data storage criteria.

PoRs and PDPs have been extended in order to ensure data reliability over time introducing the notion of *Proofs of Data Reliability*[6,7,8,9,10]. 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. 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. Similarly, 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. As a result, most Proof of Data Reliability schemes[6,7,8,9] require some interaction with the user to repair damaged data. However, such an approach is at odds with the current cloud system model as (i) it transfers the redundancy generation and data repair responsibilities to the user, and (ii) it obstructs the automatic maintenance mechanisms.

Main objective. Informally, our main objective is to design a Proof of Data Reliability scheme tailored to distributed cloud storage systems that makes use of erasure codes to guarantee data reliability and achieves the following properties against a rational cloud storage provider:

- *Data reliability against t storage node failures:* The protocol scheme 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:* In addition to the integrity of the data outsourced to a cloud storage system, the protocol should assure the users that the cloud storage provider stores sufficient amount of redundancy information along with original data segments to be able to guarantee the maintenance of the data in the face of corruption or loss.

- *Automatic data maintenance by the cloud storage provider:* The cloud storage provider should have 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.

- *Real-world cloud storage architecture:* The protocol should conform to the current model of erasure-code-based distributed storage systems. Moreover, it should not make any assumption regarding the system's underlying technology.

Contributions. In this paper, we propose two novel Proof of Data Reliability schemes for erasure-code-based distributed storage systems, namely, POROS and PORTOS. We followed an incremental approach during the design of POROS and PORTOS, hence even though they share the same overall goal, they differ in several aspects. To begin with, both schemes enable automatic maintenance by the cloud storage provider, and they guarantee that a *rational* cloud storage provider does not compute the redundancy information on demand upon a Proof of Data Reliability request but instead stores it at rest. However, POROS assumes a specific storage model that allows for the detection of a cheating cloud storage provider; whereas PORTOS relies on time-locked puzzles in order to augment the recourses a cheating cloud storage provider has to provision in order to produce a valid Proof of Data Reliability. As a result, PORTOS design conforms to the current cloud storage model at the cost of a less efficient data repair procedure compared to POROS.

More specifically, our paper makes the following contributions:

- We introduce the concept of Proofs of Data Reliability, a new family of Proofs of Storage that resolves the conflict between reliable data storage verification and automatic maintenance. We provide the formal definition and security requirements of a Proof of Data Reliability scheme, which is more generic than the definitions presented in prior work[10].
- We propose POROS, a Proof of Data Reliability scheme, that on the one hand, enables a user to efficiently verify the correct storage of her outsourced data and its reliable data storage; and on the other hand, allows the cloud to perform automatic maintenance operations. POROS guarantees that a *rational* cloud storage provider would not compute redundancy information on demand upon Proof of Data Reliability requests but instead would store it at rest. As a result of bestowing the cloud storage provider with the repair function, POROS allows for the automatic maintenance of data by the storage provider without any interaction with the customers. We analyze the security of POROS both theoretically and through experiments measuring the time difference between an honest cloud and some malicious adversaries.
- We propose PORTOS, a Proof of Data Reliability scheme, that enables the verification of reliable data storage without disrupting the automatic maintenance operations performed by cloud storage provider. PORTOS's design allows for the fulfillment of the same Proof of Data Reliability requirements as POROS, while overcoming the shortcoming of POROS. PORTOS design does not make any assumptions regarding the underlying storage medium technology nor it deviates from the current distributed storage architecture. 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. Moreover, we analyze the performance of PORTOS in terms of storage, communication, and verification cost.
- We propose a more efficient version of PORTOS which improves the performance of both the cloud storage provider and the verifier at the cost of reduced granularity with respect to the detection of corrupted storage nodes.

Organization. The remaining of this paper is organized as follows: In Section 2, we give an overview of prior work in the field. We provide the formal definition, adversary model, and security requirements of a Proof of Data Reliability scheme in Section 3. In Section 4, we introduce POROS, a novel proof data reliability scheme. We provide its specification, and we analyze its security. In Section 5, we provide the specification of PORTOS, the analysis of its security, and the evaluation of its performance. Thereafter, we describe a more efficient version of the protocol in Section 5.6 and we conclude in Section 6.

2 | PRIOR WORK

Replication-based Proofs of Data Reliability. Curtmola et al.[6] proposed a multi-replica Proof of Data Possession protocol (MR-PDP), a Proof of Data Reliability scheme where a client outsources to an untrusted cloud storage provider t distinct replicas of a file, each of which can be used for the recovery of the original file. To force the cloud storage provider to store all t replicas in their entirety, each replica is masked with some randomness generated by a pseudo-random function (PRF). Hence each replica is unique and differentiable from the others. The scheme makes use of a modified version of the RSA-based homomorphic verification tags proposed by Ateniese et al.[5] so that a single set of tags –generated for the original file– can be used to verify any of the t replicas of that file. This solution suffers from the weakness that the verifier cannot discern whether a proof generated by the cloud storage provider was computed using the actual replica the challenge was issued for or not.

Barsoum and Hassan[11] proposed a multi-replica PDP scheme for static files. The user generates t replicas of a file by encrypting it using t different encryption keys and then generates one aggregated homomorphic verification tag for all the replica symbols that share the same index. The verifier issues to a PDP challenge for a random subset of the replicas' symbols and the cloud storage provider generates a proof which shows that all t replicas are correctly stored. This solution suffers from the weakness that multi-replica is not necessarily equivalent to multi-storage node: indeed a single storage node may keep all the replicas, and thus not fulfilling the reliability guarantee t. Later on, Barsoum and Hassan[12,13] propose a multi-replica dynamic PDP scheme that enables clients to update/insert/delete selected data blocks and to verify the integrity of multiple replicas of their outsourced files. In addition to the creation of the replicas and the generation of the homomorphic verification tag as in Barsoum and Hassan[11], the user constructs a Merkle hash tree for each replica and subsequently the roots of the t Merkle hash trees are used to construct another Merkle hash tree called the directory. The user uploads to the cloud storage provider the t replicas together with the homomorphic verification tags and the Merkle hash trees and keeps in local storage the root of the directory tree and the keying material. The verification of storage is similar to the one in Barsoum and Hassan[11] with the addition of the use of the Merkle hash trees to verify that the proof is computes over the updated data.

Etemad and Küpçü[14] proposed a dynamic Proof of Data Possession (DR-DPDP) scheme that supports transparent distribution and replication in cloud storage systems. This scheme extends the dynamic PDP scheme proposed by Erway et al.[15] which makes use of rank-based authenticated skip list: a data structure that enables efficient insertions and deletions of data segments by the user, while being able to verify updates. The scheme introduces a new entity, namely, the organizer: a gateway of the cloud storage provider that is responsible for all communication with the client and the storage nodes. The user processes the file she wishes to outsource as in Erway et al.[15] and sends it to the organizer. In turn, the organizer divides the file and its corresponding skip list into partitions each one with its own rank-based authenticated skip list and distributes each the partitions to an agreed-upon number of storage nodes. Each server stores the blocks, builds the corresponding part of the rank-based authenticated skip list, and sends the root value back to the organizer who constructs its own part of the rank-based authenticated skip list and sends the root to the client. This solution also suffers from the weakness that the verifier has no way of distinguishing whether the file is stored on a single node or multiple storage nodes due to the presence of the organizer. The verifier is only ensured that at least one working copy of the file is present.

Chen and Curtmola[16,7], elaborate on the idea of MR-PDP and propose two Proof of data reliability schemes that enable server-side repair operations. The user prepares t replicas each one comprising a masked version of the outsourced file and its own set of RSA-based homomorphic verification tags. Both schemes make use of a tunable masking mechanism in order to produce distinct replicas. In the first scheme, each symbol of the file is masked with a random value generated by η evaluation of a pseudo-random function (PRF), where η depends on the computational capacity of the cloud storage provider. The second scheme constructs each replica by putting the outsourced file through a butterfly network[17]: a cryptographic transformation that results in each symbol of the replica depending on a large set of symbols of the original file. In order to verify that the cloud storage provider correctly stores all replicas, the verifier sends a challenge for all t of them. When the verifier detects a corrupted replica, it acts as a repair coordinator and instructs the cloud storage provider to recover the original file by removing the masking from one of the healthy replicas and thereafter constructing a new one. Leontiadis and Curtmola[18] extend the scheme in Chen and Curtmola[7] in order to propose a Proof of Data Reliability scheme that is compatible with file-level deduplication. The scheme enable the cloud storage provider to keep a single copy of a file's replicas uploaded by different users however it does not allow the deduplication of the homomorphic verification tags.

Armknecht et. al[10] propose a multi-replica Proof of Retrievability scheme that delegates the replica construction to the cloud storage provider. Similarly to Chen and Curtmola[16], the replicas are masked: the scheme uses tunable multiplicative homomorphic puzzles and linear feedback shift registers to build a replication mechanism that requires the cloud storage provider to dedicate significant computational resources in order to produce a replica. This way the verifier is able to detect a cheating cloud storage provider who does not store all replicas at rest and attempts to reconstruct the missing replicas on-the-fly.

Erasure-code-based Proofs of Data Reliability. Bowers et al.[8] propose HAIL: an availability and integrity layer for distributed cloud storage. HAIL uses pseudo-random functions, error-correcting codes, and universal hash functions to construct an integrity-protected error-correcting code that produces parity symbols which are at the same time verification tags of the underlying data segment. HAIL disperses the n symbols of a codeword across n storage nodes and it further applies a second layer of error-correcting code over the symbols of each storage node. This code protects against the low-level corruption of file blocks that may occur when integrity checks fail. Before uploading a file the user divides it into equally-sized segments and apply the two layers of error-correcting codes. In HAIL time is divided to epochs during which the adversary can corrupt some storage nodes (less than the maximum number of symbols the code can repair). At the end of each epoch, the verifier issues a challenge involving a random subset of the encoded segments. Upon the detection of data corruption in a storage node the verifier retrieves the necessary data and parity symbols from the remaining healthy storage nodes and reconstructs the content of the failed storage node. Later on, Chen et al.[9] redesign parts of HAIL in order to achieve a more efficient repair phase that shifts the bulk computations to the cloud side. The new scheme uses an erasure code with generator matrix G which remains secret from the cloud storage provider. At the end of each epoch, if data corruption is detected one healthy storage node is assigned with the task to reconstruct the lost symbols. More precisely, the verifier derives a set of intermediate coefficients from the generator matrix G which subsequently masks using an algebraic function and sends them to the storage node that will perform the repair operation. Thanks to the algebraic properties of both the erasure code and the masking function, the storage node is able to reconstruct the corrupted symbols without the knowledge of the generator matrix G.

Chen et al.[19] present a Proof of Data Reliability scheme that relies on network coding. Compared to erasure codes, network codes offer optimally minimum communication cost during the reconstruction of lost symbols at the cost of not exact repair: the new symbols are functionally equivalent but not the same as the lost ones. The user divides the to-be-outsourced file into equally-sized segments and encodes it. Thereafter the user computes two types of verification tags: one linearly-homomorphic "challenge" tag for each symbol in a segment that is used for the integrity verification of the file; and on "repair" tag for each segment that indicates the symbols of the original file the segment depends on. The scheme adopts the adversary model of Bowers et al.[8] where the adversary corrupts a number of storage nodes within an epoch, at the end of which the verifier computes new symbols in the place of the corrupted ones. Le and Markopoulou[20] extends the scheme in Chen et al.[19] by leveraging a more efficient homomorphic tag scheme called SpaceMac[21] and a customized encryption scheme that exploits random linear combinations. These changes reduce the computation cost of the repair mechanism for the client and make possible the verification of the data integrity by a third party auditor. Based on the introduction of this new entity, Thao and Omote[22] 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 user.

Bowers et al.[23] 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: specifically, RAFT relies on the fact that the time required for n parallel reeds from n rotational hard drives is significantly less from the time required for the same number of reads – some of which become sequential in this case – from less than n drives. The scheme defines a mapping between encoded symbols and the storage devices that is used by the verifier to issue a challenge comprising $l \geq n - t$ parallel symbol access from l distinct storage devices. This way, the verifier can detect whether users' encoded data are stored at multiple hard drives within the same data center in such a manner that they can be recovered in the face of t hard drive failures. The symbols composing the proof are not aggregated likewise in a typical PoS scheme hence, RAFT incurs high bandwidth consumption for the reliable data storage verification. Moreover, the basic RAFT[23] protocol requires that the verifier keeps a local copy of the outsourced file in order to determine whether a proof is valid or not. Besides, one can devise a more sophisticated RAFT scheme that relies on verification tags and therefore does not require a local copy of the data at the verifier.

Conclusions on the state of the art. From the review of existing work, we are able to draw some conclusions that guide our own research in the field of Proofs of Data Reliability.

- Most of the proof of data reliability schemes presented above share a common system model where the user generates locally the required redundancy, 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 user.
- Proof of Data Reliability schemes that allow for automatic[10,23] or "semi-automatic" maintenance by the cloud storage provider[7,18] set a time threshold $\mathsf{T}_{\mathsf{thr}}$ in order to decide whether to accept or reject a proof produced by cloud storage provider C. This time threshold is defined as a function of C's computational capacity.
- There exists no solution for an erasure-code-based Proof of Data Reliability scheme that allows for automatic maintenance by cloud storage provider. Even though the scheme in Bowers et al.[23] enables the cloud storage provider to autonomously repair corrupted codeword symbols, its adversarial model does not guarantee the extractability of an outsourced file.

3 | PROOFS OF DATA RELIABILITY

3.1 | Environment

A Proof of Data Reliability scheme comprises the three following entities:

User U. User U wishes to outsource her file D to a cloud storage provider C.

Cloud storage provider C. Cloud storage provider C commits to store file D in its entirety together with sufficient redundancy generated by its reliable data storage mechanisms. Cloud storage provider C is considered as a potentially *malicious* party.

Verifier V. Verifier V interacts with cloud storage provider C in the context of a challenge-response protocol and validates whether C is storing file D in its entirety.

We consider a setting where user U uploads a file D to cloud storage provider C and thereafter, verifier V periodically queries C for some proofs on the integrity and reliable data storage of D. In reality, user U produces a verifiable data object $\mathcal{D}$ from file D that contains some additional information that will further help for the verification of its reliable storage. If the data reliability scheme is not compatible with automatic maintenance, $\mathcal{D}$ also incorporates the required redundancy for the reliable storage of file D. At this point, user U uploads the data object $\mathcal{D}$ to cloud storage provider C and deletes both file D and data object $\mathcal{D}$ retaining in local storage only the key material she used during the generation of $\mathcal{D}$. In turn, cloud storage provider stores $\mathcal{D}$ across a set of n storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ with *reliability guarantee* t: some storage service guarantee against t storage node failures.

We define a *Proof of Data Reliability* scheme as a protocol executed between user U and verifier V on the one hand and cloud storage provider C with its storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ on the other hand. The aim of such a protocol is to enable the verifier V to check (i) the *integrity* of $\mathcal{D}$ and, (ii) whether the *reliability guarantee* t is satisfied. Verifier V issues a Proof of Data Reliability challenge, which refers to a subset of the data and redundancy symbols, and send it to the cloud storage provider C. Henceforth, C generates a proof and V checks whether this proof is well formed and accepts it or rejects it accordingly. 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 $\mathcal{D}$ from $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$.

3.2 | Formal Definition

We now give the formal definition and security requirements of a Proof of Data Reliability scheme.

Definition 1. (Proof of Data Reliability scheme). A Proof of Data Reliability scheme is defined by seven polynomial time algorithms:

$\mathsf{Setup}\ (1^\lambda, t) \rightarrow (\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}, \mathsf{param}_{\mathsf{system}})$**:** This algorithm takes as input the security parameter λ and the reliability parameter t, and returns the set of storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$, the system parameters $\mathsf{param}_{\mathsf{system}}$, and the specification of the redundancy mechanism: the number of replicas or the erasure code scheme that will be used to generate the redundancy.

$\mathsf{Store}\ (1^\lambda, D) \rightarrow (K_{\mathsf{u}}, K_{\mathsf{v}}, \mathcal{D}, \mathsf{param}_{\mathcal{D}})$**:** This randomized algorithm invoked by user U takes as input the security parameter λ and the to-be-outsourced file D, and outputs the user key K_{u}, the verifier key K_{v}, and the verifiable data object $\mathcal{D}$, which also includes a unique identifier fid, and a set of data object parameters $\mathsf{param}_{\mathcal{D}}$.

$\mathsf{GenR}\ (\mathcal{D}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\tilde{\mathcal{D}})$**:** This algorithm takes as input the verifiable data object $\mathcal{D}$, the system parameters $\mathsf{param}_{\mathsf{system}}$, and optionally, the data object parameters $\mathsf{param}_{\mathcal{D}}$, and outputs the data object $\tilde{\mathcal{D}}$. Algorithm GenR may be invoked either by 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, $\tilde{\mathcal{D}}$ may comprise multiple copies of $\mathcal{D}$ or an encoded version of it. Additionally, algorithm GenR generates the necessary integrity values that will further help for the integrity verification of $\tilde{\mathcal{D}}$'s redundancy.

$\mathsf{Chall}\ (K_{\mathsf{v}}, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{chal})$**:** This stateful and probabilistic algorithm invoked by verifier V takes as input the verifier key K_{v} and the system parameters $\mathsf{param}_{\mathsf{system}}$, and outputs a challenge chal.

$\mathsf{Prove}\ (\mathsf{chal}, \tilde{\mathcal{D}}) \rightarrow (\mathsf{proof})$**:** This algorithm invoked by C takes as input the challenge chal and the data object $\tilde{\mathcal{D}}$, and returns C's proof.

$\mathsf{Verify}\ (K_{\mathsf{v}}, \mathsf{chal}, \mathsf{proof}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{dec})$**:** 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 $\mathsf{param}_{\mathcal{D}}$, and outputs a decision $\mathsf{dec} \in \{\mathsf{accept}, \mathsf{reject}\}$ indicating a successful or failed verification of the proof, respectively.

$\mathsf{Repair}\ ({}^*\tilde{\mathcal{D}}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\tilde{\mathcal{D}}, \perp)$**:** This algorithm takes as input a corrupted data object ${}^*\tilde{\mathcal{D}}$ together with its parameters $\mathsf{param}_{\mathcal{D}}$ and the system parameters $\mathsf{param}_{\mathsf{system}}$, and either reconstructs $\tilde{\mathcal{D}}$ or outputs a failure symbol $\perp$. Algorithm Repair may be invoked either by U or C depending on the Proof of Data Reliability scheme.

A Proof of Data Reliability scheme should be *correct* and *sound*.

3.3 | Correctness

A Proof of Data Reliability scheme should be correct: if both cloud storage provider C and verifier V are *honest*, then on input chal sent by the verifier V, using algorithm Chall, algorithm Prove (invoked by C) generates a Proof of Data Reliability proof such that algorithm Verify yields accept with probability 1.

Definition 2. **(*Req* 0 : Correctness).** A Proof of Data Reliability scheme (Setup, Store, GenR, Chall, Prove, Verify, Repair) is **correct** if for all honest cloud storage providers C and verifier V, and for all verifier keys $K_{\mathsf{v}} \leftarrow \mathsf{Store}\ (1^\lambda, D)$, all files D with file identifiers fid and for all challenges $\mathsf{chal} \leftarrow \mathsf{Chall}\ (\mathcal{K}, \mathsf{param}_{\mathsf{system}})$:

$$\Pr[\mathsf{Verify}(K_{\mathsf{v}}, \mathsf{chal}, \mathsf{proof}, \mathsf{param}_{\mathcal{D}}) \rightarrow \mathsf{accept} \mid \mathsf{proof} \leftarrow \mathsf{Prove}(\mathsf{chal}, \mathcal{D})] = 1$$

3.4 | Soundness

A Proof of Data Reliability scheme is *sound* if it fulfills three security requirements, namely, *extractability*, *soundness of redundancy generation*, and *storage allocation commitment*. The extractability requirement protects user U against cloud storage provider C that does not store that data object $\mathcal{D}$ in its entirety. The soundness of redundancy generation requirement ensures user U that reliable data storage mechanisms correctly generate $\mathcal{D}$'s redundancy. Finally, the storage allocation commitment requirement protects a user U against a cloud storage provider C that does not allocate sufficient storage space to store the whole redundancy. We now give the formal definition for each of the requirements.

3.4.1 | 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. We capture this requirement using the notion of *extractability* as introduced in Juels and Kaliski[3], and Shacham and Waters[4]. Extractability guarantees that if a cloud storage provider C can convince an honest verifier V with high probability that it is storing the data object $\mathcal{D}$ together with its respective redundancy, then there exists an extractor algorithm $\mathsf{Extract}$ that given sufficient interaction with C, can extract the file D. To define this requirement, we adapt the retrievability game proposed by Shacham and Waters[4] between an adversary $\mathcal{A}$ and an environment to the Proof of Data Reliability definition. The environment simulates all honest users and an honest verifier. Furthermore, the environment provides $\mathcal{A}$ with oracles for the algorithms Encode, Chall and Verify:

$\mathcal{O}_{\mathsf{Encode}}$**:** On input of a file D and a user key K_{u}, this oracle returns a file identifier fid and a verifiable data object $\mathcal{D}$ that will be outsourced by $\mathcal{A}$.

$\mathcal{O}_{\mathsf{Chall}}$**:** On input of verifier key K_{v} and a file identifier fid, this oracle returns a query chal to adversary $\mathcal{A}$.

$\mathcal{O}_{\mathsf{Verify}}$**:** On input of verifier key K_{v}, file identifier fid, challenge chal and proof proof, this oracle returns $\mathsf{dec} = \mathsf{accept}$ if proof is a valid proof and $\mathsf{dec} = \mathsf{reject}$ otherwise.

Thereafter, we consider the following game between the adversary and the environment:

1. $\mathcal{A}$ interacts with the environment and asks for an honest user U.
2. $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Store}}$ providing, for each query, some file D. The oracle returns to $\mathcal{A}$ the tuple $(K_{\mathsf{u}}, \mathcal{D}, \mathsf{param}_{\mathcal{D}}) \leftarrow \mathcal{O}_{\mathsf{Store}}$ $(1^\lambda, D)$, for a given user key K_{u}.
3. For any data object $\mathcal{D}$ with file identifier fid of a file D it has previously sent to $\mathcal{O}_{\mathsf{Store}}$, $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Chall}}$ to generate a random challenge $(\mathsf{chal}) \leftarrow \mathcal{O}_{\mathsf{Chall}}\ (K_{\mathsf{v}}, \mathsf{param}_{\mathsf{system}})$, for a given verifier key K_{v}.
4. Upon receiving challenge chal, $\mathcal{A}$ generates a proof proof either by invoking algorithm Prove or at random.
5. $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Verify}}$ to check the proof proof and gets the decision $(\mathsf{dec}) \leftarrow \mathcal{O}_{\mathsf{Verify}}\ (K_{\mathsf{v}}, \mathsf{chal}, \mathsf{proof}, \mathsf{param}_{\mathcal{D}})$, for a given verifier key K_{v}.

   Steps $1 - 5$ can be arbitrarily interleaved for a polynomial number of times.

6. Finally, $\mathcal{A}$ picks a file D^* that was sent to $\mathcal{O}_{\mathsf{Store}}$ together with the corresponding user U, and outputs a simulation of a cheating cloud storage provide C'.

We say that the cheating cloud storage provider C' is ϵ-admissible if the probability that the algorithm Verify yields $\mathsf{dec} := \mathsf{accept}$ is at least ϵ:

$$\Pr[\mathcal{O}_{\mathsf{Verify}}(K_{\mathsf{v}}, \mathsf{chal}^*, \mathsf{proof}^*, \mathsf{param}^*_{\mathcal{D}}) \rightarrow \mathsf{accept} \mid \mathsf{proof}^* \leftarrow \mathsf{C}', \mathsf{chal}^* \leftarrow \mathcal{O}_{\mathsf{Chall}}(K_{\mathsf{v}}, \mathsf{param}_{\mathsf{system}})] > \epsilon,$$

for a given verifier key K_{v}.

We say that the Proof of Data Reliability scheme meets the extractability guarantee, if there exists an extractor algorithm $\mathsf{Extract}\ (K_{\mathsf{v}}, \mathsf{fid}^*, \mathsf{param}^*_{\mathcal{D}}, \mathsf{C}') \rightarrow D^*$ such that given sufficient interactions with C', it recovers D.

Definition 3. **(*Req* 1 : Extractability).** We say that a Proof of Data Reliability scheme (Setup, Store, GenR, Chall, Prove, Verify, Repair) is ϵ-sound if there exists an extraction algorithm $\mathsf{Extract}$ such that, for every adversary $\mathcal{A}$ who plays the aforementioned game and outputs an ϵ-admissible cloud storage provider C' for a file D^*, the extraction algorithm $\mathsf{Extract}$ recovers D^* from C' with overwhelming probability.

$$\Pr[\mathsf{Extract}(K_{\mathsf{v}}, \mathsf{fid}^*, \mathsf{param}^*_{\mathcal{D}}, \mathsf{C}') \rightarrow D^*] \leq 1 - \epsilon$$

where ϵ is a negligible function in security parameter λ.

3.4.2 | 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 GenR is implemented by 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.

Definition 4. **(*Req* 2 : Soundness of redundancy generation).** For any adversary $\mathcal{A}$, a Proof of Data Reliability scheme guarantees the *soundness of redundancy computation* if the only way $\mathcal{A}$ can generate a valid proof is by correctly computing the redundancy information.

3.4.3 | Storage Allocation Commitment

A crucial aspect of a Proof of Data Reliability scheme, is forcing cloud storage provider C to store *at rest* the outsourced data object $\mathcal{D}$ *together with* the relevant redundancy. This requirement is formalized similarly to the storage allocation guarantee introduced in Armknecht et al.[10] 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* $\mathcal{D}$ and its redundancy in their entirety, cannot convince verifier V to accept its proof with overwhelming probability.

Definition 5. **(*Req* 3 : Storage allocation commitment).** A Proof of Data Reliability scheme guarantees the *storage allocation commitment* if for any adversary $\mathcal{A}$ who does not store both the data object $\mathcal{D}$ and its redundancy in their entirety and outputs a Proof of Data Reliability proof, the probability that an honest verifier V accepts this proof is negligible (in the security parameter).

3.5 | Rational Adversary

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 this attack.

Why we do not consider a malicious adversary. A malicious adversary is one that may dedicate arbitrary large resources in order to deviate from the correct protocol execution. Its goal is to store the outsourced data object $\mathcal{D}$ only, without the required redundancy, and invoke algorithm GenR when needed in order to produce a valid Proof of Data Reliability while not fulfilling the reliability guarantee t is fulfilled. Besides, if we require that user U generates the required redundancy and further encrypts $\mathcal{D}$ in order to hide the relationship between data and redundancy symbols – akin to a PoR scheme – a Proof of Data Reliability scheme can be secure against a malicious adversary however, it is no longer compatible with *automatic maintenance*.

4 | POROS: PROOF OF DATA RELIABILITY FOR OUTSOURCED STORAGE

In this section, we propose POROS, an erasure-code-based Proof of Data Reliability scheme that enables a cloud storage system to provide data reliability guarantees without disrupting its automatic maintenance operations.

With respect to the integrity verification of the original data (c.f. *Req* 1 in Section 3.4), our scheme leverages the *linearly-homomorphic tags* used in the Private Compact PoR scheme proposed by Shacham and Waters[4] in order to construct a Proof of Data Possession (PDP) scheme which ensures the eventual detection of any attempt by a malicious cloud storage provider C to tamper with the outsourced data. As depicted in Figure 1(a), prior to uploading their data to the cloud storage system, users compute a set of linearly-homomorphic tags that afterwards are used by C to prove the storage of original data.

After receiving the data object $\mathcal{D}$ comprising users' original data and its associated tags (see Figure 1(b)), the cloud storage provider C invokes algorithm GenR which generates $\mathcal{D}$'s redundancy based on the reliable data storage mechanism. In order to facilitate the separate handling of original data and redundancy information we opt for a *systematic linear code* that allows for a clear delimitation between the two: thereby, redundancy symbols are a linear combination of $\mathcal{D}$'s symbols and the latter remain unaltered by the application of the erasure code hence $\mathcal{D}$'s integrity verification –through the use of the PDP scheme– is not affected. Concerning the integrity verification of redundancy symbols, our scheme also leverages the PDP protocol used for data object $\mathcal{D}$ and hence provides users with both guarantees. More precisely, as with the symbols of data object $\mathcal{D}$, algorithm GenR applies the same systematic linear code to the associated tags, yielding a new set of tags that are linear combinations of $\mathcal{D}$'s tags. Assuming that the tags of our PDP scheme are *homomorphic* with respect to the systematic linear code used by the reliable data storage mechanism, the tags resulting from this computation turn out to be PDP tags associated with redundancy symbols. In other words, the linear combination of the tags associated with original data can be used to verify both the correct computation and the integrity of redundancy symbols that are themselves the same linear combination of original data symbols. Figure 1(c) depicts the application of the systematic erasure code on both the data object $\mathcal{D}$ and its redundancy.

Thanks to the homomorphism of the underlying tag scheme at the core of our PDP protocol and to the systematic linear code, the cloud storage provider C does not need any keying material owned by the users in order to compute the tags for the redundancy information. Furthermore, users are able to perform PDP verification on redundancy information using these new tags. Any misconduct by C regarding either the integrity of redundancy symbols or their proper generation will be eventually detected by the PDP verification since the malicious C cannot forge the computed tags.

The scheme described so far suffers from a limitation in that, the malicious cloud storage provider C can take advantage of its capability to independently compute both redundancy information and the corresponding tags, paving the way for C to fool the Proof of Data Reliability verification by computing in real time the responses to PDP checks on redundancy symbols (c.f. *Req* 3 in Section 3.4). The last feature of our scheme thus is a countermeasure to this kind of attacks. This countermeasure relies on timing features of rotational hard drives that are a common component of cloud storage infrastructures. Due to their technical characteristics, such drives achieve much higher throughput during execution of sequential disk access compared to random disk access. The latter results in the execution of multiple expensive seek operations in order for the disk head to reach the different locations on the drive. In order to leverage this performance variation between the two disk access operations, we require that redundancy information is stored in a tailored format with the property of augmenting the random disk access operations of a misbehaving cloud storage provider C. Hence, we are able to introduce a time-threshold $\mathsf{T}_{\mathsf{thr}}$, such that whenever C receives a Proof of Data Reliability challenge, it is compelled to generate and deliver the proof before the time-threshold $\mathsf{T}_{\mathsf{thr}}$ is exceeded; otherwise the proof is rejected.

More precisely, our storage model assumes that all redundancy symbols of a given data object $\mathcal{D}$ is handled as a separate object $\mathcal{R}$ (see Figure 1(c)). Similarly to the permutation-based hourglass scheme in Van Dijk et al.[17], the cloud storage provider C uses a pseudo-random permutation PRP to permute the symbols that compose $\mathcal{R}$ and, stores the result on a *single* storage node

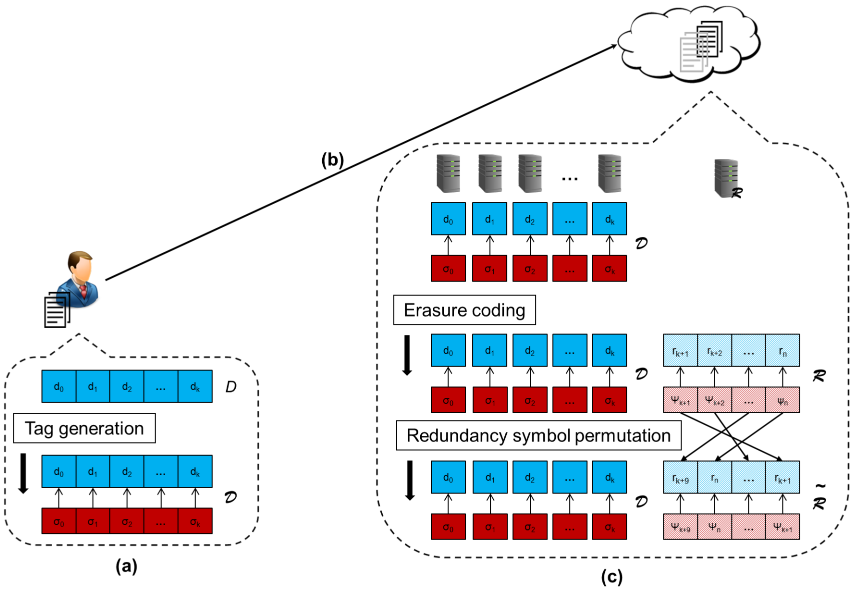


FIGURE 1 Overview of POROS outsourcing process: (a) The user U computes the linearly-homomorphic tags for the original data symbols; (b) U outsources the data object $\mathcal{D}$ to cloud storage provider C; (c) Using **G**, C applies the systematic erasure code on both data symbols and their tags yielding the redundancy symbols and their corresponding tags; thereafter, C permute all redundancy symbols and derives the redundancy object $\tilde{\mathcal{R}}$.

without fragmentation. Disk access operations are done at the granularity of file system blocks[1]. Hence, the resulting redundancy object $\tilde{\mathcal{R}}$ is going to be stored in n contiguous file system blocks, each comprising m symbols. Notice that the newly obtained $\tilde{\mathcal{R}}$ does not prevent the cloud storage provider C from performing automatic maintenance operations since the redundancy object $\mathcal{R}$ can be extracted from $\tilde{\mathcal{R}}$ given the inverse permutation PRP^{-1}.

A user U can challenge C to prove that it stores $\tilde{\mathcal{R}}$ at rest by requesting l consecutive redundancy symbols starting from a randomly chosen position in $\tilde{\mathcal{R}}$. Assuming a random permutation PRP that uniformly distributes the symbols of the redundancy object $\mathcal{R}$ over the file system blocks occupied by $\tilde{\mathcal{R}}$, a compliant cloud storage provider C has to perform one seek operation and then access $\lceil l/m \rceil$ file system blocks sequentially. On the contrary, a malicious C that does not store $\tilde{\mathcal{R}}$ will have difficulty to respond to the challenge in a timely manner, as it has to perform up to l seek operations on the storage nodes that store the original data object $\mathcal{D}$ in order to access the file system blocks that contain the data symbols corresponding to each of the l requested redundancy symbol, transmit these symbols over its internal network and, apply the erasure code up to l times in order to generate all the requested redundancy symbols.

The time-threshold $\mathtt{T}_{\mathsf{thr}}$ is thus defined as a function of time $\mathtt{T}_{\mathsf{h}}$ that an *honest* cloud storage provider C takes to generate the Proof of Reliability and, time $\mathtt{T}_{\mathsf{m}}$ being the response time of a malicious C who does not store $\tilde{\mathcal{R}}$ and thus generates the proof by first recomputing the redundancy the requested redundancy symbols. Preferably, time threshold $\mathtt{T}_{\mathsf{thr}}$ should satisfy the inequality $\mathtt{T}_{\mathsf{h}} < \mathtt{T}_{\mathsf{thr}} \ll \mathtt{T}_{\mathsf{m}}$, implying that the time required for the proof generation is much shorter than the time needed to compute the redundancy.

4.1 | Building Blocks

POROS relies on the following building blocks.

MDS codes. *Maximum distance separable* (MDS) codes[24,25] 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 $[n, k]$–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 $\mathbb{Z}_p$, where p is a large prime and $k \leq n \leq p$. In the event

[1]Typically the block size in current file systems is 4 KB.

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 $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$ and a parity check matrix of the form $\mathbf{H} = [-\mathbf{P}^\top \mid \mathbf{I}_{n-k}]$, where $\mathbf{I}$ denoted the identity matrix. Hence, in a systematic code, the code symbols of a codeword include the data symbols of the original segment. Reed-Solomon codes[24] are a typical example of MDS codes, their generator matrix $\mathbf{G}$ can be easily defined for any given values of (k, n), and are used by a number of storage systems[26].

Linearly–homomorphic tags. Linearly-homomorphic tags[5,4] are additional verification information computed by the user U and outsourced together with file D to the cloud storage provider C. We denote by σ_i the linearly-homomorphic tag corresponding to the data symbol d_i. Linearly-homomorphic tags have the following properties:

- *Unforgeability:* Apart from the user U who owns the signing key, no other party can produce a valid tag σ_i for a data symbol d_i, for which it has not been provided with a tag yet.
- *Verification without the data:* The cloud storage provider C can construct a PoS proof that allows the verifier V to verify the integrity of certain file symbols without having access to the actual data symbols.
- *Linear homomorphism:* Given two linearly-homomorphic tags σ_i and σ_j corresponding to data symbols d_i and d_j, respectively, anyone can compute a linear combination $\alpha\sigma_i + \beta\sigma_j$ that corresponds to the linear combination of the data symbols $\alpha d_i + \beta d_j$, where $\alpha, \beta \in \mathbb{Z}_p$.

Pseudo-random permutation PRP**.** POROS uses a pseudo-random permutation in order to permute the redundancy symbols and construct the redundancy object $\tilde{\mathcal{R}}$. A pseudo-random permutation[27] is a function $\mathsf{PRP} : \mathcal{K} \times \mathcal{X} \to \mathcal{X}$, where $\mathcal{K}$ denotes the set of keys and $\mathcal{X}$ denotes the set of possible inputs and outputs, that has the following properties:

- For any key $K \in \mathcal{K}$ and any input $X \in \mathcal{X}$, there exists an *efficient* algorithm to evaluate $\mathsf{PRP}(K, X)$.
- For any key $K \in \mathcal{K}$, $\mathsf{PRP}(K, .)$ is one-to-one function from $\mathcal{X}$ to $\mathcal{X}$. That means that there exists an inverse function $\mathsf{PRP}^{-1} : \mathcal{K} \times \mathcal{X} \to \mathcal{X}$ such that for any key $K \in \mathcal{K}$ and any input $X \in \mathcal{X}$, there exists an *efficient* algorithm to evaluate $\mathsf{PRP}^{-1}(K, X)$
- PRP cannot be distinguished from a random permutation π, chosen from the uniform distribution of all permutations $\pi : \mathcal{X} \to \mathcal{X}$.

Rotational hard drives. POROS leverages the technical characteristics of rotational hard derives to force the *rational* cloud storage provider C to store the redundancy object $\tilde{\mathcal{R}}$. More specifically, it takes advantage of the difference in throughput such drives achieve during the execution of sequential disk access compared to random disk access in order to devise a time-constrained challenge and detect a cheating cloud storage provider that does not store the redundancy object $\tilde{\mathcal{R}}$ (c.f. *Req* 3 in Section 3.4).

4.2 | Overview of POROS's Integrity Mechanism

POROS's integrity guarantee (c.f. *Req* 1 in Section 3.4) derives from the use of the linearly-homomorphic tags proposed by Shacham and Waters, for the design of the Private Compact PoR scheme[4]. This scheme makes use of a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \to \mathbb{Z}_p$, where λ is the security parameter.

Here, we present the Proof of Data Possession verification for the codeword symbols of an $[n, k]$–MDS code with generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$. The codeword has the form $\left(d^{(1)}, \dots, d^{(k)} \mid r^{(k+1)}, \dots, r^{(n)}\right)$, where $d^{(j)}$ (for $1 \le j \le k$), denote the original data symbols and $r^{(j)}$ (for $k + 1 \le j \le n$), denote the corresponding redundancy symbols. The user U first chooses a random $\alpha \in \mathbb{Z}_p$ and a key $\mathrm{k}_{\mathsf{prf}}$ for function PRF. The tuple $(\alpha, \mathrm{k}_{\mathsf{prf}})$ serves as the user's secret key. She then calculates a tag for each data symbol of the segment as follows:

$$\sigma^{(j)} := \alpha d^{(j)} + \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, j) \in \mathbb{Z}_q, \quad \text{for } 1 \le j \le k.$$

Thereafter, the symbols $\{d^{(j)}\}_{1\le j\le k}$ together with their tags $\{\sigma^{(j)}\}_{1\le j\le k}$ are uploaded to the cloud storage provider C. The verifier V picks l random elements $v_c \in \mathbb{Z}_q$ and l random symbol indices j_c, and sends to C the challenge $\mathsf{chal} := \left\{(j_c, v_c)\right\}_{1\le c\le l}$. The cloud storage provider C then calculates its proof $\mathsf{proof} = (\mu, \tau)$ as follows:

$$\mu := \sum_{(j_c,v_c)\in\mathsf{chal}} v_c\, d^{(j_c)}\,, \quad \tau := \sum_{(j_c,v_c)\in\mathsf{chal}} v_c\, \sigma^{(j_c)}.$$

The verifier V checks that the following equation holds:

$$\tau \overset{?}{=} \alpha\mu + \sum_{(j_c,v_c)\in\mathsf{chal}} v_c\, \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, j_c).$$

As regards to the verification of the redundancy symbols, we observe that $r^{(j)} := \mathbf{d}\cdot\mathbf{G}^{(j)}$, where vector $\mathbf{d} := \left(d^{(1)},\ldots,d^{(k)}\right)$ denotes the vector of data symbols and $\mathbf{G}^{(j)}$ denotes the j^{th} column of generator matrix $\mathbf{G}$, for $k+1\le j\le n$. Hence, algorithm GenR computes the corresponding redundancy tags as: $\psi^{(j)} := \boldsymbol{\sigma}\cdot\mathbf{G}^{(j)}$.

The cloud storage provider C calculates its response $\mathsf{proof} = (\tilde{\mu}, \tilde{\tau})$, the challenge $\mathsf{chal} := \left\{(j_c, v_c)\right\}_{1\le c\le l}$ as follows:

$$\tilde{\mu} := \sum_{(j_c,v_c)\in\mathsf{chal}} v_c\, r^{(j_c)}\,, \quad \tilde{\tau} := \sum_{(j_c,v_c)\in\mathsf{chal}} v_c\, \psi^{(j_c)}.$$

Finally, the verifier V checks that the following equation holds:

$$\tilde{\tau} \overset{?}{=} \alpha\tilde{\mu} + \sum_{(j_c,v_c)\in\mathsf{chal}} v_c\, \mathbf{prf}_{i_c^{(j)}}\cdot\mathbf{G}^{(j)}.$$

where $\mathbf{prf}_{i_c^{(j)}} := \left(\mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, 1),\ldots,\mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, k)\right)$ is the vector of PRFs for the codeword data symbols $\left(d^{(1)},\ldots,d^{(k)}\right)$.

4.3 | POROS Description

POROS is a symmetric Proof of Data Reliability scheme: user U is the verifier V. Thereby, only the user key K_{u} is required for the creation of the data object $\mathcal{D}$, the generation of the Proof of Data Reliability challenge, and the verification of C's proof.

We now describe in detail the algorithms on POROS (the notation used in the description of POROS is summarized in Table 1).

Setup $(1^\lambda, t) \to (\left\{\{S^{(j)}\}_{1\le j\le k}, S_{\mathcal{R}}\right\}, \mathsf{param}_{\mathsf{system}})$**:** Algorithm Setup first picks a prime number p, whose size is chosen according to the security parameter λ. Afterwards, given the reliability parameter t, algorithm Setup yields the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$ of a systematic linear $[n,k]$–MDS code in $\mathbb{Z}_p$, for $t < k < n < p$ and $t \le n-k+1$. In addition, algorithm Setup chooses k storage nodes $\{S^{(j)}\}_{1\le j\le k}$ that are going to store the data object $\mathcal{D}$ and one storage node $S_{\mathcal{R}}$ that is going to store the redundancy object $\mathcal{R}$.

Algorithm Setup then terminates its execution by returning the system parameters $\mathsf{param}_{\mathsf{system}} := (k, n, \mathbf{G}, p)$ and the storage nodes $\left\{\{S^{(j)}\}_{1\le j\le k}, S_{\mathcal{R}}\right\}$.

$\mathsf{U.Store}$ $(1^\lambda, D, \mathsf{param}_{\mathsf{system}}) \to (K_{\mathsf{u}}, \mathcal{D}, \mathsf{param}_{\mathcal{D}})$**:** On input security parameter λ, file $D \in \{0,1\}^*$ and, system parameters $\mathsf{param}_{\mathsf{system}}$, 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 $\mathbb{Z}_p$ and is denoted by $d_i^{(j)}$ for $1\le i\le s$ and $1\le j\le k$.

Algorithm $\mathsf{U.Store}$ also picks a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda\times\{0,1\}^* \to \mathbb{Z}_p$, together with its pseudo-randomly generated key $\mathsf{k}_{\mathsf{prf}} \in \{0,1\}^\lambda$, and a non-zero element $\alpha \xleftarrow{R} \mathbb{Z}_p$. Hereafter, $\mathsf{U.Store}$ computes for each data symbol a *linearly homomorphic* MAC as follows:

$$\sigma_i^{(j)} = \alpha d_i^{(j)} + \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, (i-1)k + j) \in \mathbb{Z}_p.$$

In addition, algorithm $\mathsf{U.Store}$ produces a pseudo-random permutation $\mathsf{PRP} : \{0,1\}^\lambda \times [(n-k)s] \to [(n-k)s]$, together with its pseudo-randomly generated key $\mathsf{k}_{\mathsf{prp}} \in \{0,1\}^\lambda$, and a unique identifier fid.

Algorithm $\mathsf{U.Store}$ then terminates its execution by returning the user key

$$K_{\mathsf{u}} := \left(\mathsf{fid},\ (\alpha, \mathsf{k}_{\mathsf{prf}})\right),$$

Notation	Description
D	File to-be-outsourced
$\mathcal{D}$	Outsourced data object ($\mathcal{D}$ consists of data symbols and PDP tags)
$\mathcal{R}$	Redundancy object ($\mathcal{R}$ consists of redundancy symbols and their PDP tags)
$\tilde{\mathcal{R}}$	Permuted redundancy object
$\mathcal{S}$	Storage node
$\mathcal{S}_{\mathcal{R}}$	Storage node that stores $\tilde{\mathcal{R}}$
$\mathbf{G}$	Generator matrix of the $[n, k]$–MDS code
$\alpha, \mathsf{k}_{\mathsf{prf}}$	Secret key used by the linearly homomorphic tags
j	Codeword symbol index, $1 \leq j \leq n$
i	Data segment index, $1 \leq i \leq s$, ($\mathcal{D}$ consist of s segments)
$d_i^{(j)}$	Data symbol, $1 \leq j \leq k$ and $1 \leq i \leq s$
$\sigma_i^{(j)}$	Data symbol tag, $1 \leq j \leq k$ and $1 \leq i \leq s$
$r_i^{(j)}$	Redundancy symbol, $k+1 \leq j \leq n$ and $1 \leq i \leq s$
$\psi_i^{(j)}$	Redundancy symbol tag, $k+1 \leq j \leq n$ and $1 \leq i \leq s$
$\tilde{r}_i^{(j)}$	Permuted redundancy symbol, $k+1 \leq j \leq n$ and $1 \leq i \leq s$
$\tilde{\psi}_i^{(j)}$	Permuted redundancy symbol tag, $k+1 \leq j \leq n$ and $1 \leq i \leq s$
l	Size of the challenge
$\mathsf{T}_{\mathsf{thr}}$	Time threshold for the proof generation
$i_c^{(j)}$	Indices of challenged symbols, $1 \leq j \leq n$ and $1 \leq c \leq l$
ν_c	Challenge coefficients, $1 \leq c \leq l$
$\mu^{(j)}$	Aggregated data symbols, $1 \leq j \leq n$
$\tau^{(j)}$	Aggregated data tags, $1 \leq j \leq n$
$\tilde{\mu}^{(j)}$	Aggregated redundancy symbols, $1 \leq j \leq n$
$\tilde{\tau}^{(j)}$	Aggregated redundancy tags, $1 \leq j \leq n$
J_{f}	Set of failed storage nodes
J_{r}	Set of surviving storage nodes

TABLE 1 Notation used in the description of POROS.

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

$$\mathcal{D} := \left\{ \mathsf{fid};\ \{d_i^{(j)}\}_{\substack{1 \leq j \leq k \\ 1 \leq i \leq s}};\ \{\sigma_i^{(j)}\}_{\substack{1 \leq j \leq k \\ 1 \leq i \leq s}} \right\},$$

and the data object parameters

$$\mathsf{param}_{\mathcal{D}} := \left(\mathsf{PRP}, \mathsf{k}_{\mathsf{prp}}\right).$$

$\mathsf{C.GenR}$ $(\mathcal{D}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\tilde{\mathcal{R}})$**:** Upon reception of data object $\mathcal{D}$, algorithm $\mathsf{C.GenR}$ starts computing the redundancy symbols $\{r_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}}$ by multiplying each segment $\mathbf{d}_i := \left(d_i^{(1)}, \ldots, d_i^{(k)}\right)$ with the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

$$\mathbf{d}_i \cdot [\mathbf{I}_k \mid \mathbf{P}] = \left(d_i^{(1)}, \ldots, d_i^{(k)} \mid r_i^{(k+1)}, \ldots, r_i^{(n)}\right).$$

Similarly, algorithm $\mathsf{C.GenR}$ multiplies the vector of linearly-homomorphic tags $\boldsymbol{\sigma}_i := \left(\sigma_i^{(1)}, \ldots, \sigma_i^{(k)}\right)$ with $\mathbf{G}$:

$$\boldsymbol{\sigma}_i \cdot [\mathbf{I}_k \mid \mathbf{P}] = \left(\sigma_i^{(1)}, \ldots, \sigma_i^{(k)} \mid \psi_i^{(k+1)}, \ldots, \psi_i^{(n)}\right).$$

One can easily show that $\{\psi_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}}$ are the *linearly-homomorphic* authenticators of $\{r_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}}$.

Thereafter, algorithm $\mathsf{C.GenR}$ uses the pseudo-random permutation $\mathsf{PRP} : \{0,1\}^{\lambda} \times [(n-k)s] \rightarrow [(n-k)s]$ and key $\mathsf{k}_{\mathsf{prp}}$ in order to permute both the redundancy symbols $\{r_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}}$ and the corresponding homomorphic-tags $\{\psi_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}}$ yielding the redundancy object

$$\tilde{\mathcal{R}} := \left\{ \mathsf{fid};\ \{\tilde{r}_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}};\ \{\tilde{\psi}_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}} \right\}.$$

More precisely, if we denote $(\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_{(n-k)s})$ the vector of the permuted redundancy symbols $\{\tilde{r}_i^{(j)}\}_{\substack{k+1 \leq j \leq n \\ 1 \leq i \leq s}}$, then the redundancy symbol $r_i^{(j)}$ is mapped to the position $\mathsf{PRP}(\mathsf{k}_{\mathsf{prp}}, (i-1)(n-k)+j)$ in the permuted redundancy object $\mathcal{R}$. Similarly, if we denote $(\tilde{\psi}_1, \tilde{\psi}_2, \ldots, \tilde{\psi}_{(n-k)s})$ the homomorphic tags' vector after permutation, then tag $\psi_i^{(j)}$ is mapped to to the position $\mathsf{PRP}(\mathsf{k}_{\mathsf{prp}}, (i-1)(n-k)+j)$.

At this point, algorithm C.GenR terminates its execution by storing the data object $\mathcal{D}$ and the redundancy object $\mathcal{R}$ on the storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}$ and $\mathcal{S}_{\mathcal{R}}$, respectively.

U.Chall $(\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{chal})$**:** Provided with the object identifier fid, the secret key K_{u}, and the system parameters $\mathsf{param}_{\mathsf{system}}$, algorithm U.Chall generates a vector $\boldsymbol{v} := (v_c)_{c=1}^{l}$ of l random elements in $\mathbb{Z}_p$, generates a vector $\boldsymbol{c_d} := (i_c, j_c)_{c=1}^{l}$ of l random indice tuples corresponding to data symbols $\{d_i^{(j)}\}_{\substack{1\leq j\leq k\\ 1\leq i\leq s}}$, and picks one random index $1 \leq c_r \leq (n-k)s - l$. Then, algorithm U.Chall terminates by sending C the challenge

$$\mathsf{chal} := \big(\mathsf{fid}, \big(\boldsymbol{c_d},\ c_r,\ \boldsymbol{v}\big)\big).$$

C.Prove $(\mathsf{chal}, \tilde{\mathcal{D}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{proof})$**:** On receiving challenge $\mathsf{chal} = \big(\mathsf{fid},\ (\boldsymbol{c_d},\ c_r,\ \boldsymbol{v})\big)$, algorithm C.Prove first retrieves the authenticated data object $\mathcal{D}$ and the corresponding authenticated redundancy $\tilde{\mathcal{R}}$ that match identifier fid.

Thereupon, algorithm C.Prove processes data object $\mathcal{D}$ as follows:

1. It reads the l requested blocks defined by $\boldsymbol{c_d}$. Without loss of generality, we denote these blocks $\hat{\mathbf{d}} := (\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_l)$.
2. It reads the l tags associated with blocks $\hat{\mathbf{d}}$. We denote these MACs $\hat{\boldsymbol{\sigma}} := (\hat{\sigma}_1, \hat{\sigma}_2, \ldots, \hat{\sigma}_l)$.
3. It computes the inner products

$$\mu = \hat{\mathbf{d}} \cdot \boldsymbol{v} = \sum_{c=1}^{l} \hat{d}_c v_c \tag{1}$$

$$\tau = \hat{\boldsymbol{\sigma}} \cdot \boldsymbol{v} = \sum_{c=1}^{l} \hat{\sigma}_c v_c \tag{2}$$

In the same manner, algorithm C.Prove processes the redundancy object $\mathcal{R}$:

1. It reads l consecutive redundancy blocks starting from block $\tilde{r}_{c_r}$. Let $\tilde{\mathbf{r}}$ denote the l consecutive redundancy blocks $(\tilde{r}_{c_r}, ..., \tilde{r}_{(c_r+l-1)})$.
2. It reads the l consecutive homomorphic MACs associated with redundancy blocks $\tilde{\mathbf{r}}$. Let $\tilde{\boldsymbol{\psi}} := (\tilde{\psi}_{c_r}, ..., \tilde{\psi}_{(c_r+l-1)})$ denote these MACs.
3. It computes the inner products

$$\tilde{\mu} = \tilde{\mathbf{r}} \cdot \boldsymbol{v} = \sum_{c=1}^{l} \tilde{r}_{(c_r+c-1)} v_c \tag{3}$$

$$\tilde{\tau} = \tilde{\boldsymbol{\psi}} \cdot \boldsymbol{v} = \sum_{c=1}^{l} \tilde{\psi}_{(c_r+c-1)} v_c \tag{4}$$

Finally, algorithm C.Prove terminates its execution by returning the proof

$$\mathsf{proof} := \{(\mu, \tau),\ (\tilde{\mu}, \tilde{\tau})\}\ .$$

U.Verify $(K_{\mathsf{u}}, \mathsf{chal}, \mathsf{proof}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{dec})$**:** On input of user key $K_{\mathsf{u}} = (\alpha, \mathrm{k}_{\mathsf{prf}})$, challenge $\mathsf{chal} = \big(\mathsf{fid},\ (\boldsymbol{i_d},\ i_r,\ \boldsymbol{v})\big)$, proof $\mathsf{proof} := \{(\mu, \tau),\ (\tilde{\mu}, \tilde{\tau})\}$, and data object parameters $\mathsf{param}_{\mathcal{D}}$ algorithm U.Verify performs the following checks:

- *Response time verification:* It first checks whether the response time of the server was under time threshold $\mathtt{T}_{\mathsf{thr}}$. If not algorithm U.Verify outputs reject; otherwise it executes the next step.
- *Data possession verification:* Given vector $\boldsymbol{v} = (v_1, ..., v_l)$ and vector $\boldsymbol{c_d} := (i_c, j_c)_{c=1}^{l}$ algorithm U.Verify verifies whether

$$\tau = \alpha\mu + \sum_{c=1}^{l} v_c \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, (i_c - 1)k + j_c) \tag{5}$$

If it is not the case, algorithm U.Verify returns reject; otherwise it moves onto verifying the integrity of the redundancy.

- *Redundancy possession verification:* Algorithm $\mathsf{U.Verify}$ uses the pseudo-random permutation PRP and key $\mathsf{k_{prp}}$, and then for all $1 \leq c \leq l$ it computes the shuffling function preimage $(x_c, y_c) = \mathsf{PRP}^{-1}(\mathsf{k_{prp}}, c_r + c - 1)$. Finally, having matrix $\mathbf{P} = [\mathbf{P}^1 \mid \mathbf{P}^2 \mid \ldots \mid \mathbf{P}^{(n-k)}]$, algorithm $\mathsf{U.Verify}$ checks whether the following equation holds:

$$\tilde{\tau} = \alpha\tilde{\mu} + \sum_{c=1}^{l} v_c \mathbf{P}^{y_c} \cdot \mathbf{prf}_{(x_c)} \tag{6}$$

  wherein for all $1 \leq c \leq l$:

$$\mathbf{prf}_{(x_c)} = \big(\mathsf{PRF}(\mathsf{k_{prf}}, (x_c - 1)k + 1), \ldots, \mathsf{PRF}(\mathsf{k_{prf}}, x_c k)\big)$$

  If so, algorithm $\mathsf{U.Verify}$ outputs accept; otherwise it returns reject.

$\mathsf{C.Repair}$ $({}^*\mathcal{D}, J_\mathsf{f}, \mathsf{param_{system}}, \mathsf{param}_\mathcal{D}, \mathsf{maskGen}) \to (\mathcal{D}, \perp)$**:** On input of a corrupted data object ${}^*\mathcal{D}$ and a set of failed storage node indices $J_\mathsf{f} \subseteq [1, k]$, algorithm $\mathsf{C.Repair}$ first checks if $|J| > n-k+1$, i.e., the lost symbols cannot be reconstructed due to an insufficient number of remaining storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}$.In this case, algorithm $\mathsf{C.Repair}$ terminates outputting $\perp$; otherwise, it uses the surviving storage nodes $\{\mathcal{S}^{(j)}\}_{j\in J_\mathsf{r}}$, where $J_\mathsf{r} \subseteq [1, k] \setminus J_\mathsf{f}$, the redundancy object $\tilde{\mathcal{R}}$, the pseudo-random permutation PRP^{-1} and the parity check matrix $\mathbf{H} = [-\mathbf{P}^\top \mid \mathbf{I}_{n-k}]$ to reconstruct the original data object $\mathcal{D}$.

4.4 | Security Evaluation

In this section, we show that POROS is *correct* and *sound*.

4.4.1 | Correctness

We now show that verification Equations 5 and 6 always hold when algorithm $\mathsf{C.Prove}$ is executed correctly. Subsequently we argue that if time threshold $\mathrm{T_{thr}}$ is correctly tuned then the probability of wrongly accusing C of misbehavior is close to none.

Upon invocation, algorithm $\mathsf{C.Prove}$ first reads l data symbols $\hat{\mathbf{d}} = (\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_l)$ and their corresponding tags $\hat{\boldsymbol{\sigma}} = (\hat{\sigma}_1, \hat{\sigma}_2, \ldots, \hat{\sigma}_l)$, whereby $\hat{d}_1$ is the data symbol $d_{i_1}^{(j_1)}$ specified by the index tuple $c_{d_1} = (i_1, j_1)$. By the definition of linearly-homomorphic tags $\hat{\sigma}_c$, the following equality ensues:

$$\hat{\sigma}_c = \alpha\hat{d}_c + \mathsf{PRF}(\mathsf{k_{prf}}, (i_c - 1)k + j_c),\ \forall\ 1 \leq c \leq l \tag{7}$$

Moreover, algorithm $\mathsf{C.Prove}$ reads l consecutive redundancy symbols $\tilde{\mathbf{r}} = (\tilde{r}_{c_r}, \ldots, \tilde{r}_{(c_r+l-1)})$ together with their corresponding MACs $\tilde{\boldsymbol{\psi}} = (\tilde{\psi}_{c_r}, \ldots, \tilde{\psi}_{(c_r+l-1)})$. Note that for all $1 \leq c \leq l$ redundancy symbol $\tilde{r}_{c_r+c-1}$ corresponds to redundancy symbol $r_{(x_c)}^{y_c} = \mathbf{P}^{y_c} \cdot \mathbf{d}_{(x_c)}$ and MAC $\tilde{\psi}_{c_r+c-1}$ corresponds to $\psi_{y_c}^{(x_c)} = \mathbf{P}^{y_c} \cdot \boldsymbol{\sigma}_{(x_c)}$ whereby $(x_c, y_c) = \mathsf{PRP}^{-1}(\mathsf{k_{prf}}, c_r + c - 1)$ and $\mathbf{P}^{y_c}$ is the y_cth column of the linear code matrix $\mathbf{P}$. Therefore, the following equality always holds.

$$\tilde{\psi}_{c_r+c-1} = \mathbf{P}^{y_c} \cdot (\alpha\mathbf{d}_{(x_c)} + \mathbf{prf}_{(x_c)}) = \alpha\tilde{r}_{c_r+c-1} + \mathbf{P}^{y_c} \cdot \mathbf{prf}_{(x_c)} \tag{8}$$

Where $\mathbf{prf}_{(x_c)} = (\mathsf{PRF}(\mathsf{k_{prf}}, (x_c - 1)k + 1), \ldots, \mathsf{PRF}(\mathsf{k_{prf}}, x_c k))$.

Finally, algorithm $\mathsf{C.Prove}$ finishes its execution by computing four inner products. These inner products are computed as follows:

$$\mu = \hat{\mathbf{d}} \cdot \boldsymbol{v}\ ;\ \tau = \hat{\boldsymbol{\sigma}} \cdot \boldsymbol{v}$$
$$\tilde{\mu} = \tilde{\mathbf{r}} \cdot \boldsymbol{v}\ ;\ \tilde{\tau} = \tilde{\boldsymbol{\psi}} \cdot \boldsymbol{v}$$

Where $\boldsymbol{v} = (v_1, v_2, \ldots, v_l)$ is the random vector generated by the client and transmitted in the challenge message chal.

By plugging Equations 7 and 8 in the inner products, we derive the following equalities:

$$\tau = \alpha\hat{d} + \sum_{c=1}^{l} v_c \mathsf{PRF}(\mathsf{k_{prf}}, (i_c - 1)k + j_c)$$
$$\tilde{\tau} = \alpha\tilde{r} + \sum_{c=1}^{l} v_c \mathbf{P}^{y_c} \cdot \mathbf{prf}_{(x_c)}$$

We can easily see that the above equations are the same as Equations 5 and 6. This means that if the cloud server executes algorithm $\mathsf{C.Prove}$ correctly, then it will pass the verification so long as *its response time is smaller than time threshold* $\mathrm{T_{thr}}$.

4.4.2 | Soundness

Req 1 : **Extractability** We now show that POROS 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 parallelized version of the algorithms SW.Prove and SW.Verify of the Private Compact PoR[4] (see Appendix A for details on the scheme) executed over both the data object $\mathcal{D}$ and and the redundancy object $\mathcal{R}$. More precisely, we assume that the MDS–code parameters $[n, k]$ outputted by algorithm Setup fulfills the requirements in Shacham and Waters[4], in addition to the reliability guarantee t.

We argue that given a sufficient number of interactions with an ϵ-admissible cheating cloud storage provider C', algorithm Extract eventually gathers linear combinations of at least ρ code symbols for each segment of data object $\mathcal{D}$, where $k \leq \rho \leq n$. These linear combinations are of the form

$$\mu = \hat{\mathbf{d}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \hat{d}_c \nu_c$$

$$\tilde{r} = \tilde{\mathbf{r}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{r}_{(c_r+c-1)} \nu_c$$

for known coefficients $(\nu_c)_{c=1}^{l}$ and known indices c_r and c.

Hereby, the extractability arguments given in Shacham and Waters[4] can be applied to the aggregated output of algorithms C.Prove and U.Verify. In particular, given that C' succeeds in making algorithm U.Verify yield dec := accept in an ϵ fraction of the interactions, the pseudo-random permutation PRP uniformly distributes the redundancy symbols in object $\tilde{\mathcal{R}}$, and the indices $\boldsymbol{c_d}$ and c_r of the challenge chosen at random, then algorithm Extract has at its disposal at least $\rho - \epsilon > k$ correct code symbols for each segment of data object $\mathcal{D}$. Therefore, algorithm Extract is able to reconstruct the data object $\mathcal{D}$ using the parity check matrix $\mathbf{H} = [-\mathbf{P}^\top \mid \mathbf{I}_{n-k}]$.

Req 2 : **Soundness of Redundancy Computation** From the definition of linearly-homomorphic tags (c.f. Section 4.1), if the underlying pseudo-random function PRF is secure, then no other party –except for user U who owns the signing key– can can produce a valid tag σ_i for a data symbol d_i, for which it has not been provided with a tag yet. Therefore, 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

$$\tilde{\mu} = \tilde{\mathbf{r}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{r}_{(c_r+c-1)} \nu_c \tag{9}$$

$$\tilde{\tau} = \tilde{\boldsymbol{\psi}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{\psi}_{(c_r+c-1)} \nu_c \tag{10}$$

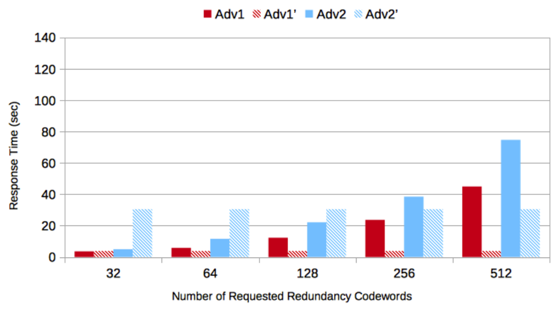
that are computed correctly: i.e., by computing the pair $(\tilde{\mu}, \tilde{\tau})$ using values $\tilde{r}_{(c_r+c-1)}$ and $\tilde{\psi}_{(c_r+c-1)}$ which are the output of algorithm C.GenR.

Req 3 : **Storage Allocation Commitment.** Similarly to Van Dijk et al.[17] storage allocation commitment is met as long as $\mathtt{T}_{\mathsf{thr}} \ll \mathtt{T}_{\mathsf{m}}$, where $\mathtt{T}_{\mathsf{m}} = min(\mathtt{T}_{\mathsf{m}_1}, \mathtt{T}_{\mathsf{m}_2})$ is the response time of a malicious C where $\mathtt{T}_{\mathsf{m}}$ is defined as the minimum of the following values:

- $\mathtt{T}_{\mathsf{m}_1}$: the response time of a malicious C who stores the redundancy information in its original order (i.e. without permutation).
$$\mathtt{T}_{\mathsf{m}_1} = l\mathtt{T}_{\mathsf{Seek}} + l\mathtt{T}_{\mathsf{SeqRead}}(1)$$
- $\mathtt{T}_{\mathsf{m}_2}$: the response time of a malicious C who stores the data object $\mathcal{D}$, only.
$$\mathtt{T}_{\mathsf{m}_2} = l\mathtt{T}_{\mathsf{Seek}} + l\mathtt{T}_{\mathsf{SeqRead}}(k) + l\mathtt{T}_{\mathsf{Encode}}$$

In the above equations, l is the number of sequential redundancy blocks $\tilde{r}$ requested in a POROS challenge, $\mathtt{T}_{\mathsf{Seek}}$ is the time required for a seek operation on the hard drive, $\mathtt{T}_{\mathsf{SeqRead}}(n)$ is the required time to read n data blocks sequentially from the hard disk, $\mathtt{T}_{\mathsf{Encode}}$ is the time required to apply the erasure code defined by the generator matrix $\mathbf{G}$ over a data chunk and, k the number of blocks that comprise a data chunk. Intuitively, $\mathtt{T}_{\mathsf{m}_1}$ should be less than $\mathtt{T}_{\mathsf{m}_2}$.

Moreover, in order to take into account the variations in RTT, the time threshold $\mathtt{T}_{\mathsf{thr}}$ should also satisfy the following condition: $\mathtt{T}_{\mathsf{thr}} > \mathtt{RTT}_{\mathsf{max}} + \mathtt{T}_{\mathsf{h}}$, wherein $\mathtt{RTT}_{\mathsf{max}}$ is the worst-case RTT and $\mathtt{T}_{\mathsf{h}} = \mathtt{T}_{\mathsf{Seek}} + \mathtt{T}_{\mathsf{SeqRead}}(l)$ is the response time of an honest C.



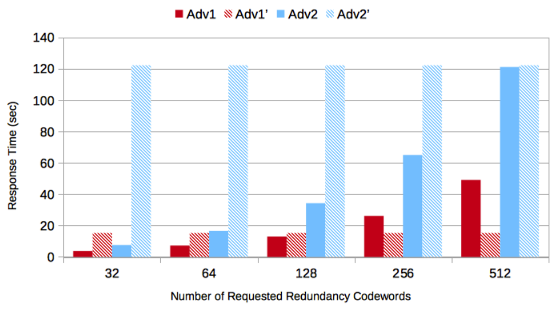


FIGURE 2 Response times of adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ for different challenge sizes l; data object size of 4GB (left) vs. 16GB (right).

Challenge size l	32	64	128	256	512
D (4GB)	21.315 ms	21.159 ms	21.363 ms	20.962 ms	21.825 ms
D (16GB)	26.752 ms	26.479 ms	28.117 ms	27.566 ms	29.234 ms

TABLE 2 Response times of an honest CSP for deferent challenge sizes l.

Fortunately, by carefully tuning parameter l, we can make sure that time-threshold $\mathtt{T}_{\mathsf{thr}}$ satisfies both conditions: This is achieved actually by picking a value for l that guarantees that $\mathtt{RTT}_{\mathsf{max}} \ll \mathtt{T}_{\mathsf{m}} - \mathtt{T}_{\mathsf{h}}$. This makes the scheme robust against false positives.

To conclude *Req* 3 is met as long as the time threshold $\mathtt{T}_{\mathsf{thr}}$ (and therewith l) is tuned such that it fulfills

$$\mathtt{T}_{\mathsf{Seek}} + \mathtt{T}_{\mathsf{SeqRead}}(l) \leq \mathtt{T}_{\mathsf{thr}} < l\mathtt{T}_{\mathsf{Seek}} + l\mathtt{T}_{\mathsf{SeqRead}}(1)$$

Section 4.4.3 provides some hints on the order of $\mathtt{T}_{\mathsf{thr}}$ through an experimental study.

4.4.3 | Evaluation

We have performed an experimental evaluation of POROS' C.Prove algorithm in order to assess the time-constrained proof generation at C. We would like to measure the time that an honest C would take to generate a legitimate proof and compare it with the time a malicious C take to compute its proof. We implemented our prototype in Python and we modified the zfec library[2] in order to compute the generation matrix **G** and apply the erasure code to some test files. All measurements were performed on a local machine with the following characteristics: i5-3470 64 bit processor with 4 cores running at 3.20 GHz, 32GB of RAM at 1600 MHz and, two 320GB HDD at 7200 rpm with a SATA-III 6 Gbps interface. The operating system was Ubuntu Server 14.04.5 LTS with Ext4 as file system and a file system block size of 4 KB. We also measured the sequential throughput of our machine at 131.1 MB per second[3].

We consider two types of adversaries that deviate from the protocol in different ways. The cloud is required to read l consecutive redundancy symbols from the redundancy object $\tilde{\mathcal{R}}$ in the order defined by the permutation PRP and return them to the verifier. Adversary $\mathcal{A}_1$, stores the original redundancy object $\mathcal{R}$ in its original order.

- $\mathcal{A}_1$ attempts to elude detection by seeking on the hard disk the requested redundancy symbols in order to produce the response.
- $\mathcal{A}_2$, does not store any redundancy information at rest: $\mathcal{A}_2$ seeks and retrieves the required data chunks $\mathbf{d}_i$ in order to compute the corresponding redundancy chunks $\mathbf{r}_i$ and then composes the response according to permutation PRP.

For each type of adversary, we also consider another strategy whereby the new adversary $\mathcal{A}'_i$ can take advantage of the available RAM. Hence:

[2]https://pypi.python.org/pypi/zfec
[3]We used bonnie++ to benchmark the performance of the hard drive and file system of our machine. https://www.coker.com.au/bonnie++/

File size	Challenge size l	$T_{\mathcal{A}_1}/T_h$	$T_{\mathcal{A}_2}/T_h$
D (4GB), $\tilde{\mathbf{R}}$ (512MB)	32	167.51	232.19
	64	180.39	546.88
	128	178.66	1035.93
	256	182.39	1459.14
	512	174.38	1399.05
D (16GB), $\tilde{\mathbf{R}}$ (2GB)	32	140.65	282.93
	64	272.77	626.21
	128	461.76	1218.56
	256	553.84	2359.08
	512	552.24	4145.43

TABLE 3 Disadvantage of adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ relative to an honest CSP.

- $\mathcal{A}_1'$ will load the original redundancy object $\tilde{\mathcal{R}}$ within the RAM and subsequently compose her response according to PRP.
- $\mathcal{A}_2'$ will load the whole data object $\mathcal{D}$ to the RAM to further compute the $\tilde{\mathcal{R}}$ and respond with the required symbols.

Finally, we assume that both adversaries choose the strategy that results in the shortest response time for each challenge they receive.

The results presented in Figure 2 and Tables 2 and 3 are the median of 20 independent measurements of the cloud response time; before each measurement we flushed all file system caches.

Figure 2 depicts the response time for $\mathcal{A}_1$, $\mathcal{A}_1'$, $\mathcal{A}_2$ and $\mathcal{A}_1'$ who are expected to store a 4GB data object (left) and a 16GB data object (right) with 12.5% redundancy (512MB and 2GB respectively). Table 2 presents the response time of an honest C which stores the same data objects. The redundancy is computed using a systematic linear $[288, 256]$–MDS code that operates over 64-bit symbols yielding 32 redundancy symbols. In order to apply the code, the 4GB data object $\mathcal{D}$ is divided into data chunks $\mathbf{d}_i$ of size 2KB each. The redundancy object $\mathcal{R}$ is composed of the corresponding redundancy chunks $\mathbf{r}_i$ of 256 Bytes each. All disk access operations are done at the granularity of file system block, whose size is 4KB, therefore each block contains 512 symbols. At this point, the honest C computes $\tilde{\mathcal{R}}$ using the random permutation PRP, $\mathcal{A}_1$ stores $\mathcal{R}$ without permuting it and, $\mathcal{A}_2$ discards the redundancy object.

We observe that the response time of an honest C is on the order of milliseconds whereas the ones of all four adversaries are on the order of seconds. Due to the size of the challenge, an honest C responds by performing one seek operation and by reading from the hard disk one or two consecutive file system blocks. On the contrary, $\mathcal{A}_1$ has to perform up to l seek operations in order to read the required redundancy chunks $\mathbf{r}_i$ or load the whole redundancy object $\mathcal{R}$ to RAM which can take significant more time. In the same way, $\mathcal{A}_2$ has to perform up to l seek operations to retrieve the required data chunks $\mathbf{d}_i$ or read the whole data object $\mathcal{D}$ and further apply the erasure code in order to produce the response. Similarly to the analysis in Van Dijk et al.[17], in the case of the 4GB data object, when the size of the challenge l is larger than 32 redundancy symbols, it is faster for $\mathcal{A}_1$ to load the whole $\mathcal{R}$ in RAM and subsequently compose the response. As regards to adversary $\mathcal{A}_2$, she reaches at this point for a value of l larger than 256 symbols.

In Table 3 we show the ratio between the response time of a malicious adversary and the one of a legitimate C. For example, for a 4GB file and a challenge of size 128, $\mathcal{A}_1$ is 178 times slower than an honest C.

To conclude, our experimental study confirms that by storing redundancy information as a single permuted object, separately from original data, a rational C would chose to conform to the actual POROS protocol and thus it would be forced to store redundancy information at rest. Furthermore, our study also reveals that given the significant gap between the response time of a malicious cloud and that of an honest one, T_{thr} can set to be quite close to the lower bound defined by the time an honest C would take to compute the POROS response for a given file.

5 | PORTOS: PROOF OF DATA RELIABILITY FOR REAL-WORLD DISTRIBUTED OUTSOURCED STORAGE

POROS enables a user to efficiently verify that the cloud server stores her outsourced data correctly and additionally that it complies with the claimed reliable data storage guarantees. In addition, POROS does not prevent the cloud from performing

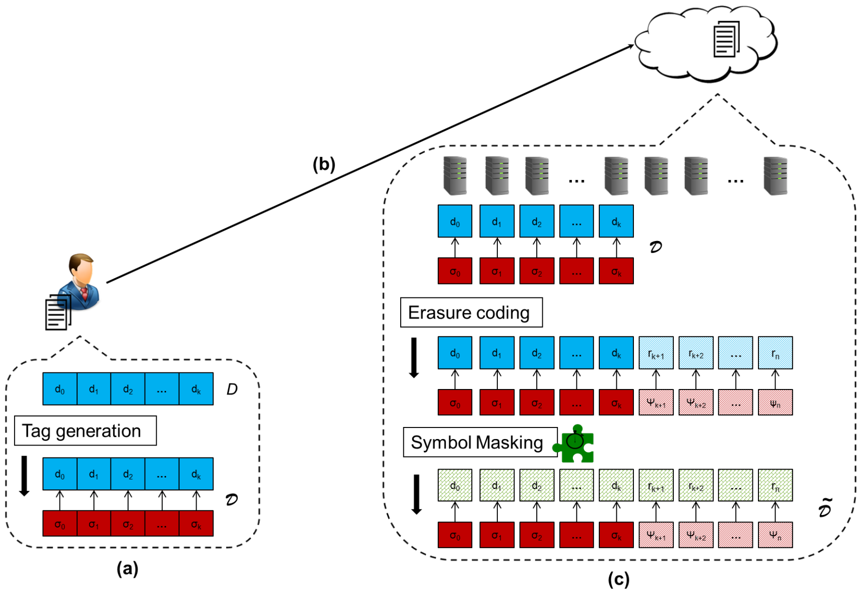


FIGURE 3 Overview of PORTOS outsourcing process: (a) The user U computes the linearly-homomorphic tags for the original data symbols; (b) U outsources the data object $\mathcal{D}$ to cloud storage provider C; (c) Using **G**, C applies the systematic erasure code on both data symbols and their tags yielding the redundancy symbols and their corresponding tags; thereafter, C derives the masking coefficients and masks all data and redundancy symbols.

functional operations such as automatic repair and does not induce any interaction with the client during such maintenance operation. Besides all these advantages, POROS unfortunately comes with its own limitations:

- To begin with, POROS security relies on the underlying technology of cloud storage systems, namely, the use of rotational hard drives as the storage medium. Even though, rotational hard drives are expected to maintain their price per gigabyte advantage compared to other storage technologies (e.g. SSD, NVMe) and thereby remain the preferred storage medium in object storage applications, the eventual introduction of such technologies either as the primary storage medium or as some short of cache in the storage system stack is going to break POROS security.

- Moreover, POROS assumes a back-end storage architecture that deviates from the traditional architecture of erasure-code-based distributed storage systems, wherein each codeword symbol is stored on a distinct storage unit (hard drive, storage node, etc.). POROS's requirement that the redundancy object $\tilde{\mathcal{R}}$ is stored on a single storage node $\mathcal{S}_{\mathcal{R}}$ raises concerns regarding the reliable data storage of $\tilde{\mathcal{R}}$ itself.

In order to cope with these challenges, in this section we introduce a new Proof of Data Reliability scheme called PORTOS. Similarly to POROS, PORTOS uses a systematic linear $[n, k]$–MDS erasure code to add redundancy to the outsourced data. Nevertheless, unlike POROS, PORTOS stores the encoded data across multiple storage nodes: each codeword symbol – both data and redundancy – is stored on a distinct storage node. Hence, the system can tolerate the failure of up to t storage nodes, and successfully reconstructs the original data using the contents of the surviving ones.

Concerning the integrity verification of the outsourced data and its redundancy, PORTOS uses the same Proof of Data Possession (PDP) tags as POROS (c.f. Section 4.2). More specifically, this PDP scheme relies on linearly-homomorphic tags of Private Compact PoR[4] to verify the integrity of the data symbols and it further takes advantage of the homomorphic properties of these tags, in order to verify the integrity of the redundancy symbols. Moreover, thanks to the combination of the PDP tags with the systematic linear $[n, k]$–MDS erasure code, PORTOS ensures a user that she can recover her data in their entirety.

In PORTOS the cloud storage provider has the means to generate the required redundancy, detect failures –either hardware or software– and repair corrupted data entirely on its own, without any interaction with the user. Likewise in POROS, however, this setting 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 in order to augment the resources (storage and computational) a cheating cloud storage provider has to provision in order to produce a valid Proof of Data Reliability.

Nonetheless, this mechanism does not induce any additional storage or computational cost to an honest cloud storage provider that generates the same proof. In this way, a rational adversary is provided with a strong incentive to conform to the Proof of Data Reliability protocol. As a result, PORTOS conforms to the current model of erasure-code-based distributed storage systems. Furthermore, PORTOS does not make any assumption regarding the cloud storage system's underlying technology as opposed to POROS. Figure 3 depicts the outsourcing process of PORTOS.

5.1 | Building Blocks

PORTOS relies on the following building blocks.

MDS code and back-end storage architecture. To generate the necessary redundancy for the reliable storage of users' data, PORTOS uses a systematic linear $[n,k]$–MDS code (c.f. Section 4.1). The code encodes a data segment of size k symbols into a codeword comprising n code symbols. Moreover, in accordance with the current distributed storage architecture, each codeword symbol is stored on a distinct storage node $\{S^{(j)}\}_{1\leq j\leq n}$.

Linearly–homomorphic tags. Similarly to POROS (c.f. Section 4.2), PORTOS relies on the same Proof of Data Possession scheme, based on the linearly-homomorphic tags proposed by Shacham and Waters. Due to the properties of linearly–homomorphic tags, the verifier is able to check the integrity of all codeword symbols as well as the correct generation of redundancy by the cloud storage provider.

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.[28] requires the repeated squaring of a given value β modulo N, where $N := p'q'$ is a publicly known RSA modulus, p' and q' are two safe primes[4] 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 β 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 $\beta^e\ (mod\ N)$.

5.2 | Overview of PORTOS's Masking Mechanism

PORTOS leverages the cryptographic puzzle of Rivest et al.[28] 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 $\mathsf{maskGen}$ which is sent to C together with $\mathcal{D}$ and is used by algorithms GenR and Repair; and the function $\mathsf{maskGenFast}$ which is used by U within the scope of algorithm Verify.

$\mathsf{maskGen}((i,j),\ \mathsf{param_m}) \rightarrow m_i^{(j)}$: This function takes as input the indices $(i,\ j)$, and the tuple $\mathsf{param_m} := (N, \mathcal{T}, \mathsf{PRF_{mask}}, \eta_\mathsf{m})$ comprising the RSA modulus $N := p'q'$, the squaring coefficient $\mathcal{T}$, a pseudo-random function $\mathsf{PRF_{mask}} : \mathbb{Z}_N \times \{0,1\}^* \rightarrow \mathbb{Z}_N$ (such that its output is guaranteed to have a large order modulo N) and a seed $\eta_\mathsf{m} \in \mathbb{Z}_N$.

Function $\mathsf{maskGen}$ computes the masking coefficient $m_i^{(j)}$ as follows:

$$m_i^{(j)} := \left(\mathsf{PRF_{mask}}(\eta_\mathsf{m}, i \parallel j)\right)^{2^{\mathcal{T}}}\ (mod\ N).$$

$\mathsf{maskGenFast}((i,j),\ (p',q'),\ \mathsf{param_m}) \rightarrow m_i^{(j)}$: In addition to $(i,\ j)$ and $\mathsf{param_m} := (N, \mathcal{T}, \mathsf{PRF_{mask}}, \eta_\mathsf{m})$, this function takes as input the secret factors $(p',\ q')$. Knowing p' and q', function $\mathsf{maskGenFast}$ efficiently computes the masking coefficient $m_i^{(j)}$ by first computing the value e:

$$\phi(N) := (p'-1)(q'-1),\quad e := 2^{\mathcal{T}}\ (mod\ \phi(N)),$$
$$m_i^{(j)} := \left(\mathsf{PRF_{mask}}(\eta_\mathsf{m}, i \parallel j)\right)^{e}\ (mod\ N).$$

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

The puzzle's difficulty can be adapted to the computational capacity of C as it evolves over time such that the evaluation of the function maskGen requires a noticeable amount of time to yield $m_i^{(j)}$. 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.

5.3 | PORTOS Description

PORTOS is a symmetric Proof of Data Reliability scheme: user U is the verifier V. Thereby, only the user key K_{u} is required for the creation of the data object $\mathcal{D}$, the generation of the Proof of Data Reliability challenge, and the verification of C's proof.

We now describe in detail the algorithms of PORTOS (the notation used in the description of PORTOS is summarized in Table 4).

$\mathsf{Setup}\ (1^\lambda, t) \rightarrow (\{S^{(j)}\}_{1\le j\le n}, \mathsf{param}_{\mathsf{system}})$: Algorithm Setup first picks a prime number q, whose size is chosen according to the security parameter λ. Afterwards, given the reliability parameter t, algorithm Setup yields the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$ of a systematic linear $[n,k]$–MDS code in $\mathbb{Z}_q$, for $k < n < q$ and $t \le n-k+1$. In addition, algorithm Setup chooses n storage nodes $\{S^{(j)}\}_{1\le j\le n}$ 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 terminates its execution by returning the storage nodes $\{S^{(j)}\}_{1\le j\le n}$ and the system parameters $\mathsf{param}_{\mathsf{system}} := (k, n, \mathbf{G}, q)$.

$\mathsf{U.Store}\ (1^\lambda, D, \mathsf{param}_{\mathsf{system}}) \rightarrow (K_{\mathsf{u}}, \mathcal{D}, \mathsf{param}_{\mathcal{D}}, \mathsf{maskGenFast})$: On input security parameter λ, file $D \in \{0,1\}^*$ and, system parameters $\mathsf{param}_{\mathsf{system}}$, 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 $1 \le i \le s$ and $1 \le j \le k$.

Algorithm U.Store also picks a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{Z}_q$, together with its pseudo-randomly generated key $\mathsf{k}_{\mathsf{prf}} \in \{0,1\}^\lambda$, and a non-zero element $\alpha \overset{R}{\leftarrow} \mathbb{Z}_q$. Hereafter, U.Store computes for each data symbol a *linearly homomorphic* MAC as follows:

$$\sigma_i^{(j)} = \alpha d_i^{(j)} + \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i \parallel j) \in \mathbb{Z}_q.$$

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 λ bits each, and specifies the puzzle difficulty coefficient $\mathcal{T}$, and the time threshold $\mathsf{T}_{\mathsf{thr}}$. Thereafter, algorithm U.Store picks a pseudo-random function $\mathsf{PRF}_{\mathsf{mask}} : \mathbb{Z}_N \times \{0,1\}^* \rightarrow \mathbb{Z}_N$ (such that its output is guaranteed to have a large order modulo N) together with a random seed $\eta_{\mathsf{m}} \overset{R}{\leftarrow} \mathbb{Z}_N$, and constructs the functions maskGen and maskGenFast as described in Section 5.1

$$\mathsf{maskGen}\big((i,j),\ \mathsf{param}_{\mathcal{D}}\big) \rightarrow m_i^{(j)},$$
$$\mathsf{maskGenFast}\big((i,j),\ (p',q'),\ \mathsf{param}_{\mathcal{D}}\big) \rightarrow m_i^{(j)}.$$

Finally, algorithm U.Store picks a pseudo-random generator $\mathsf{PRG}_{\mathsf{chal}} : \{0,1\}^\lambda \rightarrow [1,s]^{l}$ [27] and a unique identifier fid.

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

$$K_{\mathsf{u}} := \big(\mathsf{fid},\ (\alpha, \mathsf{k}_{\mathsf{prf}}),\ (p', q', \mathsf{maskGenFast})\big),$$

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

$$\mathcal{D} := \left\{\mathsf{fid};\ \{d_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}};\ \{\sigma_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}}\right\},$$

and the data object parameters:

$$\mathsf{param}_{\mathcal{D}} := \big(\mathsf{PRG}_{\mathsf{chal}}, \mathsf{maskGen}, \mathsf{param}_{\mathsf{m}} := (N, \mathcal{T}, \eta_{\mathsf{m}}, \mathsf{PRF}_{\mathsf{mask}})\big)\,.$$

$\mathsf{C.GenR}\ (\mathcal{D}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\tilde{\mathcal{D}})$: Upon reception of data object $\mathcal{D}$, algorithm C.GenR starts computing the redundancy symbols $\{r_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ by multiplying each segment $\mathbf{d}_i := \big(d_i^{(1)}, \ldots, d_i^{(k)}\big)$ with the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

$$\mathbf{d}_i \cdot [\mathbf{I}_k \mid \mathbf{P}] = \big(d_i^{(1)}, \ldots, d_i^{(k)} \mid r_i^{(k+1)}, \ldots, r_i^{(n)}\big).$$

Notation	Description
D	File to-be-outsourced
$\mathcal{D}$	Outsourced data object ($\mathcal{D}$ consists of data and PDP tags)
$\tilde{\mathcal{D}}$	Encoded and masked data object
$\mathcal{S}$	Storage node
$\mathbf{G}$	Generator matrix of the $[n, k]$–MDS code
$\alpha, \mathsf{k}_{\mathsf{prf}}$	Secret key used by the linearly homomorphic tags
j	Storage node index, $1 \leq j \leq n$, (there are n $\mathcal{S}$s in total)
i	Data segment index, $1 \leq i \leq s$, ($\mathcal{D}$ consist of s segments)
$d_i^{(j)}$	Data symbol, $1 \leq j \leq k$ and $1 \leq i \leq s$
$\tilde{d}_i^{(j)}$	Masked data symbol, $1 \leq j \leq k$ and $1 \leq i \leq s$
$\sigma_i^{(j)}$	Data symbol tag, $1 \leq j \leq k$ and $1 \leq i \leq s$
$r_i^{(j)}$	Redundancy symbol, $k + 1 \leq j \leq n$ and $1 \leq i \leq s$
$\tilde{r}_i^{(j)}$	Masked redundancy symbol, $k + 1 \leq j \leq n$ and $1 \leq i \leq s$
$\psi_i^{(j)}$	Redundancy symbol tag, $k + 1 \leq j \leq n$ and $1 \leq i \leq s$
$m_i^{(j)}$	Masking coefficient, $1 \leq j \leq n$ and $1 \leq i \leq s$
η_{m}	Random seed used to generate $m_i^{(j)}$
p', q'	Primes for RSA modulus $N := p'q'$ of the time-lock puzzle
$\mathcal{T}$	Time-lock puzzle's difficulty coefficient
l	Size of the challenge
$\mathsf{T}_{\mathsf{thr}}$	Time threshold for the proof generation
$i_c^{(j)}$	Indices of challenged symbols, $1 \leq j \leq n$ and $1 \leq c \leq l$
$\eta^{(j)}$	Random seed used to generate $i_c^{(j)}$, $1 \leq j \leq n$
ν_c	Challenge coefficients, $1 \leq c \leq l$
$\tilde{\mu}^{(j)}$	Aggregated data/redundancy symbols, $1 \leq j \leq n$
$\tau^{(j)}$	Aggregated data/redundancy tags, $1 \leq j \leq n$
J_{f}	Set of failed storage nodes
J_{r}	Set of surviving storage nodes

TABLE 4 Notation used in the description of PORTOS.

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

$$\boldsymbol{\sigma}_i \cdot [\mathbf{I}_k \mid \mathbf{P}] = \left(\sigma_i^{(1)}, \ldots, \sigma_i^{(k)} \mid \psi_i^{(k+1)}, \ldots, \psi_i^{(n)}\right).$$

One can easily show that $\{\psi_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}$ are the *linearly-homomorphic* authenticators of $\{r_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}$.

Thereafter, algorithm C.GenR generates the masking coefficients using the function maskGen:

$$\{m_i^{(j)}\}_{\substack{1\leq j\leq n\\1\leq i\leq s}} := \mathsf{maskGen}\big((i, j),\ \mathsf{param}_{\mathsf{m}}\big)\ (mod\ q),$$

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

$$\{\tilde{d}_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}} \leftarrow \{d_i^{(j)} + m_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}}, \tag{11}$$

$$\{\tilde{r}_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}} \leftarrow \{r_i^{(j)} + m_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}. \tag{12}$$

At this point, algorithm C.GenR deletes all masking coefficients $\{m_i^{(j)}\}_{\substack{1\leq j\leq n\\1\leq i\leq s}}$ and terminates its execution by returning the encoded data object

$$\tilde{\mathcal{D}} := \left\{\mathsf{fid}\ ;\ \left(\{\tilde{d}_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}} \,\middle|\, \{\tilde{r}_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}\right)\ ;\ \left(\{\sigma_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}} \,\middle|\, \{\psi_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}\right)\right\},$$

and by storing the data symbols $\{\tilde{d}_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}}$ together with $\{\sigma_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}}$ and the redundancy symbols $\{\tilde{r}_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}$ together with $\{\psi_i^{(j)}\}_{\substack{k+1\leq j\leq n\\1\leq i\leq s}}$ at the corresponding storage nodes.

U.Chall $(\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{chal})$**:** Provided with the object identifier fid, the secret key K_{u}, and the system parameters $\mathsf{param}_{\mathsf{system}}$, algorithm U.Chall generates a vector $(\nu_c)_{c=1}^{l}$ of l random elements in $\mathbb{Z}_q$ together with a vector of n random seeds $(\eta^{(j)})_{j=1}^{n} \in \{0, 1\}^{\lambda}$, and then, terminates by sending to all storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ the challenge

$$\mathsf{chal} := \left(\mathsf{fid}, \left((\eta^{(j)})_{j=1}^{n},\ (\nu_c)_{c=1}^{l}\right)\right).$$

$\mathsf{C.Prove}\ (\mathsf{chal}, \tilde{\mathcal{D}}, \mathsf{param}_{\mathcal{D}}) \to (\mathsf{proof})$**:** On input of challenge $\mathsf{chal} := \big(\mathsf{fid},\ \big((\eta^{(j)})_{j=1}^{n}, (\nu_c)_{c=1}^{l}\big)\big)$, object parameters $\mathsf{param}_{\mathcal{D}} := (\mathsf{PRG}_{\mathsf{chal}}, \mathsf{maskGen}, \mathsf{param}_{\mathsf{m}})$, and data object $\mathcal{D}$ each storage node $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ invokes an instance of this algorithm and computes the response tuple $(\tilde{\mu}^{(j)}, \tau^{(j)})$ as follows:

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

$$(i_c^{(j)})_{c=1}^{l} := \mathsf{PRG}_{\mathsf{chal}}(\eta^{(j)}), \quad \text{for } 1 \le j \le n,$$

and subsequently, it computes the following linear combination

$$\tilde{\mu}^{(j)} \leftarrow \begin{cases} \sum_{c=1}^{l} \nu_c\, \tilde{d}^{(j)}_{i_c^{(j)}}, & \text{if } 1 \le j \le k \\ \sum_{c=1}^{l} \nu_c\, \tilde{r}^{(j)}_{i_c^{(j)}}, & \text{if } k+1 \le j \le n, \end{cases} \tag{13}$$

$$\tau^{(j)} \leftarrow \begin{cases} \sum_{c=1}^{l} \nu_c\, \sigma^{(j)}_{i_c^{(j)}}, & \text{if } 1 \le j \le k \\ \sum_{c=1}^{l} \nu_c\, \psi^{(j)}_{i_c^{(j)}}, & \text{if } k+1 \le j \le n. \end{cases} \tag{14}$$

Algorithm $\mathsf{C.Prove}$ terminates its execution by returning the set of tuples:

$$\mathsf{proof} := \big\{(\tilde{\mu}^{(j)}, \tau^{(j)})\big\}_{1\le j\le n}.$$

$\mathsf{U.Verify}\ (K_{\mathsf{u}}, \mathsf{chal}, \mathsf{proof}, \mathsf{maskGenFast}, \mathsf{param}_{\mathcal{D}}) \to (\mathsf{dec})$**:** On input of user key $K_{\mathsf{u}} := (\alpha,\ \mathrm{k}_{\mathsf{prf}},\ p',\ q')$, challenge $\mathsf{chal} := \big(\mathsf{fid},\ \big((\eta^{(j)})_{j=1}^{n},\ (\nu_c)_{c=1}^{l}\big)\big)$, proof $\mathsf{proof} := \big\{(\tilde{\mu}^{(j)},\ \tau^{(j)})\big\}_{1\le j\le n}$, function $\mathsf{maskGenFast}$, and data object parameters $\mathsf{param}_{\mathcal{D}} := (\mathsf{PRG}_{\mathsf{chal}},\ \mathsf{maskGen}, \mathsf{param}_{\mathsf{m}})$, this algorithm first checks if the response time of all storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ is shorter than the time threshold $\mathtt{T}_{\mathsf{thr}}$. If not algorithm $\mathsf{U.Verify}$ terminates by outputting $\mathsf{dec} := \mathsf{reject}$; otherwise it continues its execution and checks that all tuples $(\tilde{\mu}^{(j)}, \tau^{(j)})$ in proof are well formed as follows:

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

$$(i_c^{(j)})_{c=1}^{l} := \mathsf{PRG}_{\mathsf{chal}}(\eta^{(j)}), \quad \text{for } 1 \le j \le n,$$

and it generates the corresponding masking coefficients

$$\{m^{(j)}_{i_c^{(j)}}\}_{\substack{1\le j\le n\\ 1\le c\le l}} := \mathsf{maskGenFast}\big((i_c^{(j)}, j),\ (p', q'),\ \mathsf{param}_{\mathcal{D}}\big)$$

Subsequently, it computes

$$\tilde{\tau}^{(j)} := \tau^{(j)} + \alpha \cdot \sum_{c=1}^{l} \nu_c\, m^{(j)}_{i_c^{(j)}}, \tag{15}$$

and then it verifies that the following equations hold

$$\tilde{\tau}^{(j)} \overset{?}{=} \begin{cases} \alpha\tilde{\mu}^{(j)} + \sum_{c=1}^{l} \nu_c\, \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, i_c^{(j)} \| \, j) & \text{if } 1 \le j \le k, \\ \alpha\tilde{\mu}^{(j)} + \sum_{c=1}^{l} \nu_c\, \mathbf{prf}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)} & \text{if } k+1 \le j \le n, \end{cases} \tag{16}$$

where $\mathbf{G}^{(j)}$ denotes the jth column of generator matrix $\mathbf{G}$, and $\mathbf{prf}_{i_c^{(j)}} := \big(\mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, i_c^{(j)} \| \, 1),\ \dots, \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, i_c^{(j)} \| \, k)\big)$ is the vector of PRFs for segment $i_c^{(j)}$.

If the responses from all storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ are correctly computed, algorithm $\mathsf{U.Verify}$ outputs $\mathsf{dec} := \mathsf{accept}$; otherwise it returns $\mathsf{dec} := \mathsf{reject}$.

$\mathsf{C.Repair}\ ({}^{*}\tilde{\mathcal{D}}, J_{\mathsf{f}}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}, \mathsf{maskGen}) \to (\tilde{\mathcal{D}})$**:** On input of a corrupted data object ${}^{*}\tilde{\mathcal{D}}$ and a set of failed storage node indices $J_{\mathsf{f}} \subseteq [1, n]$, algorithm $\mathsf{C.Repair}$ first checks if $|J| > n - k + 1$, i.e., the lost symbols cannot be reconstructed due to an insufficient number of remaining storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$. In this case, algorithm $\mathsf{C.Repair}$ terminates outputting $\perp$; otherwise, it picks a set of k surviving storage nodes $\{\mathcal{S}^{(j)}\}_{j\in J_{\mathsf{r}}}$, where $J_{\mathsf{r}} \subseteq [1, n] \setminus J_{\mathsf{f}}$ and, computes the masking coefficients $\{m_i^{(j)}\}_{\substack{j\in J_{\mathsf{r}}\\ 1\le i\le s}}$ and $\{m_i^{(j)}\}_{\substack{j\in J_{\mathsf{f}}\\ 1\le i\le s}}$, using the function $\mathsf{maskGen}$, together with the parity check matrix $\mathbf{H} = [-\mathbf{P}^{\top} \mid \mathbf{I}_{n-k}]$.

Thereafter, algorithm C.Repair unmasks the symbols held in $\{\mathcal{S}^{(j)}\}_{j\in J_r}$ and reconstructs the original data object $\mathcal{D}$ using $\mathbf{H}$. Finally, algorithm C.Repair uses the generation matrix $\mathbf{G}$ and the coefficients $\{m_i^{(j)}\}_{\substack{j\in J_f\\1\le i\le s}}$ to compute and subsequently mask the content of storage nodes $\{\mathcal{S}^{(j)}\}_{j\in J_f}$.

Algorithm C.Repair then terminates by outputting the repaired data object $\tilde{\mathcal{D}}$.

5.4 | Security Analysis

In this section, we show that PORTOS is *correct* and *sound*.

5.4.1 | Correctness

We now show that the verification Equation 16 must hold if algorithm C.Prove is executed correctly. In particular, Equation 16 consists of two parts: the first one defines the verification of the proofs $\left\{(\tilde{\mu}^{(j)}, \tau^{(j)})\right\}_{1\le j\le k}$ generated by the data storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le k}$; and the second part corresponds to the proofs $\left\{(\tilde{\mu}^{(j)}, \tau^{(j)})\right\}_{k+1\le j\le n}$ generated by the redundancy storage nodes $\{\mathcal{S}^{(j)}\}_{k+1\le j\le n}$. By definition the following equality holds:

$$\sigma_i^{(j)} = \alpha d_i^{(j)} + \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i \parallel j),\ \forall 1 \le i \le s, 1 \le j \le k \tag{17}$$

We begin with the first part of Equation 16. By plugging Equations 15 and 14 to $\tilde{\tau}^{(j)}$ we get

$$\begin{aligned}\tilde{\tau}^{(j)} &= \tau^{(j)} + \alpha \cdot \sum_{c=1}^{l} \nu_c\, m_{i_c^{(j)}}^{(j)} \\ &= \sum_{c=1}^{l} \nu_c\, \sigma_{i_c^{(j)}}^{(j)} + \alpha \cdot \sum_{c=1}^{l} \nu_c\, m_{i_c^{(j)}}^{(j)}.\end{aligned}$$

Thereafter, by Equation 17 we get

$$\begin{aligned}\tilde{\tau}^{(j)} &= \sum_{c=1}^{l} \nu_c \left(\alpha d_{i_c^{(j)}}^{(j)} + \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c^{(j)} \parallel j)\right) + \alpha \cdot \sum_{c=1}^{l} \nu_c\, m_{i_c^{(j)}}^{(j)} \\ &= \alpha \cdot \sum_{c=1}^{l} \nu_c \left(d_{i_c^{(j)}}^{(j)} + m_{i_c^{(j)}}^{(j)}\right) + \sum_{c=1}^{l} \nu_c\, \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c^{(j)} \parallel j).\end{aligned}$$

Finally, by plugging Equations 11 and 13 to Equation 17 we get

$$\tilde{\tau}^{(j)} = \alpha\tilde{\mu}^{(j)} + \sum_{c=1}^{l} \nu_c\, \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c^{(j)} \parallel j).$$

As regards to the second part of Equation 16 that defines the verification of the proofs $\left\{(\tilde{\mu}^{(j)}, \tau^{(j)})\right\}_{k+1\le j\le n}$ generated by the redundancy storage nodes $\{\mathcal{S}^{(j)}\}_{k+1\le j\le n}$, we observe that for all $c \in [1, l]$ it holds that redundancy symbols $r_{i_c^{(j)}}^{(j)} = \boldsymbol{d}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)}$ and tags $\psi_{i_c^{(j)}}^{(j)} = \boldsymbol{\sigma}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)}$, whereby $\mathbf{G}^{(j)}$ is the j^{th} column of generator matrix $\mathbf{G}$, $\boldsymbol{d}_{i_c^{(j)}} := (d_{i_c^{(j)}}^{(1)}, \ldots, d_{i_c^{(j)}}^{(k)})$ is the vector of data symbols for segment i_c, and $\boldsymbol{\sigma}_{i_c^{(j)}} := (\sigma_{i_c^{(j)}}^{(1)}, \ldots, \sigma_{i_c^{(j)}}^{(k)})$ is the corresponding vector of linearly homomorphic tags. Hence, by Equation 17 the following equality always holds:

$$\begin{aligned}\psi_{i_c^{(j)}}^{(j)} &= (\alpha \boldsymbol{d}_{i_c^{(j)}} + \mathbf{prf}_{i_c^{(j)}}) \cdot \mathbf{G}^{(j)} \\ &= \alpha r_{i_c^{(j)}}^{(j)} + \mathbf{prf}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)},\end{aligned}$$

where $\mathbf{prf}_{i_c^{(j)}} := \left(\mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c^{(j)} \parallel 1), \ldots, \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c^{(j)} \parallel k)\right)$ is the vector of PRFs for segment $i_c^{(j)}$. Thereby, given the same straightforward calculations as in the case of data storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le k}$, we derive the following equality:

$$\tilde{\tau}^{(j)} = \alpha\tilde{\mu}^{(j)} + \sum_{c=1}^{l} \nu_c\, \mathbf{prf}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)}.$$

and this proves correctness.

5.4.2 | Soundness

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 $\mathsf{C.Prove}$ and $\mathsf{U.Verify}$ can be seen as a distributed version of the algorithms $\mathsf{SW.Prove}$ and $\mathsf{SW.Verify}$ of the Private Compact PoR[4] (see Appendix A for details on the scheme) executed across all storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$. More precisely, we assume that the MDS–code parameters $[n, k]$ outputted by algorithm Setup, satisfy the requirements in Shacham and Waters[4], in addition to the reliability guarantee t.

We argue that given a sufficient number of interactions with an ϵ-admissible cheating cloud storage provider C', algorithm $\mathsf{Extract}$ eventually gathers linear combinations of at least ρ code symbols for each segment of data object $\mathcal{D}$, where $k \leq \rho \leq n$. These linear combinations are of the form

$$\tilde{\mu}^{(j)} \leftarrow \begin{cases} \sum_{c=1}^{l} \nu_c\, \tilde{d}^{(j)}_{i_c^{(j)}}, & \text{for } 1 \leq j \leq k \\ \sum_{c=1}^{l} \nu_c\, \tilde{r}^{(j)}_{i_c^{(j)}}, & \text{for } k+1 \leq j \leq n, \end{cases}$$

for known coefficients $(\nu_c)_{c=1}^{l}$ and known indices $i_c^{(j)}$ and j. Furthermore, U can efficiently derive the unmasked expressions

$$\mu^{(j)} \leftarrow \begin{cases} \sum_{c=1}^{l} \nu_c\, d^{(j)}_{i_c^{(j)}}, & \text{for } 1 \leq j \leq k \\ \sum_{c=1}^{l} \nu_c\, r^{(j)}_{i_c^{(j)}}, & \text{for } k+1 \leq j \leq n \end{cases}$$

by computing the masking coefficients $\{m^{(j)}_{i_c^{(j)}}\}_{\substack{1\leq j\leq n \\ 1\leq c\leq l}}$ using the function $\mathsf{maskGenFast}$, and subtracting from $\tilde{\mu}^{(j)}$ the corresponding linear combination $\sum_{c=1}^{l} \nu_c\, m^{(j)}_{i_c^{(j)}}$.

Hereby, the extractability arguments given in Shacham and Waters[4] can be applied to the aggregated output of algorithms $\mathsf{C.Prove}$ and $\mathsf{U.Verify}$. More precisely, given that C' succeeds in making algorithm $\mathsf{U.Verify}$ yield $\mathsf{dec} := \mathsf{accept}$ in an ϵ fraction of the interactions, and the indices $i_c^{(j)}$ of the challenge chosen at random, then algorithm $\mathsf{Extract}$ has at its disposal at least $\rho - \epsilon > k$ correct code symbols for each segment of data object $\mathcal{D}$. Therefore, algorithm $\mathsf{Extract}$ is able to reconstruct the data object $\mathcal{D}$ using the parity check matrix $\mathbf{H} = [-\mathbf{P}^{\top} \mid \mathbf{I}_{n-k}]$.

Req 2 : **Soundness of Redundancy Generation.** From the definition of linearly-homo-morphic tags (c.f. Section 4.1), if the underlying pseudo-random function PRF is secure, then no other party – except user U who owns the signing key – can produce a valid tag σ_i for a data symbol d_i, for which it has not been provided with a tag yet. Therefore, 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

$$\tilde{\mu}^{(j)} \leftarrow \sum_{c=1}^{l} \nu_c\, \tilde{r}^{(j)}_{i_c^{(j)}}, \quad \text{for } k+1 \leq j \leq n, \tag{18}$$

$$\tau^{(j)} \leftarrow \sum_{c=1}^{l} \nu_c\, \psi^{(j)}_{i_c^{(j)}}, \quad \text{for } k+1 \leq j \leq n. \tag{19}$$

that are computed correctly: i.e., by computing the pair $(\tilde{\mu}, \tau)$ using values $\tilde{r}^{(j)}_{i_c^{(j)}}$ and $\psi^{(j)}_{i_c^{(j)}}$ which are the output of algorithm $\mathsf{C.GenR}$.

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 $\mathtt{T}_{\mathsf{thr}}$ is tuned properly.

In essence, PORTOS consists of parallel proof of data possession challenges over all storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$: 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 $\mathtt{T}_{\mathsf{resp}}$ and $\mathtt{T}'_{\mathsf{resp}}$, we can determine the lower and upper bounds for the time threshold $\mathtt{T}_{\mathsf{thr}}$.

A fundamental design feature of PORTOS is that C' has to compute one masking coefficient for each symbol of the encoded data object $\mathcal{D}$. 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 $n-k+1$

Scenario	**Proof generation complexity for one $\mathcal{S}$**	**$\mathcal{S}$'s Response Time**
Honest C:	$2l$ mult $+ 2(l+1)$ add	$\mathtt{T}_{\mathsf{resp}} = \left\lceil \frac{2l}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}} + \left\lceil \frac{2(l-1)}{\Pi} \right\rceil \mathtt{T}_{\mathsf{add}} + \mathtt{RTT}$
C' stores $\mathcal{D}$ unmasked:	$l\mathcal{T}$ exp $+2l$ mult $+ (3l-2)$ add	$\mathtt{T}'_{\mathsf{resp}_1} = \left\lceil \frac{l}{\Pi} \right\rceil \mathtt{T}_{\mathsf{puzzle}} + \left\lceil \frac{2l}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}} + \left\lceil \frac{3l-2}{\Pi} \right\rceil \mathtt{T}_{\mathsf{add}} + \mathtt{RTT}$
C' deletes up to $s(n-k+1)$ symbols of $\tilde{\mathcal{D}}$:	**$\mathcal{S}$ with missing symbol:** $\mathcal{T}$ exp $+2l$ mult $+ (2l+k-1)$ add **k $\mathcal{S}$s participating in symbol generation:** $\mathcal{T}$ exp $+ (2l+1)$ mult $+ (2l-1)$ add **Remaining $\mathcal{S}$s:** $2l$ mult $+ 2(l-1)$ add	$\mathtt{T}'_{\mathsf{resp}_2} = \mathtt{T}_{\mathsf{puzzle}} + \left\lceil \frac{2l+1}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}} + \left\lceil \frac{2l-1}{\Pi} \right\rceil \mathtt{T}_{\mathsf{add}} + \mathtt{RTT}$

TABLE 5 Evaluation of the response time and the effort required by a storage node $\mathcal{S}$ 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 $\mathcal{D}$ encoded but unmasked; and (ii) by not storing the data object $\tilde{\mathcal{D}}$ in its entirety. $\mathtt{RTT}$ is the round trip time between the user U and C; Π is the number of computations $\mathcal{S}$ can perform in parallel; and $\mathtt{T}_{\mathsf{puzzle}} := \mathcal{T} \cdot \mathtt{T}_{\mathsf{exp}}$ is the time required by $\mathcal{S}$ to generate one masking coefficient.

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 $\{m^{(j)}_{i^{(j)}_c}\}_{\substack{1\le j\le n \\ 1\le c\le l}}$.

(ii) The second way C' may misbehave, is by not storing the data object $\tilde{\mathcal{D}}$ 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 $\tilde{\mathcal{D}}$ either by not provisioning up to $n-k+1$ storage nodes; or by uniformly dropping symbols from all n storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$, ensuring that it preserves at least k symbols for each data segment.

In order to determine the lower bound for the time threshold $\mathtt{T}_{\mathsf{thr}}$ we evaluate the response time $\mathtt{T}_{\mathsf{resp}}$ 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 $\mathtt{T}_{\mathsf{thr}}$ as $\mathtt{T}'_{\mathsf{resp}} = \min(\mathtt{T}'_{\mathsf{resp}_1}, \mathtt{T}'_{\mathsf{resp}_2})$, where $\mathtt{T}'_{\mathsf{resp}_1}$ and $\mathtt{T}'_{\mathsf{resp}_2}$ respectively denote C''s response time when it opts to keep $\mathcal{D}$ unmasked and delete symbols of $\tilde{\mathcal{D}}$, respectively. Concerning the evaluation of $\mathtt{T}'_{\mathsf{resp}_2}$, we consider the most favorable scenario for C' where it has to generate only one missing symbol for a PORTOS challenge. Table 5 presents the effort required by a storage node $\mathcal{S}$ 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 $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ have a bounded capacity of Π concurrent threads of execution, that computations – exponentiations, multiplications, additions, etc. – require a minimum execution time, and that $\mathtt{T}_{\mathsf{add}} \ll \mathtt{T}_{\mathsf{exp}}$ and $\mathtt{T}_{\mathsf{add}} \ll \mathtt{T}_{\mathsf{mult}}$. Furthermore, we assume that $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ are connected with premium network connections (low latency and high bandwidth), and hence the communication among them has negligible impact on C' response time. As given in Table 5, *Req* 3 is met as long as the time threshold $\mathtt{T}_{\mathsf{thr}}$ is tuned such that it fulfills the following relations:

$$\mathtt{T}_{\mathsf{thr}} > \mathtt{RTT}_{\mathsf{max}} + \left\lceil \frac{2l}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}} \text{ (Lower bound)},$$
$$\mathtt{T}_{\mathsf{thr}} < \mathtt{RTT}_{\mathsf{max}} + \mathtt{T}_{\mathsf{puzzle}} + \left\lceil \frac{2l+1}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}} \text{ (Upper bound)},$$

where $\mathtt{RTT}_{\mathsf{max}}$ is the worst-case $\mathtt{RTT}$, $\mathtt{T}_{\mathsf{puzzle}} := \mathcal{T} \cdot \mathtt{T}_{\mathsf{exp}}$ is the time required by $\mathcal{S}$ to evaluate the function $\mathsf{maskGen}$ ans Π is the number of computations $\mathcal{S}$ can perform in parallel. By carefully setting the puzzle difficulty coefficient $\mathcal{T}$, we can guarantee that $\mathtt{T}_{\mathsf{resp}} \ll \mathtt{T}'_{\mathsf{resp}}$ and $\mathtt{RTT}_{\mathsf{max}} \ll \mathtt{T}'_{\mathsf{resp}} - \mathtt{T}_{\mathsf{resp}}$, 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 $\mathcal{T}$ accordingly.

Metric		**Cost**
Storage:		$2 \cdot s \cdot n$ symbols
Bandwidth:	Challenge:	$n \cdot \lambda$ bits and l symbols
	Proof:	$2 \cdot n$ symbols
U.Store complexity:		$s \cdot k$ PRF $+ s \cdot k$ mult $+ s \cdot k$ add
C.Prove complexity:		n $\mathsf{PRG}_{\mathsf{chal}} + 2 \cdot n \cdot l$ mult $+ 2 \cdot n \cdot (l+1)$ add
U.Verify complexity:		n $\mathsf{PRG}_{\mathsf{chal}} + 2 \cdot n \cdot l$ exp $+ k \cdot l \cdot (n-k+1)$ PRF $+2 \cdot n \cdot (l+1) + k \cdot l \cdot (n-k)$ mult $+ (n-k) \cdot (kl+k+2)$ add

TABLE 6 Performance analysis of PORTOS.

5.5 | Performance Analysis

Table 6 summarizes the computational, storage, communication costs of PORTOS.

Storage After outsourcing the verifiable data object $\mathcal{D}$, user U is only required to store her secret key K_{u}. On the other hand, cloud storage provider C stores the encoded data object $\tilde{\mathcal{D}}$ which amounts to $2 \cdot s \cdot n$ symbols, where s is the number of segments in $\tilde{\mathcal{D}}$, n is the number of symbols in each segment, and a symbol is an element of $\mathbb{Z}_q$. This value includes the storage overhead induced by the application of the $[n,k]$–MDS code – $s \cdot (n-k)$ symbols – and the PDP-tags – $s \cdot n$ symbols.

Bandwidth The transmission of a PORTOS challenge $\mathsf{chal} := \left(\mathsf{fid}, \left((\eta^{(j)})_{j=1}^{n}, (\nu_c)_{c=1}^{l}\right)\right)$ requires bandwidth of $n \cdot \lambda$ bits and l symbols, where n the number of random seeds $\eta^{(j)}$ – one for each storage node $\{S^{(j)}\}_{1 \le j \le n}$) – and l is the size of the challenge – the number of requested symbols on each storage node. Furthermore, the bandwidth required to transmit the proof $\mathsf{proof} := \left\{(\tilde{\mu}^{(j)}, \tau^{(j)})\right\}_{1 \le j \le n}$ generated by cloud storage provider C is $2 \cdot n$ symbols: a pair of aggregated symbols and tags for each storage node $\{S^{(j)}\}_{1 \le j \le n}$.

Computation Algorithm U.Store computes one linearly-homomorphic tag for each symbol of file D. This operation translates to $s \cdot k$ PRF computation, $s \cdot k$ multiplications, and $s \cdot k$ additions, where s is the number of segments in D and k is the number of symbols in each segment.

Algorithm C.Prove is executed in parallel on all storage nodes $\{S^{(j)}\}_{1 \le j \le n}$. Each storage node eluates one pseudo-random generator $\mathsf{PRG}_{\mathsf{chal}}$ to derive the indices of the requested symbols and thereafter performs $2 \cdot l$ multiplications and $2 \cdot (l+1)$ additions to compute its response.

As regards to algorithm U.Verify, we observe that the computational effort required to verify the response of a redundancy node $\{S^{(j)}\}_{k+1 \le j \le n}$ is much higher than the effort required to verify the response of a data node $\{S^{(j)}\}_{1 \le j \le k}$. Namely, in addition to the $2nl$ exponentiations required by the function maskGenFast to generate the masking coefficients of the nl involved symbols, algorithm U.Verify evaluates kl PRFs and $k + 2(l+1)$ multiplications in 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.

5.6 | Performance Improvements

In order to improve the performance of our scheme with respect to all three storage, communication, and verification costs, we adopt the storage efficient variant of the linearly-homomorphic tags of Private Compact PoR[4] (see Appendix A for details on the scheme). More specifically, algorithm U.Store computes a linearly homomorphic tag for each data segment, comprising k symbols, instead of a tag per symbol. As regards to the verification of C's proof, upon the timely reception of all storage node responses, algorithm U.Verify first uses the aggregated tags in order to validate the integrity of the data node responses. Thereafter, algorithm U.Verify multiplies the data node responses with the generator matrix $\mathbf{G}$, and compares the output to the redundancy node responses. Unfortunately, the new scheme does not meet the extractability requirement (c.f. *Req* 1 in Section 3.4): the proof of reliability verification is done at the segment level and there is no redundancy among the segments of $\tilde{D}$, thus there cannot exist an extractor algorithm that can recover the original file D against a cheating cloud storage provider C′, who succeeds in making algorithm Verify yield $\mathsf{dec} := \mathsf{accept}$ in an non-negligible ϵ fraction of proof of data reliability executions. In order to fulfill the extractability requirement, the new algorithm U.Store first encodes the file D using an error-correcting code, that meets the retrievability requirements stated in Shacham and Waters[4], and subsequently permutes and encrypts D, before computing the homomorphic tags and outputting the data object $\mathcal{D}$.

The proofs of data reliability generated by the two schemes have different granularity with respect to the detection of corrupted storage nodes $\{S^{(j)}\}_{1\leq j\leq n}$. On the one hand, the basic PORTOS scheme individually verifies the response of each storage node. Hence, the misbehaving storage nodes are identified with great precision. On the other hand, in the storage efficient variant of PORTOS, a failed verification of the data node responses, positively detects that there is corruption on one or more data nodes $\{S^{(j)}\}_{1\leq j\leq k}$ without further specifying neither the number nor the identity of the misbehaving nodes. Moreover, if the the verification of the data node responses fail, algorithm $\mathsf{U.Verify}$ cannot proceed to the verification of the responses of the redundancy nodes $\{S^{(j)}\}_{k+1\leq j\leq n}$.

5.6.1 | Description

We now describe in detail the algorithms of PORTOS with segment tags.

$\mathsf{Setup}\ (1^\lambda, t) \to (\{S^{(j)}\}_{1\leq j\leq n}, \mathsf{param}_{\mathsf{system}})$**:** Algorithm Setup first picks a prime number q, whose size is chosen according to the security parameter λ. Afterwards, given the reliability parameter t, algorithm Setup yields the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$ of a systematic linear $[n, k]$–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 terminates its execution by returning the storage nodes $\{S^{(j)}\}_{1\leq j\leq n}$, and the system parameters $\mathsf{param}_{\mathsf{system}} := (k, n, \mathbf{G}, q)$.

$\mathsf{U.Store}\ (1^\lambda, D, \mathsf{param}_{\mathsf{system}}) \to (K_{\mathsf{u}}, \mathcal{D}, \mathsf{param}_{\mathcal{D}}, \mathsf{maskGenFast})$**:** On input security parameter λ, file $D \in \{0,1\}^*$, and system parameters $\mathsf{param}_{\mathsf{system}}$, this randomized algorithm first applies the ECC code, and subsequently permutes and encrypts D obtaining D'. Thereafter it 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 $\mathsf{U.Store}$ also picks a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \to \mathbb{Z}_q$, together with its pseudo-randomly generated key $\mathsf{k}_{\mathsf{prf}} \in \{0,1\}^\lambda$, and k non-zero elements $\{\alpha^{(j)}\}_{1\leq j\leq k} \overset{R}{\leftarrow} \mathbb{Z}_q$. Hereafter, $\mathsf{U.Store}$ computes for each data symbol a *linearly homomorphic* MAC as follows:

$$\sigma_i = \sum_{j=1}^{k} \alpha^{(j)} d_i^{(j)} + \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i) \in \mathbb{Z}_q.$$

In addition, algorithm $\mathsf{U.Store}$ 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 λ bits each, and specifies the puzzle difficulty coefficient $\mathcal{T}$, and the time threshold $\mathsf{T}_{\mathsf{thr}}$. Thereafter, algorithm $\mathsf{U.Store}$ picks a pseudo-random generator $\mathsf{PRF}_{\mathsf{mask}} : \mathbb{Z}_N \times \{0,1\}^* \to \mathbb{Z}_N$[5] together with a random seed $\eta_{\mathsf{m}} \overset{R}{\leftarrow} \mathbb{Z}_N$, and constructs the functions $\mathsf{maskGen}$ and $\mathsf{maskGenFast}$ as described in Section 5.1

$$\mathsf{maskGen}\big((i,j),\ \mathsf{param}_{\mathcal{D}}\big) \to m_i^{(j)},$$
$$\mathsf{maskGenFast}\big((i,j),\ (p',q'),\ \mathsf{param}_{\mathcal{D}}\big) \to m_i^{(j)}.$$

Finally, algorithm $\mathsf{U.Store}$ picks a pseudo-random generator $\mathsf{PRG}_{\mathsf{chal}} : \{0,1\}^\lambda \to [1,s]^l$ and a unique identifier fid.

Algorithm $\mathsf{U.Store}$ then terminates its execution by returning the user key

$$K_{\mathsf{u}} := \big(\mathsf{fid},\ \{\alpha^{(j)}\}_{1\leq j\leq k}, \mathsf{k}_{\mathsf{prf}}),\ (p', q', \mathsf{maskGenFast})\big),$$

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

$$\mathcal{D} := \left\{\mathsf{fid};\ \{d_i^{(j)}\}_{\substack{1\leq j\leq k\\1\leq i\leq s}};\ \{\sigma_i\}_{1\leq i\leq s}\right\},$$

and the data object parameters

$$\mathsf{param}_{\mathcal{D}} := \big(\mathsf{PRG}_{\mathsf{chal}}, \mathsf{maskGen}, \mathsf{param}_{\mathsf{m}} := (N, \mathcal{T}, \eta_{\mathsf{m}}, \mathsf{PRF}_{\mathsf{mask}})\big).$$

[5]such that its output is guaranteed to have a large order modulo N.

$\mathsf{C.GenR}\ (\mathcal{D}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}) \to (\tilde{\mathcal{D}})$**:** Upon reception of data object $\mathcal{D}$, algorithm $\mathsf{C.GenR}$ starts computing the redundancy symbols $\{r_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ by multiplying each segment $\mathbf{d}_i := \left(d_i^{(1)}, d_i^{(2)}, \dots, d_i^{(k)}\right)$ with the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

$$\mathbf{d}_i \cdot [\mathbf{I}_k \mid \mathbf{P}] = \left(d_i^{(1)}, d_i^{(2)}, \dots, d_i^{(k)} \mid r_i^{(k+1)}, r_i^{(k+2)}, \dots, r_i^{(n)}\right).$$

Thereafter, algorithm $\mathsf{C.GenR}$ generates the masking coefficients using the function $\mathsf{maskGen}$:

$$\{m_i^{(j)}\}_{\substack{1\le j\le n\\ 1\le i\le s}} := \mathsf{maskGen}\big((i,j),\ \mathsf{param}_{\mathsf{m}}\big)\ (mod\ q),$$

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

$$\{\tilde{d}_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}} \leftarrow \{d_i^{(j)} + m_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}},$$
$$\{\tilde{r}_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}} \leftarrow \{r_i^{(j)} + m_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}.$$

At this point, algorithm $\mathsf{C.GenR}$ deletes all masking coefficients $\{m_i^{(j)}\}_{\substack{1\le j\le n\\ 1\le i\le s}}$ and terminates its execution by returning the encoded data object

$$\tilde{\mathcal{D}} := \left\{ \mathsf{fid}\ ;\ \left(\{\tilde{d}_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}} \,\middle|\, \{\tilde{r}_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}} \right);\ \{\sigma_i\}_{1\le i\le s} \right\},$$

and by storing the data symbols $\{\tilde{d}_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}}$ and the redundancy symbols $\{\tilde{r}_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ at the corresponding storage nodes together with $\{\sigma_i\}_{1\le i\le s}$.

$\mathsf{U.Chall}\ (\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}}) \to (\mathsf{chal})$: Provided with the object identifier fid, the secret key K_{u}, and the system parameters $\mathsf{param}_{\mathsf{system}}$, algorithm $\mathsf{U.Chall}$ generates a vector $(\nu_c)_{c=1}^{l}$ of l random elements in $\mathbb{Z}_q$ together with a random seed $\eta \in \{0,1\}^{\lambda}$, and then, terminates by sending to all storage nodes $\{S^{(j)}\}_{1\le j\le n}$ the challenge

$$\mathsf{chal} := \big(\mathsf{fid}, \big((\eta,\ (\nu_c)_{c=1}^{l}\big)\big).$$

$\mathsf{C.Prove}\ (\mathsf{chal}, \tilde{\mathcal{D}}, \mathsf{param}_{\mathcal{D}}) \to (\mathsf{proof})$**:** On input of challenge $\mathsf{chal} := \big(\mathsf{fid}, \big((\eta, (\nu_c)_{c=1}^{l}\big)\big)$, object parameters $\mathsf{param}_{\mathcal{D}} := (\mathsf{PRG}_{\mathsf{chal}}, \mathsf{maskGen}, \mathsf{param}_{\mathsf{m}})$, and data object $\mathcal{D}$, each storage node $\{S^{(j)}\}_{1\le j\le n}$ invokes an instance of this algorithm and computes its response $\tilde{\mu}^{(j)}$ as follows:

It first derives the indices of the requested segments

$$(i_c)_{c=1}^{l}\ :=\ \mathsf{PRG}_{\mathsf{chal}}(\eta),$$

and subsequently, it computes the following linear combination

$$\tilde{\mu}^{(j)}\ \leftarrow\ \begin{cases} \sum_{c=1}^{l} \nu_c\, \tilde{d}_{i_c}^{(j)}, & \text{if } 1 \le j \le k \\ \sum_{c=1}^{l} \nu_c\, \tilde{r}_{i_c}^{(j)}, & \text{if } k+1 \le j \le n. \end{cases}$$

In addition, algorithm $\mathsf{C.Prove}$ the linear combination of tags

$$\tau\ \leftarrow\ \sum_{c=1}^{l} \nu_c\, \sigma_{i_c}, \tag{20}$$

and terminates its execution by returning the proof:

$$\mathsf{proof} := \big\{\{\tilde{\mu}^{(j)}\}_{1\le j\le n}, \tau\big\}.$$

$\mathsf{U.Verify}\ (K_{\mathsf{u}}, \mathsf{chal}, \mathsf{proof}, \mathsf{maskGenFast}, \mathsf{param}_{\mathcal{D}}) \to (\mathsf{dec})$**:** On input of user key $K_{\mathsf{u}} := (\{\alpha^{(j)}\}_{1\le j\le k},\ \mathrm{k_{prf}},\ p',\ q')$, challenge $\mathsf{chal} := \big(\mathsf{fid}, \big((\eta,\ (\nu_c)_{c=1}^{l}\big)\big)$, proof $\mathsf{proof} :=$ $\big\{\{\tilde{\mu}^{(j)}\}_{1\le j\le n}, \tau\big\}$, function $\mathsf{maskGenFast}$, and data object parameters $\mathsf{param}_{\mathcal{D}} := (\mathsf{PRG}_{\mathsf{chal}}, \mathsf{maskGen}, \mathsf{param}_{\mathsf{m}})$, this algorithm first checks if the response time of all storage nodes $\{S^{(j)}\}_{1\le j\le n}$ is shorter than the time threshold $\mathtt{T}_{\mathsf{thr}}$. If not algorithm $\mathsf{U.Verify}$ terminates by outputting $\mathsf{dec} := \mathsf{reject}$; otherwise it continues its execution and checks that the $\mathsf{proof} := \big\{\{\tilde{\mu}^{(j)}\}_{1\le j\le n}, \tau\big\}$ is well formed as follows:

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

$$(i_c)_{c=1}^{l}\ :=\ \mathsf{PRG}_{\mathsf{chal}}(\eta),$$

and subsequently, it generates the corresponding masking coefficients and unmasks the storage node responses as follows:

$$\{m_{i_c}^{(j)}\}_{\substack{1\le j\le n\\ 1\le c\le l}} := \mathsf{maskGenFast}\big((i_c, j),\ (p', q'),\ \mathsf{param}_{\mathcal{D}}\big)\ (mod\ q),$$

$$\mu^{(j)} := \tilde{\mu}^{(j)} - \sum_{c=1}^{l} \nu_c\, m_{i_c}^{(j)},\ \text{for } 1 \le j \le n.$$

Algorithm U.Verify then checks that the data node responses $\{\mu^{(j)}\}_{1\le j\le k}$ are well formed by verifying that the following equation holds:

$$\tau \overset{?}{=} \sum_{j=1}^{k} \alpha^{(j)}\mu^{(j)} + \sum_{c=1}^{l} \nu_c\ \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c),$$

and subsequently, it multiplies $\boldsymbol{\mu} := (\mu^{(1)}, \mu^{(2)}, \dots, \mu^{(k)})$ with the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

$$\boldsymbol{\mu} \cdot [\mathbf{I}_k \mid \mathbf{P}] = (\mu^{(1)}, \mu^{(2)}, \dots, \mu^{(k)} \mid \mu^{*(k+1)}, \mu^{*(k+2)}, \dots, \mu^{*(n)}).$$

Thereafter algorithm U.Verify verify the redundancy node responses are well formed as follows:

$$\{\mu^{(j)}\}_{k+1\le j\le n} \overset{?}{=} \{\mu^{*(j)}\}_{k+1\le j\le n}.$$

If the responses from all storage nodes $\{S^{(j)}\}_{1\le j\le n}$ are well formed, algorithm U.Verify outputs dec := accept; otherwise it returns dec := reject.

C.Repair ($^*\tilde{\mathcal{D}}, J_{\mathsf{f}}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}, \mathsf{maskGen}) \to (\tilde{\mathcal{D}})$: On input of a corrupted data object $^*\tilde{\mathcal{D}}$ and a set of failed storage node indices $J_{\mathsf{f}} \subseteq [1, n]$, algorithm C.Repair first checks if $|J| > n - k + 1$, i.e., the lost symbols cannot be reconstructed due to an insufficient number of remaining storage nodes $\{S^{(j)}\}_{1\le j\le n}$. In this case, algorithm C.Repair terminates outputting $\perp$; otherwise, it picks a set of k surviving storage nodes $\{S^{(j)}\}_{j\in J_{\mathsf{r}}}$, where $J_{\mathsf{r}} \subseteq [1, n] \setminus J_{\mathsf{f}}$ and, computes the masking coefficients $\{m_i^{(j)}\}_{\substack{j\in J_{\mathsf{r}}\\ 1\le i\le s}}$ and $\{m_i^{(j)}\}_{\substack{j\in J_{\mathsf{f}}\\ 1\le i\le s}}$, using the function maskGen, together with the parity check matrix $\mathbf{H} = [-\mathbf{P}^\top \mid \mathbf{I}_{n-k}]$.

Thereafter algorithm C.Repair unmasks the symbols held in $\{S^{(j)}\}_{j\in J_{\mathsf{r}}}$ and reconstructs the original data object $\mathcal{D}$ using $\mathbf{H}$. Finally, algorithm C.Repair uses the generation matrix $\mathbf{G}$ and the coefficients $\{m_i^{(j)}\}_{\substack{j\in J_{\mathsf{f}}\\ 1\le i\le s}}$ to compute and subsequently mask the content of storage nodes $\{S^{(j)}\}_{j\in J_{\mathsf{f}}}$.

Algorithm C.Repair then terminates by outputting the repaired data object $\tilde{\mathcal{D}}$.

5.6.2 | Performance Analysis

Table 7 summarizes the performance impact of the storage efficient variant of the linearly-homomorphic tags has on PORTOS. The new scheme with segment tags significantly improves the communication, proof generation, and verification complexity at the cost of a significant drop in the performance of algorithm U.Store. Furthermore, the new scheme is more space-efficient, provided that the following inequality holds: $1 + 1/k < 2\rho$. Lastly, the new scheme meets the *Req* 3 (c.f. Section 3.4) as long as the time threshold $\mathtt{T}_{\mathsf{thr}}$ is tuned such that it fulfills the following relation:

Metric		**Cost**
Storage:		$\frac{s}{\rho} \cdot (n+1)$ symbols
Bandwidth:	Challenge: Proof:	λ bits and l symbols $n+1$ symbols
U.Store complexity:		$1\ \mathsf{ECC} + \frac{s\cdot k}{\rho}\ \mathsf{PRP} + 1\ \mathsf{Enc} + \frac{s}{\rho}\ \mathsf{PRF} + \frac{s\cdot k}{\rho}\ \mathsf{mult} + \frac{s\cdot k}{\rho}\ \mathsf{add}$
C.Prove complexity:		$1\ \mathsf{PRG}_{\mathsf{chal}} + l \cdot (n+1)\ \mathsf{mult} + (l-1)\cdot(n+1)\ \mathsf{add}$
U.Verify complexity:		$1\ \mathsf{PRG}_{\mathsf{chal}} + 2 \cdot n \cdot l\ \mathsf{exp} + l\ \mathsf{PRF}$ $+(2\cdot k + l\cdot(n+1))\ \mathsf{mult} + (2\cdot(k-1) + l\cdot(n+1))\ \mathsf{add}$

TABLE 7 Performance analysis of PORTOS with storage efficient tags.

$$\mathtt{RTT}_{\mathsf{max}} + \left\lceil \frac{l}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}} < \mathtt{T}_{\mathsf{thr}} < \mathtt{RTT}_{\mathsf{max}} + \mathtt{T}_{\mathsf{puzzle}} + \left\lceil \frac{l+1}{\Pi} \right\rceil \mathtt{T}_{\mathsf{mult}},$$

where $\mathtt{RTT}_{\mathsf{max}}$ is the worst-case $\mathtt{RTT}$, $\mathtt{T}_{\mathsf{puzzle}} := \mathcal{T} \cdot \mathtt{T}_{\mathsf{exp}}$ is the time required by $\mathcal{S}$ to evaluate the function $\mathsf{maskGen}$ and Π is the number of computations $\mathcal{S}$ can perform in parallel.

6 | CONCLUSION

In this paper, we presented two novel Proof of Data Reliability solutions for erasure-code-based distributed cloud storage systems.

The first one, called called POROS, aims to provide a secure yet practical solution that enables a user to efficiently verify that the cloud storage provider stores her outsourced data correctly and additionally that it complies with the claimed reliable data storage guarantees. Running the POROS protocol, a client is assured that the cloud storage provider actually stores *at res*t both the original data and the corresponding redundancy and it does not compute the latter on-the-fly upon a Proof of Data Reliability challenge. Moreover, POROS does not prevent the cloud from performing functional operations such as automatic repair and does not induce any interaction with the client during such maintenance operation. We analyzed the security of POROS both theoretically and through experiments measuring the time difference in computing a proof between an honest cloud and some rational adversaries.

Besides all these advantages, POROS unfortunately comes with its own limitations:

- POROS security relies on the underlying technology of cloud storage systems, namely, the use of rotational hard drives as the storage medium. Unfortunately, the introduction of novel storage technologies – such as SSD or NVMe drives – is going to break POROS security.
- Moreover, POROS assumes a back-end storage architecture that deviates from the traditional architecture of erasure-code based distributed storage systems, wherein each codeword symbol is stored on a distinct storage node. POROS's requirement that the redundancy object is stored on a single storage node raises concerns regarding the reliable data storage of the redundancy object itself.

The second solution we presented, called PORTOS, satisfies the same Proof of Data Reliability requirements as POROS, while overcoming the shortcoming of POROS. In particular, PORTOS design does not make any assumptions regarding the underlying storage medium technology nor it deviates from the current distributed storage architecture. To defend against an adversary who computes the redundancy on-the-fly upon a Proof of Data Reliability challenge, PORTOS relies on time-lock puzzles in order to augment the resources (storage and computational) a cheating cloud storage provider has to provision in order to produce a valid Proof of Data Reliability. Nonetheless, this mechanism does not induce any additional storage or computational cost to an honest cloud storage provider that generates the same proof. On the other hand, PORTOS's $\mathsf{C.Repair}$ algorithm is less efficient than the POROS one because the unmasking/masking of the data object is a more computationally intensive operation compared to the in memory permutation of the redundancy object. We analyzed both the security of the protocol and we show that PORTOS is secure against a *rational* adversary. Moreover, we analyzed the performance of PORTOS in terms of storage, communication, and verification cost. Finally, we proposed a more efficient version of the protocol which improves the performance of both the cloud storage provider and the verifier at the cost of reduced in granularity with respect to the detection of corrupted storage nodes.

References

1. Vasilopoulos D, Elkhiyaoui K, Molva R, Önen M. POROS: Proof of data reliability for outsourced storage. Paper presented at: SCC '18. Proceedings of the 6th International Workshop on Security in Cloud Computing. 2018 Jun 4; Incheon, Republic of Korea. New York, NY: ACM; 2018. p. 27–37.

2. Vasilopoulos D, Önen M, Molva R. PORTOS: Proof of data reliability for real-world distributed outsourced storage. Paper presented at: SECRYPT '19. Proceedings of the 16th International Joint Conference on e-Business and Telecommunications. Volume 2: SECRYPT. 2019 Jul 26–28; Prague, Czech Republic. Setúbal, Portugal: INSTICC; 2019. p. 173–186.

3. Juels A, Kaliski BS. Pors: Proofs of retrievability for large files. Paper presented at: CCS '07. Proceedings of the 14th ACM Conference on Computer and Communications Security. 2007 Oct 29 – Nov 2; Alexandria, VA. New York, NY: ACM; 2007. p. 584–597.

4. Shacham H, Waters B. Compact proofs of retrievability. *Journal of Cryptology.* 2013 Jul;26(3):442–483.

5. Ateniese G, Burns R, Curtmola R, Herring J, Kissner L, Peterson Z et al. Provable data possession at untrusted stores. Paper presented at: CCS '07. Proceedings of the 14th ACM Conference on Computer and Communications Security. 2007 Oct 29 – Nov 2; Alexandria, VA. New York, NY: ACM; 2007. p. 598–609.

6. Curtmola R, Khan O, Burns R, Ateniese G. MR-PDP: Multiple-replica provable data possession. Paper presented at: ICDCS '08. The 28th International Conference on Distributed Computing Systems. 2008 Jun 17–20; Beijing, China. Washington, DC: IEEE Computer Society; 2008. p. 411–420.

7. Chen B, Curtmola R. Remote data integrity checking with server-side repair. *Journal of Computer Security.* 2017 Aug;25(6):537–584.

8. Bowers KD, Juels A, Oprea A. HAIL: A high-availability and integrity layer for cloud storage. Paper presented at: CCS '09. Proceedings of the 16th ACM Conference on Computer and Communications Security. 2009 Nov 9–13; Chicago, IL. New York, NY: ACM; 2009. p. 187–198.

9. Chen B, Ammula AK, Curtmola R. Towards server-side repair for erasure coding-based distributed storage systems. Paper presented at: CODASPY '15. Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. 2015 Mar 2–4; San Antonio, TX. New York, NY: ACM; 2015. p. 281–288.

10. Armknecht F, Barman L, Bohli JM, Karame GO. Mirror: Enabling proofs of data replication and retrievability in the cloud. Paper presented at: USENIX Security '16. 25th USENIX Security Symposium. 2016 Aug 10–12; Austin, TX. Berkeley, CA: USENIX Association; 2016. p. 1051–1068.

11. Barsoum A, Hasan A. Enabling dynamic data and indirect mutual trust for cloud computing storage systems. *IEEE Transactions on Parallel and Distributed Systems.* 2013 Dec;24(12):2375–2385.

12. Barsoum A, Hasan A. Integrity verification of multiple data copies over untrusted cloud servers. Paper presented at: CCGrid '12. Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. 2012 May 13–16; Ottawa, Canada. Washington, DC: IEEE Computer Society; 2012. p. 829–834.

13. Barsoum A, Hasan M. Provable multicopy dynamic data possession in cloud computing systems. *IEEE Transactions on Information Forensics and Security.* 2015 Mar;10(3):485–497.

14. Etemad M, Küpçü A. Transparent, distributed, and replicated dynamic provable data possession. Paper presented at: ACNS '13. 11th International Conference on Applied Cryptography and Network Security. 2013 Jun 25–28; Banff, Alberta, Canada. Berlin, Heidelberg: Springer; 2013. p. 1–18.

15. Erway C, Küpçü A, Papamanthou C, Tamassia R. Dynamic provable data possession. Paper presented at: CCS '09. Proceedings of the 16th ACM Conference on Computer and Communications Security. 2009 Nov 9–13; Chicago, IL. New York, NY: ACM; 2009. p. 213–222.

16. Chen B, Curtmola R. Towards self-repairing replication-based storage systems using untrusted clouds. Paper presented at: CODASPY '13. Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy. 2013 Feb 18–20; San Antonio, TX. New York, NY: ACM; 2013. p. 377–388.

17. Van Dijk M, Juels A, Oprea A, Rivest RL, Stefanov E, Triandopoulos N. Hourglass schemes: How to prove that cloud files are encrypted. Paper presented at: CCS '12. Proceedings of the 19th ACM Conference on Computer and Communications Security. 2012 Oct 16–18; Raleigh, NC. New York, NY: ACM; 2012. p. 265–280.

18. Leontiadis I, Curtmola R. Secure storage with replication and transparent deduplication. Paper presented at: CODASPY '18. Proceedings of the 8th ACM Conference on Data and Application Security and Privacy. 2018 Mar 19–21; Tempe, AZ. New York, NY: ACM; 2018. p. 13–23.

19. Chen B, Curtmola R, Ateniese G, Burns R. Remote data checking for network coding-based distributed storage systems. Paper presented at: CCSW ’10. Proceedings of the 2010 ACM Workshop on Cloud Computing Security. 2010 Oct 8; Chicago, IL. New York, NY: ACM; 2010. p. 31–42.

20. Le A, Markopoulou A. NC-Audit: Auditing for network coding storage. Paper presented at: NetCod ’12. Proceedings of the 2012 International Symposium on Network Coding. 2012 Jun 29–30; Cambridge, MA. Washington, DC: IEEE Computer Society; 2012. p. 155–160.

21. Le A, Markopoulou A. Locating byzantine attackers in intra-session network coding using SpaceMac. Paper presented at: NetCod ’10. Proceedings of the 2010 International Symposium on Network Coding. 2010 Jun 9–11; Toronto, Canada. Washington, DC: IEEE Computer Society; 2010. p. 1–6.

22. Thao TP, Omote K. ELAR: Extremely lightweight auditing and repairing for cloud security. Paper presented at: ACSAC ’16. Proceedings of the 32nd Annual Conference on Computer Security Applications. 2016 Dec 5–9; Los Angeles, CA. New York, NY: ACM; 2016. p. 40–5.

23. Bowers KD, Van Dijk M, Juels A, Oprea A, Rivest RL. How to tell if your cloud files are vulnerable to drive crashes. Paper presented at: CCS ’11. Proceedings of the 18th ACM Conference on Computer and Communications Security. 2011 Oct 17–21; Chicago, IL. New York, NY: ACM; 2011. p. 501–514.

24. Xing C, Ling S. Coding theory: A first course. Cambridge, UK: Cambridge University Press; 2003.

25. Suh C, Ramchandran K. Exact-repair MDS code construction using interference alignment. *IEEE Transactions on Information Theory*. 2011 Apr;57(3):1425–1442.

26. Blaum M, Brady J, Bruck J, Menon J. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. *ACM SIGARCH Computer Architecture News*. 1994 Apr;22(2):245–254.

27. Katz J, Lindell Y. Introduction to modern cryptography. 2nd ed. Boca Raton, FL: CRC Press; 2014.

28. Rivest RL, Shamir A, Wagner DA. Time-lock puzzles and timed-release crypto. Cambridge, MA: MIT; 1996.

How to cite this article: Vasilopoulos D, Önen M, Molva R, and Elkhiyaoui K. Proofs of data reliability: Verification of reliable data storage with automatic maintenance. *Security and Privacy*. 2020 *e137*.

APPENDIX

A THE PRIVATE POR SCHEME OF SHACHAM AND WATERS

In what follows, we briefly describe the private scheme in Shacham and Waters[4]. The scheme takes advantage of a pseudo-random function PRF and comprises the following algorithms:

SW.KeyGen $(1^\lambda) \rightarrow (K_u, \mathsf{param}_{\mathsf{system}})$: User U calls this algorithm in order to generate a secret key K_u and a set of system parameters $\mathsf{param}_{\mathsf{system}}$ (size of segment, erasure code, etc.) that will be used to prepare D for upload and to verify its retrievability later.

SW.Encode $(K_u, D) \rightarrow (\mathsf{fid}, \mathcal{D})$: User U calls this algorithm to prepare her file D for outsourcing to cloud storage provider C. It applies the erasure code to D and then uses the secret key K_u to permute and encrypt the encoded file, and further outputs the result $\hat{D}$. Subsequently, SW.Encode divides $\hat{D}$ in n equally-sized segments each comprising m symbols. We denote $\hat{d}_{ij}$ the j^{th} symbol of the i^{th} segment where $1 \leq i \leq n$ and $1 \leq j \leq m$. Algorithm SW.Encode then generates a PRF key $\mathsf{k}_{\mathsf{prf}}$, chooses m random numbers $\alpha_j \in \mathbb{Z}_p$ where $1 \leq j \leq m$ and computes for each segment the following homomorphic MAC σ_i:

$$\sigma_i := \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i) + \sum_{j=1}^{m} \alpha_j \hat{d}_{ij}$$

Algorithm $\mathsf{SW.Encode}$ then picks a unique identifier fid, and terminates its execution by outsourcing to the cloud storage provider C the authenticated data object:

$$\mathcal{D} := \left\{\mathsf{fid};\ \{\hat{d}_{ij}\}_{\substack{1\le j\le m\\1\le i\le n}};\ \{\sigma_i\}_{1\le i\le n}\right\}.$$

$\mathsf{SW.Chall}\ (K_{\mathsf{u}}, \mathsf{fid}) \rightarrow (\mathsf{chal})$: This algorithm invoked by user U picks l random elements $v_c \in \mathbb{Z}_p$ and l random symbol indices i_c, and sends to cloud storage provider C the challenge

$$\mathsf{chal} := \left\{(i_c, v_c)\right\}_{1\le c\le l}.$$

$\mathsf{SW.Prove}\ (\mathsf{fid}, \mathsf{chal}) \rightarrow (\mathsf{proof})$: Upon receiving the challenge $\mathsf{chal} := \left\{(i_c, v_c)\right\}_{1\le c\le l}$, C invokes this algorithm which computes the proof $\mathsf{proof} := (\mu_j, \tau)$ as follows:

$$\mu_j := \sum_{(i_c,v_c)\in\mathsf{chal}} v_c\ \hat{d}_{i_c j}\ ,\quad \tau := \sum_{(i_c,v_c)\in\mathsf{chal}} v_c\ \sigma_{i_c}.$$

$\mathsf{SW.Verify}\ (K_{\mathsf{u}}, \mathsf{proof}, \mathsf{chal}) \rightarrow (\mathsf{dec})$: Given user key K_{u}, proof $\mathsf{proof} := (\mu_j, \tau)$, and challenge $\mathsf{chal} := \left\{(i_c, v_c)\right\}_{1\le c\le l}$, this algorithm invoked by user U, verifies that the following equation holds:

$$\tau \overset{?}{=} \sum_{j=1}^{m} \alpha_j \mu_j + \sum_{(i_c,v_c)\in\mathsf{chal}} v_c\ \mathsf{PRF}(\mathsf{k}_{\mathsf{prf}}, i_c).$$

If proof is correctly computed, algorithm $\mathsf{SW.Verify}$ outputs $\mathsf{dec} := \mathsf{accept}$; otherwise it returns $\mathsf{dec} := \mathsf{reject}$.