POROS: Proof of Data Reliability for Outsourced Storage

Dimitrios Vasilopoulos
EURECOM
Sophia Antipolis, France
vasilopo@eurecom.fr

Kaoutar Elkhiyaoui
IBM Research Zurich
Zurich, Switzerland
kao@zurich.ibm.com

Refik Molva
EURECOM
Sophia Antipolis, France
molva@eurecom.fr

Melek Önen
EURECOM
Sophia Antipolis, France
onen@eurecom.fr

ABSTRACT

We introduce POROS that is a new solution for proof of data reliability. In addition to the integrity of the data outsourced to a cloud storage system, proof of data reliability assures the customers that the cloud storage provider (CSP) has provisioned sufficient amounts of redundant information along with original data segments to be able to guarantee the maintenance of the data in the face of corruption. In spite of meeting a basic service requirement, the placement of the data repair capability at the CSP raises a challenging issue with respect to the design of a proof of data reliability scheme. Existing schemes like Proof of Data Possession (PDP) and Proof of Retrievability (PoR) fall short of providing proof of data reliability to customers, since those schemes are not designed to audit the redundancy mechanisms of the CSP. Thus, in addition to verifying the possession of the original data segments, a proof of data reliability scheme must also assure that sufficient redundancy information is kept at storage. Thanks to some combination of PDP with time constrained operations, POROS guarantees that a *rationale* CSP 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 CSP with the repair function, POROS allows for the automatic maintenance of data by the storage provider without any interaction with the customers.

KEYWORDS

secure cloud storage; proofs of reliability; reliable storage; verifiable storage

ACM Reference Format:
Dimitrios Vasilopoulos, Kaoutar Elkhiyaoui, Refik Molva, and Melek Önen. 2018. POROS: Proof of Data Reliability for Outsourced Storage. In *SCC'18: 6th International Workshop on Security in Cloud Computing, June 4, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3201595.3201600

SCC'18, June 4, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5759-3/18/06…$15.00
https://doi.org/10.1145/3201595.3201600

1 INTRODUCTION

As the core feature of cloud computing, outsourcing raises some unprecedented security issues in the face of potentially misbehaving or malicious service providers. In the case of cloud storage systems, if the misbehaviour of a cloud storage provider (CSP) goes undetected, nothing prevents a malicious CSP from deleting significant amounts of data from storage in order to maximize the utilization of the storage system. Focusing on this type of security requirements, several research projects came up with a large body of work (e.g. [3, 4, 17, 19]) aiming at solutions for verifiable outsourced storage. The proposed solutions mainly consist of cryptographic protocols that enable (potentially lightweight) clients to efficiently verify the availability of their data in the storage system, that is, a client only needs to download a small fraction of her data in order to verify that the latter is stored correctly - safe for accidental errors or malicious attacks.

Existing verifiable storage solutions like Proof of Data Possession (PDP) and Proof of Retrievability (PoR) suffer from a major shortcoming: they do not take into account a basic feature of storage systems that is automatic maintenance performed by the storage provider. Since data repair functions are part of the basic storage service, the assurance of data availability with such services must verify not only the availability of the data outsourced by the clients but also the effectiveness of the data repair functions. A comprehensive verification scheme called "proof of data reliability" should thus verify both the availability of the clients' data and the provision by the storage provider for sufficient means to recover from minor data loss in storage. Even though some PDP or PoR schemes involve some error correction function performed by the clients, no existing PDP or PoR scheme focuses on the verification of error recovery functions performed by the storage provider.

In addition to the integrity of the data outsourced to a cloud storage system, a proof of data reliability thus assures the clients that the CSP stores sufficient amounts of redundancy information along with original data segments to be able to guarantee the maintenance of the data in the face of minor corruption or loss.

Furthermore, in the adversarial setting of cloud storage systems, the placement of the data repair capability at the CSP raises a challenging issue with respect to the design of a proof of data reliability scheme. Simple schemes verifying data availability alone - PDP and PoR - fall short of achieving data reliability assurance in that context, since those schemes cannot prevent a malicious CSP from storing only the original data and from answering PDP or

PoR queries using the CSP's data repair capability to compute the redundancy information on the fly. Thus, in addition to verifying the availability of the original data segments, a proof of data reliability scheme must also assure that redundancy information is kept at storage instead of being computed on the fly.

In this paper, we propose POROS, a new scheme that achieves data reliability assurance in the context of a potentially malicious cloud storage provider despite the challenging difficulty due to the automatic maintenance by the storage provider. Thanks to some combination of PDP with time constrained operations, POROS guarantees that a rational CSP would not compute redundancy on demand upon PDP requests but instead would store it at rest. Our idea thus is to set a time-threshold τ such that if the CSP stores the data together with the redundancy correctly, it will be able to successfully answer the POROS challenges received from the client before τ elapses; otherwise, it will fail to provide a timely response, and thereby, its misbehavior will be detected. As a result of bestowing the CSP with the repair function, POROS allows for the automatic maintenance of data by the storage provider without any interaction with the clients.

Contributions.

- We define the notion of proof of data reliability and identify the inherent conflict between proof of reliability mechanisms and automatic maintenance;
- We propose POROS, a data reliability scheme, that on the one hand, enables a client to verify the correct storage of her data and the corresponding redundancy; and on the other hand, allows the cloud to perform automatic maintenance operations;
- We analyze the security of POROS both theoretically and through experiments measuring the time difference between an honest cloud and some malicious adversaries.

2 PROBLEM STATEMENT

Data reliability is achieved by means of data redundancy and data integrity mechanisms. In the cloud setting, the cloud provider is responsible to integrate these mechanisms into its infrastructure. More precisely, in an object storage application, outsourced files are stored either in multiple copies (full replication) or in an encoded format (ECC, erasure codes, etc.), together with additional metadata (CRC, checksums, etc.) that allow for the detection of data corruption. The cloud provider deploys *automatic maintenance* processes that periodically access the stored data in order to detect storage errors and upon identifying such errors leverage the redundant information to repair the damaged files.

To address the problem of large data integrity, the literature features a number of proof of storage solutions that enable data owners to efficiently verify that their data are stored correctly. Notably, Proofs of Data Possession (PDP) [3] which provide data owners with the assurance that their outsourced data has not undergone any substantial modification and, Proofs of Retrievability (PoR) [17, 19] which ensure data owners that the entirety of their outsourced files can be recovered. Yet, PoR and PDP schemes are of limited value when it comes to the audit of data reliability mechanisms: indeed, a successful PDP/PoR verification does not indicate whether a cloud provider has in place data maintenance, 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. Additionally, even though PoR schemes rely on error-correcting codes, the relevant redundancy information is not intended for typical data repair operations: error-correcting codes assist in realizing the PoR security properties by enabling the recovery of original data from accidental errors that can go undetected from the protocol. However, neither the CSP nor the data owner can use this redundancy in order to repair corrupted data outsourced to the CSP since, according to the PoR model, the CSP cannot distinguish original data from redundancy information and, the data owner lacks the means to detect any data corruption before it grows beyond repair. As a result, the cloud provider has to apply its data redundancy mechanisms and its maintenance operations independently and on top of any PoR processing performed by the data owner.

To remedy this, the scheme in [8] enables a client to repair storage errors alongside leveraging the benefits of PoR verification on erasure code–based distributed storage systems. Later on, authors in [10] proposed a new repair algorithm for the scheme in [8] that shifts the bulk of data regeneration computations on the cloud side. Nonetheless, in both schemes, the regeneration of lost or corrupted data segments by the cloud requires some interaction with the client. Both schemes thus are at odds with automatic maintenance that is a key feature of cloud storage systems.

Furthermore, in the adversarial setting of outsourced storage, there seems to be an inherent conflict between the customers' requirement for verifying data reliability mechanisms and the automatic maintenance feature of modern storage systems. On one hand, automatic maintenance based on error-correcting codes requires the storage of redundant information along with the original data and, on the other hand, to guarantee the storage of the redundancy information the CSP 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 reliability and automatic maintenance stems from the fact that the redundant information is a function of the original data itself. This property which is the underpinning of automatic maintenance can be exploited by a malicious storage provider in order to save storage space while positively meeting the data reliability verification criteria. A malicious storage provider can indeed prove the possession of redundancy information by simply computing the latter using its automatic maintenance capability without ever storing any redundancy information. Even though such a storage provider would be able to successfully respond to data reliability verification queries, akin to a PDP/PoR scheme, the actual reliability of the data would not necessarily be assured since the storage provider would fail to retrieve lost or corrupted data segments without the redundant information. Automatic maintenance that is a very efficient feature of data reliability can thus become the main enabler towards fooling data reliability verification in an adversarial setting.

3 IDEA

Our approach leverages an existing proof of data possession (PDP) scheme to assure the integrity of the original data by the CSP and it further extends it in order to verify the possession and the

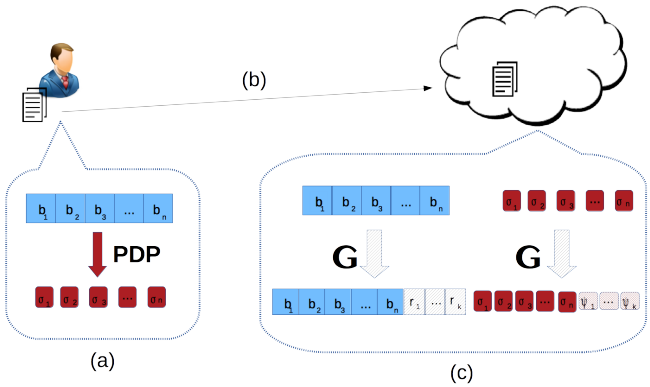

Figure 1: The process of outsourcing the computation of redundancy information and the corresponding PDP-tags to CSP: (a) The client computes the PDP-tags for the original data blocks; (b) The client outsources the data together with the PDP-tags to CSP; (c) Using G CSP applies the systematic erasure code on both data blocks and PDP-tags yielding the redundancy blocks and its corresponding PDP-tags.

correctness of the redundancy information computed by the CSP. Furthermore, as described in 2, a malicious CSP can take advantage of its automatic maintenance capability to generate the redundancy information on the fly in response to PDP queries. Hence, the last component of our scheme consist of a time-based mechanism that guarantees the storage at rest of redundancy information.

With respect to the integrity verification of the original data, our scheme leverages a proof of data possession (PDP) protocol, based on the private scheme in [19], in order to ensure the eventual detection of any attempt by a malicious CSP 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 MACs called tags that afterwards are used by the CSP to prove the storage of original data.

After receiving users' original data with the associated tags (see Figure 1(b)), the CSP generates the redundancy information based on the automatic maintenance 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 information is a linear combination of original data and the latter remains unaltered by the application of the erasure code hence its integrity verification is not affected. Concerning the correct computation as well as the integrity verification of redundancy information our scheme also leverages the PDP protocol used for original data and hence provides customers with both guarantees. More precisely, as with original data, CSP applies the same systematic linear code to the associated tags, yielding a new set of tags that are linear combinations of the original data tags. Assuming a MAC scheme that is *homomorphic* with respect to the systematic linear code used by the automatic maintenance mechanism, the tags resulting from this computation turn out to be PDP tags associated with redundancy information. 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 information that is derived from the original data based on the same linear combination. Figure 1(c) depicts the application of the systematic erasure code on both original data and redundancy information.

Thanks to the homomorphism of the underlying MAC scheme at the core of the PDP protocol and to the systematic linear code, the CSP does not need any keying material owned by the customer in order to compute the tags for the redundancy information. Furthermore, customers are able to perform PDP verification on redundancy information using these new tags. Any misconduct by the CSP regarding either the integrity of redundancy information or its proper generation will be eventually detected by the PDP checks as the malicious CSP cannot forge the computed tags.

The scheme described so far suffers from a limitation in that, a malicious CSP can take advantage of its capability to independently compute both redundancy information and the corresponding tags, paving the way for the CSP to fool reliability verification by computing in real time the responses to PDP checks on redundancy information. The last feature of our scheme thus is a countermeasure to such attacks. The basic idea underlying this countermeasure is the difference in time between simple lookup for redundancy information at rest by a legitimate CSP and the computation in real time of the same redundancy information by a malicious CSP. The solution obviously consists in ensuring that the computation of redundancy takes significantly more time than the lookup. A straightforward approach inspired by [22] relies on timing features of common cloud storage infrastructures. Based on this approach, our scheme exploits the time difference between random and sequential disk access. In order to guarantee a clear time difference between legitimate lookup and real time computation of redundancy information, our scheme scatters the redundancy information across several randomly chosen locations in the disk storage. As a result, the legitimate response to PDP queries on redundancy information only requires a few sequential disk accesses whereas a malicious CSP would have to perform multiple random accesses on top of the computation of the redundancy information itself. Malicious behavior can therefore be detected based on this time difference.

4 PROOFS OF DATA RELIABILITY

In this section we introduce the main security and functional requirements that a proof of data reliability scheme has to address and define its adversarial model.

4.1 Adversary Model

We consider the cloud storage provider (CSP) as a *rational* party whose main goal is to maximize its payoff by saving storage. More precisely, increasing its payoff with respect to storage cost is the only case where the CSP might misbehave and for all other scenarios the CSP is considered to be honest.

To further illustrate this point, assume that there is a proof of data reliability scheme in which the CSP succeeds in computing the redundancy on the fly while eluding detection. If, in order to mount its attack, the CSP is required to dedicate more storage resources than what is required when it complies with the protocol, then it is fair to argue that a *rational* CSP will prefer to follow the original protocol, rather than implement the attack.

4.2 Security Requirements

A scheme implementing a proof of data reliability oughts to meet the following requirements.

Req 0 *Correctness.* It is crucial to ensure that an *honest* CSP will always be able to generate a correct proof of data reliability. In other words, an honest CSP should always be able to pass the verification of data reliability.

Req1 *Data possession guarantee.* It is essential for any scheme of proof of data reliability to assure that the CSP cannot tamper with the outsourced data without being detected. We refer to the definition of data possession guarantee introduced in [3].

Definition 4.1. A proof of data reliability scheme guarantees *data possession* if for any adversary $\mathcal{A}$, the probability that $\mathcal{A}$ computes a correct proof of reliability for data blocks she did not have access to is negligibly close to the probability that the challenger can extract those blocks by means of a knowledge extractor.

Req 2 *Soundness of redundancy computation.* In this paper, we consider that the CSP is the party that computes the redundancy information on the outsourced data objects. Hence, it is important to ensure that the CSP performs this computation correctly.

Definition 4.2. 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.

Hence, the soundness of redundancy computation could be assured by means of *verifiable computation* [1, 13, 16] that allows the client to verify that the redundancy information has been correctly computed by an untrusted remote server.

Req 3 *At rest storage of redundancy information guarantee.* An important functionality of proofs of data reliability is ensuring that CSP stores the redundancy information, instead of computing it on the fly every time it is audited. To this effect, we propose to leverage a time threshold $T_h < \tau < T_m$ to limit the acceptable response time of CSP whereby T_h and T_m denote the response time of an honest CSP and a malicious one, respectively. For this time threshold to be meaningful, we should ensure that $\tau \ll T_m$. Therefore, we should make sure that the time it takes to generate the proof is considerably shorter than the time it takes to compute the redundancy information.

Definition 4.3. A proof of data reliability scheme guarantees the *storage at rest of redundancy information* if for any adversary $\mathcal{A}$, the probability that redundancy information can be recovered without storing it and in time duration less than τ, is negligible.

To meet this goal, we rely on *time-constrained operations*: namely, we make the memory readings that are necessary to compute the redundancy difficult and more importantly *time-consuming*. In this manner, we throttle the throughput of erasure code generation at CSP and force the latter to store the redundancy information *at rest*. The security experiment for this requirement is very similar to the one in [22] whereby authors introduce an oracle that reflects resource bounds on access to stored data. The resource-bounded oracle in this case, should also be capable of reconstructing the redundancy information.

5 POROS

In this section, we introduce a new proof of data reliability solution named POROS. We first present its main building blocks and further provide a complete description of the scheme.

5.1 Building Blocks

Erasure code. For the purposes of our proof of data reliability scheme, we consider a storage system that leverages a *systematic linear* (k,n)-MDS code [20, 24]. The generator matrix $\mathbf{G}$ of such code is of the form $[\mathbf{I}_k \mid \mathbf{M}]$, where $\mathbf{I}_k$ denotes the identity matrix of size $k \times k$. Hence the encoding procedure does not alter the original data object $\mathbf{D}$. The selected code can repair up to $d = n - k + 1$ corrupted codewords. CSP computes the redundancy information by first dividing the data object $\mathbf{D}$ into chunks and further applying the erasure code defined by $\mathbf{G}$ over each data chunk $\mathbf{d}^{(i)}$ comprising k data blocks. This yields a vector of n codewords that includes the k data blocks of chunk $\mathbf{d}^{(i)}$ followed by $n-k$ redundancy blocks.

Time-constrained proof generation at CSP. In order ensure data owners that the CSP stores the redundancy information at rest, we incorporate in our proof of reliability scheme a storage model that leverages the technical characteristics of rotational hard drives which are a common component of cloud storage infrastructure.

More precisely, our storage model assumes that all redundancy information of a given data object $\mathbf{D}$ is handled as a seperate object $\mathbf{R}$. Similarly to the permutation-based scheme in [22], CSP performs a random permutation g over the codewords that compose $\mathbf{R}$ and, stores the result on a *single* storage node without fragmentation. Disk access operations are done at the granularity of file system blocks[1]. Hence, the resulting redundancy object $\tilde{\mathbf{R}}$ is going to be stored in n contiguous file system blocks, each comprising m codewords. Notice that $\tilde{\mathbf{R}}$ does not prevent the CSP from performing automatic maintenance operations since the redundancy object $\mathbf{R}$ can be extracted from $\tilde{\mathbf{R}}$ given the inverse permutation g^{-1}.

A data owner can challenge the CSP to prove that it stores $\tilde{\mathbf{R}}$ at rest by requesting l consecutive codewords starting from a randomly chosen position in $\tilde{\mathbf{R}}$. Assuming a random permutation g that uniformly distributes the codewords in block $\mathbf{r}^{(i)}$ over the blocks in $\tilde{\mathbf{R}}$, a compliant CSP has to perform one seek operation and then access $\lceil l/m \rceil$ file system blocks sequentially. On the contrary, a

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

malicious CSP that does not store $\tilde{\mathbf{R}}$ has to perform up to l seek operations on the storage nodes that store the original data object $\mathbf{D}$ in order to access the corresponding data chunks $\mathbf{d}^{(i)}$, transmit these chunks over its internal network and, apply the erasure code for each of the l requested codewords.

The time threshold τ is thus defined as a function of time T_h that an *honest* CSP takes to generate the proof and the response time T_m of a malicious CSP who does not store $\tilde{\mathbf{R}}$ and thus has to recompute the redundancy. Preferably, τ should satisfy the inequality $T_h < \tau \ll T_m$, implying that the time required for the proof generation is much shorter than the time needed to compute the redundancy. In the next section we present our solution named POROS in detail.

5.2 POROS Description

POROS runs in three phases:

Initialization. The CSP and the client generate their parameters and keying material by invoking the following algorithms:

$\mathsf{Setup}(1^\lambda, n, k) \rightarrow \mathsf{param}$: Given security parameter λ, integers n and k such that $n > k$, algorithm Setup first selects a large prime number p (typically $|p| = 160$), a (k, n) generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{M}]$ of a linear erasure code in $\mathbb{F}_p$, a pseudo-random generator $\mathsf{prg} : \mathbb{F}_p \times \mathbb{F}_p \rightarrow \{0,1\}^*$ and the corresponding random seed η. Algorithm Setup then terminates its execution by returning the public parameters: $\mathsf{param} = (p, n, k, \mathbf{G}, \mathsf{prg}, \eta)$.

$\mathsf{KeyGen}(\mathsf{param}) \rightarrow (\mathsf{param}_u, \mathsf{param}_v, \mathsf{sk})$: On input of public parameters $\mathsf{param} = (p, n, k, \mathbf{G} = [\mathbf{I}_k \mid \mathbf{M}], \mathsf{prg}, \eta)$, algorithm KeyGen first picks a pseudo-random function $\mathsf{prf} : \mathbb{F}_p \times \{0,1\}^* \rightarrow \mathbb{F}_p$, defines the upload parameters $\mathsf{param}_u = (p, n, k, \mathsf{prf})$ and the verification parameters $\mathsf{param}_v = (p, \mathbf{M}, \mathsf{prg}, \eta, \mathsf{prf})$, and finally selects a secret key $\mathsf{sk} = (\alpha, \beta) \in \mathbb{F}_p \times \mathbb{F}_p$ randomly.

Outsourcing. The client in this phase preprocesses and uploads the file; the cloud server on the other hand takes the uploaded file, processes it further by applying the agreed-upon erasure code and stores it together with the resulting redundancy. More formally, this phase is defined by two algorithms:

$\mathsf{Upload}(\mathsf{param}_u, \mathsf{sk}, \mathbf{D}) \rightarrow (\mathsf{param}_d, \tilde{D})$: On input of upload parameters $\mathsf{param}_u = (p, n, k, \mathsf{prf})$, secret key $\mathsf{sk} = (\alpha, \beta)$ and data object $\mathbf{D}$, algorithm Upload (run by the client) first splits data object $\mathbf{D}$ into chunks $\mathbf{d}^{(i)}$ each composed of k blocks $(d_1^{(i)}, \ldots, d_k^{(i)})$ such that for all $1 \leq j \leq k$, $d_j^{(i)}$ is an element in $\mathbb{F}_p$[2]. Thereafter, algorithm Upload computes for each block $d_j^{(i)}$ a *linearly homomorphic* MAC[3]:

$$\sigma_j^{(i)} = \alpha d_j^{(i)} + \mathsf{prf}(\beta, \mathsf{fid} \parallel (i-1)k + j)$$

Where fid is $\mathbf{D}$'s unique identifier.

Finally algorithm Upload outputs the file parameters $\mathsf{param}_d = (\mathsf{fid}, n, k, L)$ wherein L is the number of chunks $\mathbf{d}^{(i)}$, and the authenticated version $\tilde{D}$ of the original data object $\mathbf{D}$ whereby:

$$\tilde{D} = \{\mathbf{d}^{(i)} = (d_1^{(i)}, \ldots, d_k^{(i)}); \boldsymbol{\sigma}^{(i)} = (\sigma_1^{(i)}, \ldots, \sigma_k^{(i)});\ \text{s.t. } 1 \leq i \leq L\}$$

$\mathsf{Store}(\mathsf{param}, \mathsf{param}_d, \tilde{D}) \rightarrow (\tilde{D}, \tilde{R})$: Given parameters $\mathsf{param} = (p, n, k, \mathbf{G}, \eta)$, file parameters $\mathsf{param}_d = (\mathsf{fid}, n, k, L)$ and the uploaded file $\tilde{D} = \{\mathbf{d}^{(i)} = (d_1^{(i)}, \ldots, d_k^{(i)}); \boldsymbol{\sigma}^{(i)} = (\sigma_1^{(i)}, \ldots, \sigma_k^{(i)});\ \text{s.t. } 1 \leq i \leq L\}$, algorithm Store (run by the CSP) proceeds as follows:

(1) *Erasure code computation:* Algorithm Store applies the erasure code defined by matrix $\mathbf{G}$ to each chunk $\mathbf{d}^{(i)}$ by computing the matrix multiplication $\mathbf{d}^{(i)}\mathbf{G} = \mathbf{d}^{(i)}[\mathbf{I}_k \mid \mathbf{M}]$. This yields vector $(\mathbf{d}^{(i)} \mid \mathbf{r}^{(i)})$ where $\mathbf{r}^{(i)} = (r_1^{(i)}, \ldots, r_{n-k}^{(i)}) = \mathbf{d}^{(i)}\mathbf{M}$ defines the redundancy blocks (i.e. the erasure code) of chunk $\mathbf{d}^{(i)}$. The set of redundancy blocks for all chunks is defined as follows:

$$\mathbf{R} = (\mathbf{r}^{(1)} \mid \ldots \mid \mathbf{r}^{(L)}) = (\mathbf{r}_1^{(1)}, \ldots, \mathbf{r}_{n-k}^{(1)}, \ldots, \mathbf{r}_1^{(L)}, \ldots, \mathbf{r}_{n-k}^{(L)})$$

(2) *Erasure code authentication:* Algorithm Store computes the vector of homomorphic MACs $\boldsymbol{\sigma}^{(i)}\mathbf{G} = \boldsymbol{\sigma}^{(i)}[\mathbf{I}_k \mid \mathbf{M}]$. In what follows, we denote by vector $\boldsymbol{\psi}^{(i)} = (\psi_1^{(i)}, \ldots, \psi_{n-k}^{(i)})$ the result of the matrix multiplication $\boldsymbol{\sigma}^{(i)}\mathbf{M}$ and by $\boldsymbol{\Psi}$ the vector:

$$(\boldsymbol{\psi}^{(1)} \mid \ldots \mid \boldsymbol{\psi}^{(L)}) = (\boldsymbol{\psi}_1^{(1)}, \ldots, \boldsymbol{\psi}_{n-k}^{(1)}, \ldots, \boldsymbol{\psi}_1^{(L)}, \ldots, \boldsymbol{\psi}_{n-k}^{(L)})$$[4]

(3) *Erasure code shuffling:* Using the pseudo-random generator prg, seed η and file identifier fid algorithm Store produces a random permutation[5] $g : [L] \times [n-k] \rightarrow [(n-k)L]$ and then permute the erasure code redundancy $\mathbf{R}$ and the corresponding homomorphic MACs $\boldsymbol{\Psi}$. More precisely, if we denote $\tilde{\mathbf{R}} = (\tilde{\mathbf{r}}_1, \ldots, \tilde{\mathbf{r}}_{(n-k)L})$ the redundancy vector after permutation, then redundancy block $\mathbf{r}_j^{(i)}$ is mapped to redundancy block $\tilde{\mathbf{r}}_{g(i,j)}$. Similarly, if we denote $\tilde{\boldsymbol{\Psi}} = (\tilde{\boldsymbol{\psi}}_1, \ldots, \tilde{\boldsymbol{\psi}}_{(n-k)L})$ the homomorphic MACs' vector after permutation, then MAC $\boldsymbol{\psi}_j^{(i)}$ is mapped to MAC $\tilde{\boldsymbol{\psi}}_{g(i,j)}$.

Algorithm Store concludes its execution by outputting $\tilde{D}$ and $\tilde{R} = (\tilde{\mathbf{R}}, \tilde{\boldsymbol{\Psi}})$.

Verification. In this phase, the client initiates a *challenge-response* protocol with the CSP to ascertain compliance (or the lack thereof). Accordingly, the CSP produces a proof of data reliability and the client verifies its correctness. In accordance with previous work on PoR and PDP, this phase consists of calling three algorithms:

$\mathsf{Challenge}(\tau, \mathsf{param}_d) \rightarrow \mathsf{chal}$: Provided with time threshold τ and file parameters $\mathsf{param}_d = (\mathsf{fid}, n, k, L)$, algorithm Challenge (run by the client) chooses an integer l, generates a vector $\boldsymbol{\nu}$ of l random elements ν_i in $\mathbb{F}_p^*$, generates a vector $\boldsymbol{i_d}$ of l random indices corresponding to data blocks of $\mathbf{D}$, picks one random index $1 \leq i_r \leq (n-k)L - l$, and accordingly sets the challenge to the tuple: $\mathsf{chal} = (\tau, \mathsf{fid}, \boldsymbol{i_d}, i_r, \boldsymbol{\nu})$.

$\mathsf{Prove}(\mathsf{chal}) \rightarrow \pi$: On receiving challenge $\mathsf{chal} = (\tau, \mathsf{fid}, \boldsymbol{i_d}, i_r, \boldsymbol{\nu})$, algorithm Prove (run by the CSP) first retrieves the authenticated data object $\tilde{D}$ and the corresponding authenticated redundancy $\tilde{R}$ that match identifier fid.

Thereupon, algorithm Prove processes data object $\tilde{D}$ as follows:

[2] This means that the size of $d_j^{(i)}$ in bits is $\log(p)$.

[3] By construction $d_j^{(i)}$ is the $((i-1)k + j)^{\text{th}}$ block in data object $\mathbf{D}$.

[4] Note that if we denote $\mathbf{M} = [\mathbf{M}_1 \mid \ldots \mid \mathbf{M}_{n-k}]$ such that each $\mathbf{M}_i$ is a column vector of size k, then $\psi_j^{(i)}$ corresponds to the inner product $\boldsymbol{\sigma}^{(i)} \cdot \mathbf{M}_j$ and by construction $\psi_j^{(i)}$ is the linearly-homomorphic MAC of the inner product $\mathbf{b}^{(i)} \cdot \mathbf{M}_j$.

[5] Algorithm Store could use pseudo-random generator prg, seed η and file identifier fid to produce a pair (δ, γ) such that $\gcd(\delta, (n-k)L) = 1$, and then shuffle the erasure code redundancy $\mathbf{R}$ and the corresponding homomorphic MACs $\boldsymbol{\Psi}$ using the following transformation for all $1 \leq i \leq L$ and $1 \leq j \leq (n-k)$: $g(i, j) = \delta((i-1)(n-k)+j)+\gamma \mod (n-k)L$.

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

$$\tilde{d} = \tilde{\mathbf{d}} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{d}_j \nu_j \tag{1}$$

$$\tilde{\sigma} = \tilde{\boldsymbol{\sigma}} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{\sigma}_j \nu_j \tag{2}$$

In the same manner, algorithm Prove processes the redundancy $\tilde{R}$:
(1) It reads l consecutive redundancy blocks starting from block $\tilde{r}_{i_r}$. Let $\tilde{\mathbf{r}}$ denote the l consecutive redundancy blocks $(\tilde{r}_{i_r}, ..., \tilde{r}_{(i_r+l-1)})$.
(2) It reads the l consecutive homomorphic MACs associated with redundancy blocks $\tilde{r}_{(i_r+j-1)}, 1 \le j \le l$. Let $\tilde{\boldsymbol{\psi}} = (\tilde{\psi}_{i_r}, ..., \tilde{\psi}_{(i_r+l-1)})$ denote these MACs.
(3) It computes the inner products

$$\tilde{r} = \tilde{\mathbf{r}} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{r}_{(i_r+j-1)} \nu_j \tag{3}$$

$$\tilde{\psi} = \tilde{\boldsymbol{\psi}} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{\psi}_{(i_r+j-1)} \nu_j \tag{4}$$

Finally, algorithm Prove outputs the proof $\pi = (\tilde{d}, \tilde{\sigma}, \tilde{r}, \tilde{\psi})$.

Verify$(\mathsf{sk}, \mathsf{param}_v, \mathsf{chal}, \pi) \rightarrow$ accept or reject : On input of secret key $\mathsf{sk} = (\alpha, \beta)$, verification parameters $\mathsf{param}_v = (p, \mathbf{M}, \mathsf{prg}, \eta, \mathsf{prf})$, challenge $\mathsf{chal} = (\tau, \mathsf{fid}, \boldsymbol{i_d}, i_r, \boldsymbol{\nu})$ and proof $\pi = (\tilde{d}, \tilde{\sigma}, \tilde{r}, \tilde{\psi})$, algorithm Verify (run by the client) performs the following checks:
- *Response time verification:* It first checks whether the response time of the server was under time threshold τ. If not algorithm Verify outputs reject; otherwise it executes the next step.
- *Data possession verification:* Given vector $\boldsymbol{\nu} = (\nu_1, ..., \nu_l)$ and vector $\boldsymbol{i_d} = (i_{d1}, ..., i_{dl})$ algorithm Verify verifies whether

$$\tilde{\sigma} = \alpha \tilde{d} + \sum_{j=1}^{l} \nu_j \mathsf{prf}(\beta, \mathsf{fid} \,\|i_{dj}) \tag{5}$$

If it is not the case, algorithm Verify returns reject; otherwise it moves onto verifying the integrity of the redundancy.
- *Redundancy possession verification:* Algorithm Verify uses the pseudo-random generator prg, seed η and file identifier fid to get the shuffling function g, and then for all $1 \le j \le l$ it computes the shuffling function preimage $(x_j, y_j) = g^{-1}(i_r + j - 1)$. Finally, having matrix $\mathbf{M} = [\mathbf{M}_1 \mid ... \mid \mathbf{M}_{(n-k)}]$, algorithm Verify checks whether the following equation holds:

$$\tilde{\psi} = \alpha \tilde{r} + \sum_{j=1}^{l} \nu_j \mathbf{M}_{y_j} \cdot \mathbf{prf}^{(x_j)} \tag{6}$$

wherein for all $1 \le j \le l$:

$$\mathbf{prf}^{(x_j)} = (\mathsf{prf}(\beta, \mathsf{fid} \parallel (x_j - 1)k + 1), ..., \mathsf{prf}(\beta, \mathsf{fid} \parallel x_j k))$$

If so, algorithm Verify outputs accept; otherwise it returns reject.

6 SECURITY EVALUATION

6.1 Security Analysis

Req 0 *Correctness.* In order to demonstrate that POROS does not yield false positives, we first show that Equations 5 and 6 always hold when algorithm Prove is correctly run, then we argue that if time threshold τ is correctly tuned then the probability of wrongly accusing the CSP of misbehavior is close to none.

Upon invocation, algorithm Prove first reads l consecutive blocks $\tilde{\mathbf{d}} = (\tilde{d}_1, ..., \tilde{d}_l)$ and their corresponding MACs $\tilde{\boldsymbol{\sigma}} = (\tilde{\sigma}_1, ..., \tilde{\sigma}_l)$, whereby $\tilde{d}_1$ is the i_f^{th} block in the original data object $\mathbf{D}$. By the definition of homomorphic MACs $\tilde{\sigma}_j$, the following equality ensues:

$$\tilde{\sigma}_j = \alpha \tilde{d}_j + \mathsf{prf}(\beta, \mathsf{fid} \| i_{dj}), \; \forall 1 \le j \le l \tag{7}$$

Moreover, algorithm Prove scans l redundancy blocks $\tilde{\mathbf{r}} = (\tilde{r}_{i_r}, ..., \tilde{r}_{(i_r+l-1)})$ together with their corresponding MACs $\tilde{\boldsymbol{\psi}} = (\tilde{\psi}_{i_r}, ..., \tilde{\psi}_{(i_r+l-1)})$. Note that for all $1 \le j \le l$ redundancy block $\tilde{r}_{i_r+j-1}$ corresponds to redundancy block $r_{y_j}^{(x_j)} = \mathbf{M}_{y_j} \cdot \mathbf{b}^{(x_j)}$ and MAC $\tilde{\psi}_{i_r+j-1}$ corresponds to $\psi_{y_j}^{(x_j)} = \mathbf{M}_{y_j} \cdot \boldsymbol{\sigma}^{(x_j)}$ whereby $(x_j, y_j) = g^{-1}(i_r + j - 1)$ and $\mathbf{M}_{y_j}$ is the y_j^{th} column of the linear code matrix $\mathbf{M}$. Therefore, the following equality always holds.

$$\tilde{\psi}_{i_r+j-1} = \mathbf{M}_{y_j} \cdot (\alpha \mathbf{d}^{(x_j)} + \mathbf{prf}^{(x_j)}) = \alpha \tilde{r}_{i_r+j-1} + \mathbf{M}_{y_j} \cdot \mathbf{prf}^{(x_j)} \tag{8}$$

Where $\mathbf{prf}^{(x_j)} = (\mathsf{prf}(\beta, \mathsf{fid} \parallel (x_j - 1)k + 1), ..., \mathsf{prf}(\beta, \mathsf{fid} \parallel x_j k))$.

Finally, algorithm Prove finishes its execution by computing four inner products. These inner products are computed as follows:

$$\tilde{d} = \tilde{\mathbf{d}} \cdot \nu \; ; \; \tilde{\sigma} = \tilde{\boldsymbol{\sigma}} \cdot \nu \; ; \; \tilde{r} = \tilde{\mathbf{r}} \cdot \nu \; ; \; \tilde{\psi} = \tilde{\boldsymbol{\psi}} \cdot \nu$$

Where $\boldsymbol{\nu} = (\nu_1, ..., \nu_l)$ 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:

$$\tilde{\sigma} = \alpha \tilde{d} + \sum_{j=1}^{l} \nu_j \mathsf{prf}(\beta, \mathsf{fid} \parallel i_{dj})$$

$$\tilde{\psi} = \alpha \tilde{r} + \sum_{j=1}^{l} \nu_j \mathbf{M}_{y_j} \cdot \mathbf{prf}^{(x_j)}$$

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 Prove correctly, then it will pass the verification so long as *its response time is smaller than time threshold* τ.

Req1 *Data possession guarantee.* As the proposed POROS solution relies on the use of linearly-homomorphic MACs defined in the privately verifiable PoR scheme in [19], we can directly show that a large portion of the outsourced file is stored intact and thus the data possession guarantee is satisfied.

Req 2 *Soundness of redundancy computation.* Following Theorem 4.1 in [19], if the prf is secure, then no adversary will cause a verifier to accept in a proof of data reliability instance, except by responding with ψ and σ, that are computed correctly. Indeed, because in POROS, redundancy information is computed by applying

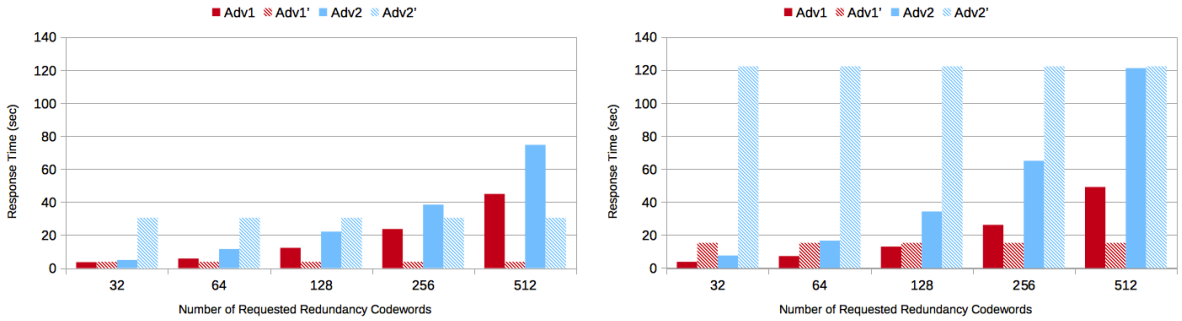

Figure 2: Response times of adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ for different challenge sizes l; data object size of 4GB (left) vs. 16GB (right).

Table 1: Response times of an honest CSP for deferent challenge sizes l.

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

a linear combination over original data blocks and consequently their MACs should derive from the MACs of the data.

Req 3 *At rest storage of redundancy information guarantee.* Similarly to [22], **Req 3** is met as long as $\tau \ll T_m$. $T_m = min(T_{m_1}, T_{m_2})$ is the response time of a malicious CSP where T_m is defined as the minimum of the following values:

- T_{m_1}: the response time of a malicious CSP who stores the redundancy information in its original order (i.e. without permutation). $T_{m_1} = lT_{Seek} + lT_{SeqRead}(1)$
- T_{m_2}: the response time of a malicious CSP who stores the data object **D**, only. $T_{m_2} = lT_{Seek} + lT_{SeqRead}(k) + lT_{Encode}$

In the above equations, l is the number of sequential redundancy blocks $\tilde{r}$ requested in a POROS challenge, T_{Seek} is the time required for a seek operation on the hard drive, $T_{SeqRead}(n)$ is the required time to read n data blocks sequentially from the hard disk, T_{Encode} is the time required to apply the erasure-code defined by the generator matrix **G** over a data chunk and, k the number of blocks that comprise a data chunk. Intuitively, T_{m_1} should be less than T_{m_2}.

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

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

To conclude **Req 3** is met as long as the time threshold τ (and therewith l) is tuned such that it fulfills

$$T_{Seek} + T_{SeqRead}(l) \leq \tau < lT_{Seek} + lT_{SeqRead}(1)$$

Section 6.2 provides some hints on the order of τ through an experimental study.

6.2 Experimental evaluation

We have performed an experimental evaluation of POROS' Prove algorithm in order to assess the time-constrained proof generation at CSP. The goal is to compare the response time required for the generation off a legitimate proof to the one required for the generation of a cheating response by a malicious CSP. We implemented our prototype in Python 3.5 and we used the zfec[6] library in order to apply the erasure code to some test files[7].

We consider two types of adversary that deviate from the protocol in different ways. The cloud is required to read l consecutive redundancy codewords from the redundancy object $\tilde{\mathbf{R}}$ in the order defined by the permutation g and return them to the verifier. Adversary $\mathcal{A}_1$, stores the original redundancy object **R** in its original order. $\mathcal{A}_1$ attempts to elude detection by seeking on the hard disk the requested redundancy codewords in order to produce the response. As regards to adversary $\mathcal{A}_2$, she 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 g; 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 $\mathcal{A}'_1$ will load the original redundancy object $\tilde{\mathbf{R}}$ within the RAM and subsequently compose her response according to g. On the other hand, $\mathcal{A}'_2$ will load the whole data object **D** to the RAM to further compute the $\tilde{\mathbf{R}}$ and respond with the required codewords. 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 1 and 2 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 1 presents the response time of an honest CSP which stores the same data objects. The redundancy is computed using a systematic linear $(256, 288)$-MDS code that operates over 64-bit codewords yielding 32 redundancy codewords. In order to apply the code, the 4GB data

[6] https://pypi.python.org/pypi/zfec

[7] 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, a 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.

Table 2: Disadvantage of adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ relative to an honest CSP.

Object Size	**D** (4GB), $\tilde{\mathbf{R}}$ (512MB)					**D** (16GB), $\tilde{\mathbf{R}}$ (2GB)				
Challenge size l	32	64	128	256	512	32	64	128	256	512
$T_{\mathcal{A}_1}/T_h$	167.51	180.39	178.66	182.39	174.38	140.56	272.77	461.76	553.84	552.24
$T_{\mathcal{A}_2}/T_h$	232.19	546.88	1035.93	1459.14	1399.05	282.93	626.21	1218.56	2359.08	4145.43

object **D** is divided into data chunks $\mathbf{d}^{(i)}$ of size 2KB each. The redundancy object **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 codewords. At this point, the honest CSP computes $\tilde{\mathbf{R}}$ using the random permutation g, $\mathcal{A}_1$ stores **R** without permuting it and, $\mathcal{A}_2$ discards the redundancy object.

We observe that the response time of an honest CSP 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 CSP 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 **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 **D** and further apply the erasure code in order to produce the response. Similarly to the analysis in [22], in the case of the 4GB data object, when the size of the challenge l is larger than 32 redundancy codewords, it is faster for $\mathcal{A}_1$ to load the whole **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 codewords.

In Table 2 we show the ratio between the response time of a malicious adversary and the one of a legitimate CSP. For example, for a 4GB file and a challenge of size 128, $\mathcal{A}_1$ is 178 times slower than an honest CSP.

To conclude, our experimental study confirms that by storing redundancy information as a single permuted object, separately from original data, a rational CSP 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, τ can set to be quite close to the lower bound defined by the time an honest CSP would take to compute the POROS response for a given file.

7 RELATED WORK

Bowers et al. [8] propose HAIL, which provides a high availability and integrity layer for cloud storage. In order to guarantee data retrievability among distributed storage servers, HAIL uses erasure codes on the single and multiple server layers respectively, enabling a user to detect and repair corruption of her data. The work in [10] redesigns parts of [8] in order to achieve a more efficient repair phase that shifts the bulk computations to the cloud side. In [11], Chen et al. present a remote data checking scheme for network-coding-based distributed storage systems that minimizes the communication overhead of the repair component compared to erasure coding-based approaches. The work in [18] extends the scheme in [11] improving the repair mechanism in order to reduce the computation cost for the client and introducing a third party auditor. Based on the introduction of this new entity, the authors in [21] design a network-coding-based PoR scheme in which the repair mechanism is executed between the cloud provider and the third party auditor without any interaction with the client. All the above schemes however share a common system model where the client initially encodes her data and then outsources it to the cloud. Moreover, when corruption is detected the cloud cannot repair autonomously because either it expects some input from another entity or all computations are performed by the client. In POROS, the cloud is enabled to perform automatic maintenance operations and thus it can repair corrupted data autonomously. In addition, even the initial encoding of the data is performed by the cloud.

In [9], Bowers et al. propose a erasure code based protocol that enables a client to verify that her encoded data is stored at multiple servers so it can be regenerated. Similarly to our solution, this scheme relies on technical characteristics of rotational hard drives in order to set a time-threshold for the cloud servers to respond to a read request for a set of data blocks. Additionally, the scheme also enables the outsourcing of the data encoding to the cloud provider as well as automatic maintenance operations without any interaction with the client. However, in contrast to POROS, the challenge verification requires that a copy of the encoded data is stored locally at the user.

Except for the schemes that rely on coding for the maintenance of the data, the literature features a number of solutions that use replication in order to provide data reliability and thus are not directly comparable to POROS. Curtmola et al. proposed in [12] a multi-replica PDP, which extends the PDP scheme in [3] and enables the client to verify that the cloud provider stored at least t replicas of her data. The authors in [5, 6] propose a multi-replica dynamic PDP scheme that enables clients to update/insert selected data blocks and to verify multiple replicas of their outsourced files. In [15], Etemad et al. extend the dynamic PDP scheme in [14] in order to transparently support replication in distributed cloud storage systems. Likewise related work in coding based protocols, the above schemes require that the client generates the redundancy by constructing the replicas locally before outsourcing to the cloud.

Finally, in [2] Armknecht et. al propose a multi-replica PoR scheme that outsources the construction of the replicas to the cloud. Nonetheless, due to the underlying PoR scheme, clients in [2] have to encode their data before uploading it to the cloud.

8 CONCLUSION

In this paper, we have introduced a new proof of data reliability solution named POROS that enables a cloud customer to efficiently verify that the cloud server stores her outsourced data correctly and additionally that it complies with the claimed data reliability and availability guarantees. Running POROS protocol, a client is assured that the cloud server actually stores both the original data and the

corresponding redundancy information. Thanks to the combination of PDPs with time-constrained operations, a rationale cloud server would not compute redundancy blocks upon request but instead would store them at rest. Contrary to existing solutions, 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.

9 ACKNOWLEDGMENTS

The authors thank Hamdi Ammar, Cedric Osornio-Gleason and Sejal Jain for implementing the POROS protocol. The authors also thank Loukas Lazos and Li Li for helpful discussions on the topic of data reliability. Finally, the authors would like to thank the anonymous reviewers for their suggestions and feedback.

REFERENCES

[1] Shweta Agrawal and Dan Boneh. 2009. Homomorphic MACs: MAC-Based Integrity for Network Coding. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security (ACNS '09)*. 292–305.

[2] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. 2016. Mirror: Enabling Proofs of Data Replication and Retrievability in the Cloud. In *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, 1051–1068.

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

[4] M. Azraoui, K. Elkhiyaoui, R. Molva, and M. Önen. 2014. StealthGuard: Proofs of Retrievability with Hidden Watchdogs. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS)* . 239–256.

[5] Ayad F. Barsoum and M. Anwar Hasan. 2012. Integrity Verification of Multiple Data Copies over Untrusted Cloud Servers. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012) (CCGRID '12)*. 829–834.

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

[7] M. Blaum, J. Brady, J. Bruck, and J. Menon. 1994. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*. 245–254.

[8] Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A High-availability and Integrity Layer for Cloud Storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. 187–198.

[9] Kevin D. Bowers, Marten van Dijk, Ari Juels, Alina Oprea, and Ronald L. Rivest. 2011. How to Tell if Your Cloud Files Are Vulnerable to Drive Crashes. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. 501–514.

[10] Bo Chen, Anil Kumar Ammula, and Reza Curtmola. 2015. Towards Server-side Repair for Erasure Coding-based Distributed Storage Systems. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)*. 281–288.

[11] Bo Chen, Reza Curtmola, Giuseppe Ateniese, and Randal Burns. 2010. Remote Data Checking for Network Coding-based Distributed Storage Systems. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW '10)*. 31–42.

[12] Reza Curtmola, Osama Khan, Randal C. Burns, and Giuseppe Ateniese. 2008. MR-PDP:Multiple-Replica ProvableDataPossession. In *ICDCS*. 411ÂŰ420.

[13] Kaoutar Elkhiyaoui, Melek Önen, Monir Azraoui, and Refik Molva. 2016. Efficient Techniques for Publicly Verifiable Delegation of Computation. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. 119–128.

[14] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2009. Dynamic Provable Data Possession. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. 213–222.

[15] Mohammad Etemad and Alptekin Küpçü. 2013. Transparent, Distributed, and Replicated Dynamic Provable Data Possession. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS'13)*. 1–18.

[16] Dario Fiore and Rosario Gennaro. 2012. Publicly Verifiable Delegation of Large Polynomials and Matrix Computations, with Applications. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 501–512.

[17] Ari Juels and Burton S. Kaliski, Jr. 2007. Pors: Proofs of Retrievability for Large Files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. 584–597.

[18] Anh Le and Athina Markopoulou. 2012. NC-Audit: Auditing for network coding storage. In *Proceedings of International Symposium on Network Coding*. 155–160.

[19] Shacham, Hovav and Waters, Brent. 2008. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT)*. 90–107.

[20] Changho Suh and K. Ramchandran. 2011. Exact-Repair MDS Code Construction Using Interference Alignment. *IEEE Trans. Inf. Theor.* 57, 3 (March 2011), 1425–1442.

[21] Tran Phuong Thao and Kazumasa Omote. 2016. ELAR: Extremely Lightweight Auditing and Repairing for Cloud Security. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC '16)*. 40–51.

[22] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. 2012. Hourglass Schemes: How to Prove That Cloud Files Are Encrypted. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 265–280.

[23] Yunnan Wu and Alexandros G. Dimakis. 2009. Reducing Repair Traffic for Erasure Coding-based Storage via Interference Alignment. In *Proceedings of the 2009 IEEE International Conference on Symposium on Information Theory - Volume 4 (ISIT'09)*. 2276–2280.

[24] Chaoping Xing and San Ling. 2003. *Coding Theory: A First Course.* Cambridge University Press, New York, NY, USA.

A MULTIPLE-CHALLENGE POROS

In order to increase clients confidence in CSP's good behavior, it is desirable to make the CSP execute multiple instances of POROS. A straightforward approach to achieve this, would be to have the client send multiple challenges to CSP. The main caveat of such a solution is that CSP can run several instances of algorithm Prove in parallel, which renders less effective the function that throttles ECC-throughput. To counter this issue, we could turn to a sequential protocol whereby the client would wait for the server's response to the current challenge before transmitting the next one. to transmit the second challenge, and so on and so forth. While this approach forces the server to generate the proofs iteratively, it increases the number of interactions between the client and the CSP, which in turn comes at the expense of bandwidth and throughput.

To address these shortcomings, we propose that the client initiates the protocol by sending a single challenge termed hereafter *initialization challenge*. This challenge will be generated exactly in the same way as the challenge in the basic version of POROS (cf. Section 5.2). Subsequent challenges however will be produced as a function of the proofs to preceding challenges. More specifically, the challenge in iteration $t + 1$, for instance, is computed as a function of the proof that the cloud server generated as a response to the challenge in iteration t. In this manner, we devise a multiple-challenge version of POROS that (i) keeps the communication between the client and CSP minimal (i.e. there are only two rounds of communication); (ii) and induces the server to generate the proofs iteratively.

A.1 Description

Multiple-challenge POROS runs as following:

The *Initialization* and the *Outsourcing* phases run in the exact same way as their counterparts in Section 5.2. Additionally, the client chooses one pseudorandom generator $\mathrm{prg}_{\mathrm{d}} : \{0,1\}^* \rightarrow [kL]^l$ and sends this one to the cloud before the *Verification* phase starts.

The main reason to have this prg_d is to reduce the size of the challenge chal. We now describe the *Verification* phase.

Challenge$(\tau, \text{param}_d) \rightarrow$ chal : On input of time threshold τ and data object parameters $\text{param}_d = (\text{fid}, n, k, L)$, algorithm Challenge (run by the client) first selects an integer s which specifies the number of iterations that the server is required to go through, then it generates a random vector $\boldsymbol{\nu} = (\nu_1, ..., \nu_l)$in $\mathbb{F}_p^{*\,l}$, chooses a vector of random seeds $\boldsymbol{\eta_d} = (\eta_d^{(1)}, ..., \eta_d^{(s)})$, and one random index $1 \leq i_{r_1} \leq (n-k)L - l$. Finally, algorithm Challenge defines the initialization challenge as: chal $= (\tau, \text{fid}, s, \boldsymbol{\eta_d}, i_{r_1}, \boldsymbol{\nu})$.

Prove(chal) $\rightarrow \pi$: Given initialization challenge chal $= (\tau, \text{fid}, s, \boldsymbol{\eta_d}, i_{r_1}, \boldsymbol{\nu})$, algorithm Prove (run by the CSP) fetches the authenticated data object $\tilde{D}$ and the authenticated redundancy object $\tilde{R}$.

Next algorithm Prove processes authenticated data object $\tilde{D}$ by executing the following operations s times. For ease of exposition, we assume that algorithm Prove is at the t^{th} iteration:

(1) Algorithm Prove reads l blocks whose indices result from $\text{prg}_\text{d}(\eta_d^{(t)})$ Henceforth, we denote these blocks $\tilde{\mathbf{d}}^{(t)} = (\tilde{d}_1^{(t)}, ..., \tilde{d}_l^{(t)})$.
(2) It reads the l MACs associated with blocks $\tilde{\mathbf{d}}^{(t)}$. From now on we denote these MACs $\tilde{\boldsymbol{\sigma}}^{(t)} = (\tilde{\sigma}_1^{(t)}, ..., \tilde{\sigma}_l^{(t)})$.
(3) It computes the inner products

$$\tilde{d}^{(t)} = \tilde{\mathbf{d}}^{(t)} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{d}_j^{(t)} \nu_j$$

$$\tilde{\sigma}^{(t)} = \tilde{\boldsymbol{\sigma}}^{(t)} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{\sigma}_j^{(t)} \nu_j$$

(4) It goes back to step 1.

Algorithm Prove processes the redundancy $\tilde{R}$ in the same way it processed authenticated data object $\tilde{D}$. More precisely it executes the following tasks s times.

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

$$\tilde{r}^{(t)} = \tilde{\mathbf{r}}^{(t)} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{r}_{(i_{r_t}+j-1)} \nu_j$$

$$\tilde{\psi}^{(t)} = \tilde{\boldsymbol{\psi}}^{(t)} \cdot \boldsymbol{\nu} = \sum_{j=1}^{l} \tilde{\psi}_{(i_{r_t}+j-1)} \nu_j$$

(4) It computes the random value $I_{r_{t+1}} = \text{prg}(\tilde{\psi}^{(t)}, \tilde{r}^{(t)})$ and sets the new index $i_{r_{t+1}}$ for the next challenge to the first $\log((n-k)L-l)$ bits of $I_{r_{t+1}}$.
(5) It goes back to step 1.

Algorithm Prove terminates by returning the proof $\pi = (\tilde{\mathbf{d}}, \tilde{\boldsymbol{\sigma}}, \tilde{\mathbf{r}}, \tilde{\boldsymbol{\psi}})$, such that:

$$\tilde{\mathbf{d}} = (\tilde{d}^{(1)}, ..., \tilde{d}^{(s)}) \; ; \; \tilde{\boldsymbol{\sigma}} = (\tilde{\sigma}^{(1)}, ..., \tilde{\sigma}^{(s)})$$

$$\tilde{\mathbf{r}} = (\tilde{r}^{(1)}, ..., \tilde{r}^{(s)}) \; ; \; \tilde{\boldsymbol{\psi}} = (\tilde{\psi}^{(1)}, ..., \tilde{\psi}^{(s)})$$

Verify(sk, param_v, chal, π) $\rightarrow$ accept **or** reject : On input of secret key sk $= (\alpha, \beta)$, verification parameters $\text{param}_v = (p, \mathbf{M}, \text{prg}, s, \text{prf})$, initialization challenge chal $= (\tau, \text{fid}, i_{d_1}, i_{r_1}, \boldsymbol{\nu}, s)$ and proof $\pi = (\tilde{\mathbf{d}}, \tilde{\boldsymbol{\sigma}}, \tilde{\mathbf{r}}, \tilde{\boldsymbol{\psi}})$, algorithm Verify (run by the client) performs the following operations:

– *Response Time verification.* It first verifies whether the response time of the sever was under time threshold τ. If not algorithm Verify outputs reject. Otherwise, algorithm Verify moves onto checking the integrity of the outsourced data.
– *Data possession verification.* Given vectors $\tilde{\mathbf{d}} = (\tilde{d}^{(1)}, ..., \tilde{d}^{(s)})$, and $\tilde{\boldsymbol{\sigma}} = (\tilde{\sigma}^{(1)}, ..., \tilde{\sigma}^{(s)})$, algorithm Verify executes the subsequent steps s times. We assume here that algorithm Verify is at the t^{th} iteration:
  (1) Given vector $\boldsymbol{\nu} = (\nu_1, ..., \nu_l)$, algorithm Verify checks whether

$$\tilde{\sigma}^{(t)} = \alpha \tilde{d}^{(t)} + \sum_{j=1}^{l} \nu_j \text{prf}(\beta, \text{fid} \parallel i_{d_t} + (j-1))$$

  If this is not the case, then algorithm Verify returns reject.
  (2) Otherwise, algorithm Verify generates $I_{d_{t+1}} = \text{prg}(\tilde{\sigma}^{(t)}, \tilde{d}^{(t)})$, sets the index $i_{d_{t+1}}$ for the next iteration to the first $\log(kL-l)$ bits of $I_{d_{t+1}}$ and goes back to step 1.

  If algorithm Verify does not return reject, it proceeds with verifying the integrity of the redundancy.
– *Redundancy possession verification.* Algorithm Verify first uses the pseudo-random generator prg, seed η and file identifier fid to get the shuffling function g. Then given vectors $\tilde{\mathbf{r}} = (\tilde{r}^{(1)}, ..., \tilde{r}^{(s)})$ and $\tilde{\boldsymbol{\psi}} = (\tilde{\psi}^{(1)}, ..., \tilde{\psi}^{(s)})$ it performs the following operations s times.
  (1) Given index r_t, algorithm Verify finds the shuffling function preimage $(x_{j_t}, y_{j_t}) = g^{-1}(i_{r_t} + j - 1)$ for all $1 \leq j \leq l$.
  (2) Given matrix $\mathbf{M} = [\mathbf{M}_1 \mid ... \mid \mathbf{M}_{(n-k)}]$, algorithm Verify checks whether the following equality holds:

$$\tilde{\psi}^{(t)} = \alpha \tilde{r}^{(t)} + \sum_{j=1}^{l} \nu_j \mathbf{M}_{y_{j_t}} \cdot \mathbf{prf}^{(x_{j_t})}$$

  whereby for all $1 \leq j \leq l$:

$$\mathbf{prf}^{(x_{j_t})} = (\text{prf}(\beta, \text{fid} \parallel (x_{j_t} - 1)k + 1), ..., \text{prf}(\beta, \text{fid} \parallel x_{j_t} k))$$

  If the equality is not satisfied, then algorithm Verify returns reject.
  (3) Otherwise, it computes $I_{r_{t+1}} = \text{prg}(\tilde{\psi}^{(t)}, \tilde{r}^{(t)})$, defines the new index $i_{r_{t+1}}$ for the next iteration by truncating the first $\log((n-k)L-l)$ bits of $I_{f_{t+1}}$ and goes back to step 1.

  If algorithm Verify does not return reject, then it concludes its execution by outputting accept.

B ERASURE CODES

Reliable storage systems store the original data with some degree of redundancy in order to tolerate component failures. The simplest way to introduce redundancy is replication where verbatim copies of a data object are stored in multiple locations in the storage system. Replication is conceptually simple and straightforward to implement, yet it inflicts high storage overhead.

Erasure codes are a generalization of replication that can ensure equivalent levels of failure tolerance with less storage overhead. A

class of erasure codes, called linear *maximum distance separable* (MDS) codes [20, 24] are desirable in storage systems' applications because they deliver the greatest error-detecting and correcting capabilities for the amount of storage space dedicated to redundancy information. In particular, a linear (k, n)-MDS code encodes a data object $\mathbf{D}$ of k data blocks into n encoded blocks, called codewords. Input data blocks and the corresponding codewords belong to the finite field $\mathbb{F}_p$, where p is a large prime and $k \leq n \leq p$. Once some storage failure is detected, the original data $\mathbf{D}$ can be recovered from any set of k encoded blocks and any corrupted codewords can be reconstructed. A linear (k, n)-MDS code can be computed by employing $k \times n$ generator matrix $\mathbf{G}$ with the property that any k columns are linearly independent. 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 [7].

Regarding the bandwidth required to repair a corrupted codeword, the performance of traditional erasure codes is suboptimal since the data object $\mathbf{D}$ has to be reconstructed in its entirety. In order to facilitate efficient repair of codewords, the literature features a number of erasure code classes from regenerating codes [23] to non-MDS codes like locally repairable codes (LRC) that are not optimal in terms of code rate, yet they allow for regeneration of lost codewords with less computation and/or communication overhead.

C THE PRIVATE POR SCHEME OF SHACHAM AND WATERS

In what follows, we briefly describe the private scheme in [19], which is the base for the PDP signature scheme used in POROS. This scheme takes advantage of a pseudo-random function prf. The data owner chooses her secret key that consists of a random number $\alpha \in \mathbb{Z}_p$ and a key β for the function prf and then she computes a homomorphic authentication tag for each block d_i of the original data object $\mathbf{D}$ as follows:

$$\sigma_i = \alpha d_i + \mathsf{prf}(\beta, i) \in \mathbb{Z}_p$$

The data object $\mathbf{D}$ together with the homomorphic MACs $\{\sigma_i\}$ are then outsourced to the CSP. The audit of the outsourced data is performed as follows. The data owner sends to the CSP a challenge chal that comprises a random l-element set $I \subset [1, n]$ and l random coefficients v_i in $\mathbb{Z}_p$. The CSP then computes the proof $\pi = (\sigma, d)$ as:

$$d = \sum_{(i, v_i) \in \mathsf{chal}} v_i d_i \ , \quad \sigma = \sum_{(i, v_i) \in \mathsf{chal}} v_i \sigma_i$$

π is transmitted to the data owner who can verify the integrity of the data by checking that

$$\sigma \stackrel{?}{=} \alpha d + \sum_{(i, v_i) \in \mathsf{chal}} v_i \cdot \mathsf{prf}(\beta, i)$$

Thanks to the unforgeability of homomorphic MACs, a malicious CSP cannot corrupt outsourced data without being detected.