Reconciling cloud storage functionalities with security : proofs of storage with data reliability and secure deduplication

Dimitrios Vasilopoulos

Sorbonne University

Doctoral School of
Informatics, Telecommunications and Electronics (Paris)
EURECOM

Reconciling Cloud Storage Functionalities with Security:
Proofs of Storage with Data Reliability and Secure Deduplication

Presented by Dimitrios VASILOPOULOS

Dissertation for Doctor of Philosophy in
Information and Communication Engineering

Directed by Melek ÖNEN

Publicly presented and defended on 23 July 2019

before a committee composed of:

Prof. Bruno MARTIN	University of Nice Sophia-Antipolis	President of the jury
Prof. Pascal LAFOURCADE	University of Clermont Auvergne	Reporters
Prof. Yves ROUDIER	University of Nice Sophia-Antipolis	
Prof. Loukas LAZOS	University of Arizona	Examiners
Prof. Aurélien FRANCILLON	EURECOM	
Prof. Melek ÖNEN	EURECOM	Thesis Supervisor

Sorbonne Université

École Doctorale
Informatique, Télécommunications et Électronique de Paris
EURECOM

Réconcilier les fonctionnalités de stockage cloud avec les besoins de sécurité:

Preuves de stockage avec la fiabilité des données et la déduplication sécurisée

Par Dimitrios VASILOPOULOS

Thèse de doctorat en
Sciences de l'Information et de la Communication

Dirigée par Melek ÖNEN

Présentée et soutenue publiquement 23 juillet 2019

devant un jury composé de:

Prof. Bruno MARTIN	Université de Nice Sophia-Antipolis	Président du jury
Prof. Pascal LAFOURCADE	Université de Clermont Auvergne	Rapporteurs
Prof. Yves ROUDIER	Université de Nice Sophia-Antipolis	
Prof. Loukas LAZOS	Université de l'Arizona	Examinateurs
Prof. Aurélien FRANCILLON	EURECOM	
Prof. Melek ÖNEN	EURECOM	Directeur de thèse

Abstract

Cloud storage is one of the most attractive services offered by cloud computing. Indeed, cloud storage systems offer users the opportunity to outsource their data without the need to invest in expensive and complex storage infrastructure. However, in this new paradigm users lose control over their data and lend it to cloud storage providers. Hence, users lose the ability to safeguard their data using traditional IT and security mechanisms. As a result, technical solutions aiming at establishing users confidence on the services provided by a cloud storage system would be highly beneficial both to users and cloud storage providers.

In this manuscript we study in depth the problem of verifiability in cloud storage systems. We study Proofs of Storage – a family of cryptographic protocols that enable a cloud storage provider to prove to a user that the integrity of her data has not been compromised – and we identify what their limitations with respect to two key characteristics of cloud storage systems, namely, *reliable data storage with automatic maintenance*, and *data deduplication*.

To cope with the first limitation, we introduce the notion of *Proofs of Data Reliability*, a comprehensive verification scheme that aims to resolve the conflict between reliable data storage verification and automatic maintenance. We further propose two Proofs of Data Reliability schemes, namely POROS and PORTOS, that succeed in verifying reliable data storage mechanism and a the same time enable the cloud storage provider to autonomously perform automatic maintenance operations.

As regards to the second limitation, we address the conflict between Proofs of Storage and deduplication. More precisely, inspired by previous attempts in solving the problem of duplicating encrypted data, we propose *message-locked PoR* a solution in combining Proofs of Storage with deduplication. In addition we propose a novel *message-locked* key generation protocol which is more resilient against off-line dictionary attacks compared to existing solutions.

Acknowledgements

The work presented in this thesis would have not been accomplished without the support, encouragement and assistance of a great number of individuals.

First and foremost, I would like to express my deep and sincere gratitude to both my academic supervisors Professor Refik Molva and Professor Melek Önen for providing an excellent guidance and a constant source of inspiration and motivation. Refik opened the door to my research path and taught me how to think out-of-the-box with discipline, creativity and meaningful criticism. Melek worked at my side on all my publications and gave a great contribution to the ideas I developed during my Ph.D. Moreover, she showed an extreme amount of patience on my stubbornness in various aspects during our collaboration. It was a rare privilege and honor to work and study under her guidance.

I am also indebted to Dr. Kaoutar Elkhyaoui whose help and comments have been crucial to the establishment of many of the results presented in this thesis. Furthermore, I want to thank the people of my group: Iraklis, Monir, Cedric, Beyza, Orhan and Clementine that were always willing to discuss about research problems and offer me valuable advice.

I would like to thank my jury committee of Pascal Lafourcade, Yves Roudier, Loukas Lazos, Bruno Martin and Aurélien Francillon who kindly agreed to review and evaluate my work.

Pursuing Ph.D. studies is truly a challenging endeavour whose success also depends on the encouragement of family and friends. I am therefore grateful to all the people who gave me their emotional support. In particular, special thanks goes to Panos, Akis, Pantelis, Thodoris, Lefteris, Thanasis and Konstantinos for their support during the compilation of this thesis.

Last but not least, I would like to thank as well as dedicate this thesis to my fiancée Ntora. Her constant support, love and boundless encouragement, significantly helped me towards completing this work. I am also appreciative to my parents who sacrificed a lot of their personal time and financial budget (unfortunately) for my education.

Contents

List of Figures

List of Tables

Papers Published during Ph.D

Dimitrios Vasilopoulos, Melek Önen and Refik Molva. 2019. PORTOS: Proof of Data Reliability for Real-World Distributed Outsourced Storage. Accepted in 16th International Conference on Security and Cryptography (SECRYPT '19). Prague, Czech Republic, July 26–28, 2019.

Dimitrios Vasilopoulos, Kaoutar Elkhiyaoui, Refik Molva, and Melek Önen. 2018. POROS: Proof of Data Reliability for Outsourced Storage. In Proceedings of the 6th International Workshop on Security in Cloud Computing (ASIACCS–SCC '18). Songdo, Incheon, Korea, June 4–8, 2018.

Dimitrios Vasilopoulos, Melek Önen, Kaoutar Elkhiyaoui, and Refik Molva. 2016. Message-Locked Proofs of Retrievability with Secure Deduplication. In Proceedings of the 2016 ACM on Cloud Computing Security Workshop (CCS–CCSW '16). Vienna, Austria, October 24–28, 2016.

Chapter 1

Introduction

In recent years, advances in the areas of networking, hardware virtualization and distributed algorithms enabled the rise of a novel computing paradigm, namely, *cloud computing*. What makes this new paradigm appealing is its *pay-as-you-go* economic model, that offers users and organizations access to a certain amount of computing power, bandwidth, and data storage capabilities without the need to invest in on-premise infrastructure, hence minimizing the startup and setup cost and time. Put differently, in cloud computing the supply of computing infrastructure becomes the service. The success of cloud computing can be attributed to several key characteristics such as:

Flexibility that enables the rapid provision of resources.

Multi-tenancy that allows multiple users to share the same instance of a service.

Elasticity that facilitates the dynamic mitigation of the variability in demand for a service by scaling up and down allocated resources.

Scalability that is the ability of a service to remain stable as demand increases by adding auxiliary resources to the service.

Yet, this massive outsourcing of data and computation to a third party, namely, the cloud provider, raises significant issues of trust and security: because users lose control over their data, they have to rely on the cloud provider for the implementation of the appropriate security measures. Indeed, public cloud providers constitute prime targets for attackers due to the amount of valuable data they concentrate. Furthermore, in the cloud setting, the cloud provider itself can be considered as the adversary: a potentially malicious cloud provider can hide data losses to preserve its reputation, delete some rarely accessed data to save storage space, or access the data it is storing without the consent of

the data owners. As a result, these privacy and security concerns appear to be the main obstacle to the widespread adoption of cloud computing.

Traditional security mechanisms such as encryption to ensure data confidentiality and cryptographic-based techniques to guarantee data integrity neutralize the advantages of cloud computing. This has led to the research of new techniques to increase security in this paradigm. Among these techniques, there are a number of cryptographic protocols that aim to provide guarantees to users with respect to the behavior of an untrusted cloud provider. Some examples include security primitives for: *privacy preserving data processing* such as *searchable encryption* that make it possible to search a database stored on a cloud provider, while limiting the amount of information that the cloud provider can obtain regarding the stored data and the queries that are made. *Verifiable data processing* that enable a user to verify that a computation carried out by the cloud provider is executed correctly without the need to repeat them locally. And *verifiable storage* such as *proofs of storage* that provide users with cryptographic guarantees that their data are stored correctly by a cloud storage provider. Ideally, these solutions should be designed in a manner that outsourcing to the cloud remains an attractive option for both users and cloud provider: the deployment of security primitives should not not hinder the principal functionalities of cloud provides and as a result cancel out the advantages of cloud computing.

Focus of this work. This thesis addresses the problem of *reconciling cloud storage functionalities with security primitives*. The goal is to devise cryptographic mechanisms used that enable users to *verify* that a cloud storage system correctly delivers the promised services of data storage. Indeed, by outsourcing their data, users lose control over their data and lend it to cloud storage providers. Hence, users lose the ability to safeguard their data using traditional IT and security mechanisms. As a result, technical solutions aiming at establishing users confidence on the services provided by a cloud storage system would be highly beneficial both to users and cloud storage providers.

Furthermore, the proposed mechanism should be compatible with the functionalities and responsibilities of the different parties as they are defined by the current cloud system model. In other words, these mechanisms should not transfer responsibilities from the cloud storage system to users, or require cloud-user communication in order to carry out functionalities that are typically performed by the cloud storage system.

1.1 Requirements of Cloud Storage Systems

Cloud storage is definitively one of the most attractive services offered by cloud computing. With the proliferation of "big data" as well as with the extensive adoption of IoT devices that increase exponentially the amount of new data generated every day, users need a way to safely and reliably store them. Cloud storage systems bring what is needed to satisfy the above-mentioned needs. In particular, cloud storage systems provide users with the option to outsource the storage of their data without the need to invest in expensive and complex storage infrastructure.

A cloud storage system should encompass basic functionalities that enable it to meet the following *functional requirements* [1]:

Reliable data storage. Storage systems should be built in such a manner that when they suffer form hardware failures, software bugs, or outages, users' data can be recovered. This property is achieved by adding redundancy to the storage system to tolerate failures.

Redundancy mechanisms. There are two mechanisms that add redundancy to a storage system:

- The first one is *replication*, that is the process of storing multiple copies of the entire data on different storage nodes. If one of the copies is deleted or corrupted due to some failure, the storage system uses other copies to recover the data and produce a new copy.
- The second one involves the use of *erasure codes* to encode the data before writing it across multiple storage devices. In the face of failure in one or more storage devices, the storage system uses the erasure code and the portion of the data on the remaining healthy storage devices to reconstruct the data.

In the cloud setting, the cloud provider is responsible to integrate these mechanisms into its infrastructure. Generally, cloud storage systems [2, 3] use replication due to its simple implementation, however with the explosion of digital content cloud storage provides [4–6] are starting to turn to erasure coding in order to reduce the volume of stored data: indeed, erasure codes provide the advantage of less storage overhead for the same amount of redundant information at the cost of the initial encoding processing.

Data repair. Complementary to the added redundancy, reliable data storage also

rely on mechanisms that allow for the detection of data corruption such as cyclic redundancy checks (CRC) or checksums. The storage system 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.

Data reduction. Data reduction techniques such as *data deduplication* and *data compression* afford cloud storage providers a better utilization of their storage capacity and the ability to serve more customers with the same infrastructure.

Data deduplication. Data deduplication is the process whereby a storage system inspects the data it stores, identifies large sections of repeated data, such as entire files or parts of files, and replaces them with a single shared copy. Data deduplication methods can be classified according their granularity in two types:

- *File-level deduplication*, where the storage system detects multiple copies of the same file uploaded by different users and ensures that only a unique copy is stored; and
- *Block-level deduplication*, where the storage system first splits files in smaller segments called data blocks and thereafter identifies common data blocks among all uploaded files keeping a single copy of each data block.

Another way to classify data deduplication methods is according to where deduplication occurs:

- In *server-side deduplication*, users upload their files in their entirety and the storage system analyses them and performs the deduplication operation transparently to the users.
- In *client-side deduplication*, users first compute a unique identifier for the files or each of the blocks of the files they wish to upload and sends them to the storage system. The latter uses these identifiers to determine whether a particular file or block has previously been uploaded in the past and request the users to upload only files or blocks that it does not already store.

Data compression. Data compression is the process whereby a storage system identifies redundant data inside individual files and encodes this redundant data more efficiently storing in this way a smaller version of the file.

Automatic maintenance. Automatic maintenance is a core property of cloud storage systems. It entails that all data management operations such as storage allocation, data reduction, redundancy generation, and data repair are performed by the cloud storage system in transparent manner to the user.

Multi-tenancy. A cloud storage system should accommodate a multi-tenant environment: namely, an environment in which multiple users share the ownership of outsourced data, or are permitted to operate on data owned by other users. Moreover, the owner of outsourced data should be given the possibility to control who accesses her data, how, and for what amount of time.

Dynamic Data. A cloud storage system should support dynamic data, i.e., data that is prone to updates. Such updates include appending new data, modifying segments of existing data, or deleting parts of the outsourced data.

Additionally, a cloud storage system should also fulfill three key *security requirements*, namely, data confidentiality, data integrity, and data availability – known as "The CIA triad" [7]. These requirements deal with the set of security functionalities that a cloud storage system should implement to assure users of the correct storage and the privacy of their data.

Data confidentiality. Confidentiality is a fundamental concept security that postulated that data data is not made available or disclosed to unauthorized parties. The two main mechanisms to ensure confidentiality are encryption and access control.

Data integrity. Integrity is a fundamental concept of security that postulates that data maintain its consistency, accuracy, and trustworthiness over its entire life cycle. Put differently, integrity ensures that data is not deleted or in any other way modified by unauthorized parties.

Data availability. Availability is a fundamental security property that ensures that data is accessible to all authorized parties at all times. Put differently, availability assures a user that she can download her data when needed.

1.2 Verifiability in Cloud Storage Systems

This thesis focuses on the aspect of verifiability in the context of cloud storage systems. Verifying the behavior of a cloud storage system provides transparency regarding the controls and functionalities it implements in order to safely handle users' data. In particular,

our goal is to design cryptographic protocols that (i) enable users to verify the *integrity*, *availability* and *reliable storage* of their outsourced data, and (ii) do not impede the mechanisms that realize the *functional requirements* presented in Section 1.1.

As regards to the *verification of the integrity* of outsourced data, literature features a large body of work called *Proofs of Storage* (PoS) [8–10]. The proposed solutions mainly consist of cryptographic protocols that enable a cloud storage provider to prove to a user that the integrity of her data has not been compromised. The user can ask the cloud storage provider to provide such proofs for an outsourced data object without the need to download the entire object in order to verify that the latter is stored correctly – only a small fraction of the data object has to be accessed in order to generate and verify the proof. Nevertheless, Proof of Storage schemes provide guarantees regarding *only* the integrity of outsourced data and do not take into account the rest of the functional and security requirements a cloud storage system oughts to fulfill.

In what follows we describe the challenges that arise when we try to extend Proofs of Storage such that they enable (i) the verification of *reliable data storage* by the user, and (ii) the deduplication of outsourced data by the cloud storage system.

Verification of reliable data storage with automatic maintenance. Reliable data storage can be seen as the underpinning mechanism that realizes *data integrity* and *data availability*. It relies on *redundancy* and *data repair* mechanisms to detect and restore corrupted data. Since data repair mechanisms are part of reliable data storage, the assurance of both data integrity and data availability by a PoS scheme must verify not only the integrity of the data outsourced by the users but also the effectiveness of the data repair mechanisms. Hence, in addition to the integrity of the data outsourced to a cloud storage system, a PoS scheme 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.

However, redundancy is a function of the original data 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. Besides, the straightforward approach where the user locally generates the required redundancy and further encrypts both the data and the redundancy in order to hide the relationship between the two, 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.

Conflict between PoS schemes and data deduplication. Proof of Storage schemes implement an encoding algorithm that incorporates some integrity values within the data before outsourcing it to a cloud storage provider. These values are further used by the user to verify the proofs provided by the cloud storage provider. Unfortunately, current PoS solutions are incompatible with data deduplication because the integrity values resulting from the encoding algorithm are generated using a secret key that is only known to the owner of the file, and thus unique. Therefore, the encoding of a given file by two different users results in two different outputs which cannot be deduplicated.

1.3 Contributions

In this thesis we answer the above challenges and we propose the following contributions.

Verification of reliable data storage with automatic maintenance. We introduce the notion of *Proofs of Data Reliability*, a comprehensive verification scheme that aims to resolve the conflict between reliable data storage verification and automatic maintenance. In particular, we provide the definition of a cryptographic proof of storage protocol and a new security model against a *rational* adversary. We propose two Proofs of Data Reliability schemes that succeed in verifying reliable data storage mechanism and a the same time enable the cloud storage provider to autonomously perform automatic maintenance operations.

Conflict between PoS schemes and data deduplication. We address the conflict between Proofs of Storage and deduplication. More precisely, inspired by previous attempts in solving the problem of duplicating encrypted data, we propose a straightforward solution in combining PoS and deduplication. In addition we propose a novel message-locked key generation protocol which is more resilient against off-line dictionary attacks compared to existing solutions.

1.4 Organization

The remaining of this thesis is organized as follows:

- In Chapter 2, we present the concept of Proofs of Storage and a particular flavor of Proofs of Storage schemes named Proofs of Retrievability. We further summarize the state of the art on Proofs of Storage and we identify what we believe to be the limitations of current Proof of Storage protocols with respect to two key characteristics

of cloud storage systems, namely, reliable data storage with automatic maintenance, and data deduplication.

The reader then can either move on to Part I of this thesis which introduces a comprehensive verification scheme that aims to resolve the conflict between reliable data storage verification and automatic maintenance, or Part II which addresses the conflict between Proof of Retrievability schemes and deduplication.

- In Chapter 3, 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 and we summarize the state of the art in this field.

- In Chapter 4, 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 *rationale* 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.

- In Chapter 5, 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.

- In Chapter 6 we address the conflict between PoR and deduplication. More precisely, inspired by previous attempts in solving the problem of duplicating encrypted data, we propose a straightforward solution in combining PoR and deduplication. In addition we propose a novel message-locked key generation protocol which is more resilient against off-line dictionary attacks compared to existing solutions.

- Finally in Chapter 7, we conclude with the results of this dissertation and we discuss future research avenues.

Chapter 2

Verifiable Storage

In this chapter, we first introduce the concept of Proofs of Storage (PoS) and a particular flavor of Proofs of Storage schemes named Proofs of Retrievability (PoR). We then review the state of the art on Proofs of Storage and present two Proofs of Retrievability schemes that are mentioned throughout this manuscript and are the basis of the work conducted during this thesis. Finally, we discuss the limitations of Proof of Storage protocols with respect to two key characteristics of cloud storage systems, namely, data reliability and storage efficiency, which becomes the main research topic of this Ph.D study.

2.1 Proofs of Storage

With the almost unlimited storage capacity offered by cloud storage providers, users tend to outsource their data and thereby offload to the cloud the burden that comes with locally storing these data: namely, the acquisition and maintenance cost of storage infrastructure. Nonetheless, through this process users lose their physical control over their data. As a result, potentially *untrusted* cloud storage providers assume the full responsibility of storing these data. Yet, cloud storage providers currently do not assume any liability for data loss [11]. Data loss occurs whenever there is deletion or unrecoverable modification of the outsourced data from any unauthorized party. More specifically, data losses may be the outcome of

(i) actions of malicious attackers that delete or tamper with the data; or

(ii) accidental data corruption due to failures of the hardware of the underlying storage infrastructure or bugs in the software stack.

Hence, users might be reluctant to adopt cloud storage services because they lack means to verify the *integrity* of their outsourced data.

In the context of cloud storage the traditional integrity checking technique that relies on the use of digital signatures does not scale because it incurs high communication cost: every time a user wants to verify the integrity of her potentially large data, she has to download it and verify it with the locally stored signature. Proofs of Storage (PoS) are cryptographic solutions that enable users to verify the correct storage of their outsourced data without canceling out the advantages of the cloud. In particular, a user can ask the cloud storage provider to furnish such proofs for her outsourced data without the need to download the data. The notion of PoS was first formalized by Ateniese et al. [8] and Juels and Kaliski [9], then updated by Shacham and Waters [10].

2.1.1 Environment

A Proof of Storage (PoS) 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. 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. Depending on the specification of the PoS scheme, user U can be the verifier V.

We consider a setting where a user U uploads a file D to a cloud storage provider C and thereafter a verifier V periodically queries C for some proofs on the correct storage of D. In practice, user U processes file D and derives a *enlarged verifiable* data object $\mathcal{D}$ which subsequently uploads to cloud storage provider C. $\mathcal{D}$ contains some additional information that will further help for the verification of its integrity. At this point, the cloud storage provider C is expected to store data object $\mathcal{D}$, while the user U deletes both file D and data object $\mathcal{D}$ retaining in local storage the key material she used during the generation of $\mathcal{D}$. We define a Proof of Storage scheme as a protocol executed between a user U and a verifier V on the one hand, and the cloud storage provider C on the other hand. In order to check the integrity of data object $\mathcal{D}$, the verifier V issues one or more PoS challenges and sends them to the cloud storage provider C. The PoS challenge may refer to either the whole data object $\mathcal{D}$ or a subset of $\mathcal{D}$'s symbols. In turn, C generates a PoS proof of each query it receives and sends it to the verifier V. Finally, V checks whether the PoS 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 the verifier V downloading the entire content associated to $\mathcal{D}$.

2.1.2 Definition of a PoS scheme

We now give the formal definition and security requirements of a Proof of Storage scheme. The definition we propose here follows the challenge-response approach proposed in [9] while defining a *stateless* protocol as suggested by Shacham and Waters [10]. The term stateless implies that the verifier V does not maintain any additional information (i.e., a state) between subsequent invocations of the protocol.

Definition 1. (PoS Scheme). *A Proof of Storage scheme is defined by five polynomial time algorithms:*

- $\mathsf{KeyGen}\ (1^\lambda) \rightarrow (K_\mathsf{u}, K_\mathsf{v})$*: This randomized key generation algorithm is executed by* U *as a first step. It takes as input a security parameter* λ*, and outputs the user key* K_u *for user* U *and the verifier key* K_v *for verifier* V*.*

- $\mathsf{Encode}\ (K_\mathsf{u}, D) \rightarrow (\mathsf{fid}, \mathcal{D})$*: User* U *calls this algorithm before the actual outsourcing of her file. It takes as inputs the user key* K_u *and the file* D *composed of* n *segments, namely,* $D = \{D_1, D_2, \ldots, D_n\}$ *and returns a unique identifier* fid *for* D *and an encoded version* $\mathcal{D} = \{\hat{D}_1, \hat{D}_2, \ldots, \hat{D}_n\}$ *which includes some additional information that will further help for the verification of its integrity.*

  Algorithm Encode *is invertible: namely, there exists an interactive algorithm* $\mathsf{Decode}\ (K_\mathsf{u}, \mathsf{fid}, \mathcal{D}) \rightarrow (D)$ *that, when called successfully by the user* U*, recovers and outputs the original file* D*.*

- $\mathsf{Chall}\ (K_\mathsf{v}, \mathsf{fid}) \rightarrow (\mathsf{chal})$*: Once the encoded file* $\mathcal{D}$ *with identifier* fid *is outsourced to* C*, verifier* V *invokes this probabilistic algorithm with verifier key* K_v *and file identifier* fid *to compute a PoS challenge* chal *and sends this challenge to cloud storage provider* C*.*

- $\mathsf{Prove}\ (\mathsf{fid}, \mathsf{chal}) \rightarrow (\mathsf{proof})$*: This algorithm is executed by the cloud storage provider* C *and returns a proof of storage* proof *given file identifier* fid *and the PoS challenge* chal*.*

- $\mathsf{Verify}\ (K_\mathsf{v}, \mathsf{fid}, \mathsf{chal}, \mathsf{proof}) \rightarrow (\mathsf{dec} \in \{\mathsf{accept}, \mathsf{reject}\})$*: Upon reception of* proof*, verifier* V *calls this algorithm which takes as input the verifier key* K_v*, the file identifier* fid*, the challenge* chal *and the proof* proof *and outputs* $\mathsf{dec} = \mathsf{accept}$ *if* proof *is a valid proof and* $\mathsf{dec} = \mathsf{reject}$ *otherwise.*

A Proof of Storage scheme should be *correct* and *sound*.

2.1.2.1 Correctness

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

2.1.2.2 Soundness

A Proof of Storage scheme is *sound* if any cheating cloud storage provider C that convinces the verifier V that it is storing the verifiable data object $\mathcal{D}$ is actually storing $\mathcal{D}$. In other words, a cheating cloud storage provider C cannot output valid PoS proof for a file D without storing the verifiable data object $\mathcal{D}$ in its entirety.

The *soundness* requirement as defined above, enables a verifier V to check the integrity of data object $\mathcal{D}$. However, due the probabilistic nature of many Proof of Storage types, namely, Proofs of Data Possession (PDP) [8] (c.f. Section 2.3), a successful PoS verification does not guarantee that the outsourced file D can be recovered in its entirety by the user U. More specifically, each PoS challenge verifies the integrity of a rather small random subset of $\mathcal{D}$'s symbols. Hence, there is the possibility that the protocol will fail to detect the presence of some minor corruption of $\mathcal{D}$.

2.1.2.3 Additional features of PoS scheme

In addition to fulfilling the security requirements of *correctness* and *soundness*, a Proof of Storage Protocol may also have the following features.

Efficiency. In order not to cancel out the storage and performance advantages of the cloud, the PoS verification should be performed without the verifier V downloading the whole data object $\mathcal{D}$ from the cloud storage provider C. The performance of a PoS scheme is evaluated in terms of five types of metrics: (i) the *storage overhead* of data object $\mathcal{D}$ induced by algorithm Encode, (ii) the *bandwidth* required to transmit the challenge chal and proof proof, (iii) the *computational cost* of algorithm Encode, (iv) the *computational cost* of algorithm Prove, and (v) the *computational cost* of algorithm Verify.

Stateless verification. A Proof of Storage protocol should be stateless in the sense that the verification of a PoS proof generated by the cloud storage provider C should not require that the verifier V keeps any historical information regarding prior executions of the protocol.

Unbounded number of PoS executions. A Proof of storage protocol should allow the verifier V to enter in an unbounded number of PoS executions. In other words, the user U should not have to retrieve end re-encode file D after a finite number of PoS executions.

Public verifiability. We consider two settings for a proof of storage scheme: a privately verifiable setting and a publicly verifiable one. In a privately verifiable PoS scheme the verifier V is the user U. In a publicly verifiable PoS scheme the role of the verifier V can be played by any third party except C.

2.2 Proofs of Retrievability

Proofs of Retrievability (PoR) are a family of Proofs of Storage that have a stronger *soundness* definition which ensures user U that not only her outsourced data are correctly stored by a malicious cloud storage provider C but also verifier V can recover her data at any time. This security guarantee is captured using the notion of *extractability*: this requirement suggests that there exists an extractor algorithm Extract that can interact with cloud storage provider C and recover user U's data. Proofs of Retrievability are probabilistic solutions, hence a successful PoR verification provides user U with the assurance that she can retrieve her data with some overwhelming probability.

As a type of Proofs of Storage, PoR schemes operate in the same environment and inherit the same algorithm and characteristics of a PoS protocol. To fulfill the extractability requirement, the PoR Encode algorithm makes use of erasure codes, a pseudo-random permutation and a semantically secure encryption. In the following, we introduce these three primitives.

Erasure codes. Erasure codes [12] are a family of error-correction codes that assume bit erasures instead of bit errors. An $[n, k, d]$–erasure code denotes a code which transforms a word which consists of k symbols into a codeword comprising n symbols. d denotes the minimum Hamming distance between codewords: the code can detect and repair up to $d - 1$ errors in a code word. Input data symbols and the corresponding codeword symbols belong to $\mathbb{Z}_p$, where p is a large prime and $k \leq n \leq p$. Once some failure (erasure) is detected, the remaining codeword symbols are used to reconstruct that lost symbols. The reconstructed codeword 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. Any erasure code is suitable for a Proof of Retrievability protocol.

Pseudo-random permutation (PRP). A pseudo-random permutation [13] is a function $\mathsf{PRP} : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{X}$, 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 $\mathsf{PRP}^{-1} : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{X}$ 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 $\pi : \mathcal{X} \rightarrow \mathcal{X}$.

Semantically secure encryption. A semantically secure encryption scheme [14] is a probabilistic encryption scheme that guarantees that given the encryption C of a message M chosen from a set of messages $\mathcal{M}$ and the size of M, a Probabilistic Polynomial-Time (PPT) adversary can neither distinguish which of the messages in $\mathcal{M}$ corresponds to the ciphertext C nor determine any partial information on the original message M.

2.2.1 Overview of the PoR Encode algorithm

Before creating the verifiable data object $\mathcal{D}$, the PoR Encode algorithm first encodes the file D by applying the erasure code. The additional redundancy symbols assure the protection of file D against small corruptions. However, erasure coding alone does not guarantee that the file D can be successfully recovered when the verifier V interacts with a malicious cloud storage provider C. Indeed, erasure codes provide resiliency only against random symbol erasures. For this reason, algorithm Encode permutes and encrypts the encoded file in order to hide the dependencies among the symbols of D as well as eliminating any segment structure arising from the application of the erasure code. These two operations effectively reduce C's adversarial symbol erasures to random ones. Thereafter, the PoR Encode algorithm proceeds with the creation of the verifiable data object $\mathcal{D}$ in the same way as the corresponding PoS Encode algorithm.

2.2.2 Security Requirements of PoR

In this section we formalize our definition for the two security requirements a Proof of Retrievability scheme must fulfill, namely *correctness* and *soundness*.

2.2.2.1 Correctness

As mentioned in the previous section, the *correctness* security requirement postulates the fact that an honest cloud storage provider C, who stores the whole data object $\mathcal{D}$, should always be able to pass the verification of proof of storage protocol.

Definition 2. (PoR Correctness). *A PoR scheme* (KeyGen, Encode, Chall, Prove, Verify) *is* ***correct*** *if for all honest cloud storage providers* C *and verifiers* V*, and for all keys* $\mathcal{K} \leftarrow \mathsf{KeyGen}\ (1^{\lambda})$*, all files* D *with file identifiers* fid *and for all challenges* $\mathsf{chal} \leftarrow \mathsf{Chall}\ (\mathcal{K}, \mathsf{fid})$*:*

$$\mathsf{Pr}[\mathsf{Verify}(\mathcal{K}, \mathsf{fid}, \mathsf{chal}, \mathsf{proof}) \rightarrow \mathsf{accept} \mid \mathsf{proof} \leftarrow \mathsf{Prove}(\mathsf{fid}, \mathsf{chal})] = 1$$

2.2.2.2 Soundness

It is essential for any Proof of Retrievability protocol that the verifier V can always detect (except with negligible probability) that the cloud storage provider C deviates from the correct protocol execution. We capture this requirement using the notion of *extractability* as introduced in [9, 10]. Extractability guarantees that if a cloud storage provider C can convince an honest verifier V that it stores data object $\mathcal{D}$, then there exists an extractor algorithm $\mathsf{Extract}$ that can interact with C and recover, with high probability, the file D.

To define this requirement, we consider a hypothetical game between an adversary $\mathcal{A}$ and an environment where the latter 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.

We consider the following retrievability game between the adversary and the environment:

1. $\mathcal{A}$ interact with the environment and queries $\mathcal{O}_{\mathsf{Encode}}$ providing, for each query, some file D. The oracle returns to $\mathcal{A}$ the tuple $(\mathsf{fid}, \mathcal{D}) \leftarrow \mathcal{O}_{\mathsf{Encode}}\ (\mathcal{K}, D)$, for a given user key K_{v}.

2. For any file identifier fid of a file D it has previously sent to $\mathcal{O}_{\mathsf{Encode}}$, $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Chall}}$ to generate a random challenge $(\mathsf{chal}) \leftarrow \mathcal{O}_{\mathsf{Chall}}\ (\mathcal{K}, \mathsf{fid})$, for a given verifier key K_{v}.

3. Upon receiving challenge chal, $\mathcal{A}$ generates a proof proof either by invoking algorithm Prove or at random.

4. $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Verify}}$ to check the proof proof and gets the decision $(\mathsf{dec}) \leftarrow \mathcal{O}_{\mathsf{Verify}}\ (K_{\mathsf{v}}, \mathsf{fid}, \mathsf{chal}, \mathsf{proof})$, for a given user key $\mathcal{K}$.

   Steps $1-4$ can be arbitrarily interleaved for a polynomial number of times.

5. Finally, $\mathcal{A}$ picks a file identifier fid^* corresponding to a file D^* that was sent to $\mathcal{O}_{\mathsf{Encode}}$ and outputs a simulation of a cheating cloud storage provide C'.

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

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

for a given verifier key K_{v}.

Hereafter, we define the extractor algorithm $\mathsf{Extract}$ which uses the simulation of a cheating cloud storage provide C' to retrieve file D^*:

- $\mathsf{Extract}\ (K_{\mathsf{v}}, \mathsf{fid}^*, \mathsf{C}') \rightarrow D^*$: On input of verifier key K_{v}, file identifier fid and the description of a cheating cloud storage provide C', the extractor algorithm $\mathsf{Extract}$ initiates a polynomial number of protocol executions with C' and outputs the file D^*. Algorithm $\mathsf{Extract}$ is allowed to rewind C'.

Definition 3. (PoR Soundness). *We say that a PoR scheme* (KeyGen, Encode, Chall, Prove, Verify) *is ϵ-sound if there exists an extraction algorithm* $\mathsf{Extract}$ *such that, for every Probabilistic Polynomial-Time adversary $\mathcal{A}$ who plays the retrievability 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.*

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

where ϵ is a negligible function in security parameter λ.

2.2.3 Classification of PoR schemes

Proof of Retrievability schemes can be classified into two categories which mainly differ with respect to their initial encoding phase where the user U invokes the algorithm Encode

in order to derive the to-be-outsourced data object $\mathcal{D}$ from the file D.

- In the first category of PoR schemes, algorithm Encode computes an authentication tag for each data symbol. The verifier V samples a subset of data object symbols together with their respective tags and verifies the integrity of the symbols. Most of the recent *tag-based* solutions [8, 10] employ *homomorphic tags* which during the verification phase, reduce the communication bandwidth: the verifier V queries the cloud storage provider C to receive a linear combination of a selected subset of data object symbols together with the same linear combination of the corresponding tags.

- In the second category of PoR schemes, named as *watchdog-based* PoRs, algorithm Encode embeds pseudo-randomly generated "watchdogs" [15] or "sentinels" [9] in data object $\mathcal{D}$ at random positions, and further encrypts the data to make the watchdogs indistinguishable from original data symbols. During the verification phase, the verifier V retrieves a subset of these watchdogs.

2.3 State of the Art on Proofs of Storage

In this section, we summarize Proof of Storage solutions that provide mechanisms to enable the integrity verification of outsourced data. These solutions include the pioneering work by Juels and Kaliski [9] on Proofs of Retrievability (POR) and Ateniese et al. [8] on Provable Data Possession (PDP). PoR and PDP aim at reducing the communication and computational complexity of the challenge-response protocol at the price of not offering an unconditional data integrity guarantee. The core idea of these proposals is to check the integrity of a subset of symbols of the data object $\mathcal{D}$ instead of checking the integrity of the entire data object $\mathcal{D}$, while still providing assurance for the integrity of the entire data object with high probability.

2.3.1 Watchdog-based solutions

Juels and Kaliski [9] proposed the Proofs of Retrievability (PoR) model that ensures that users at any point of time can retrieve outsourced data. This property is achieved by means of erasure codes, pseudo-random permutation, semantically secure encryption, and randomly generated blocks. The scheme in [9] embeds pseudo-randomly generated symbol, called sentinels, into encrypted stored data, hence data blocks and sentinels are indistinguishable. To check the retrievability of the data, the verifier selects a random subset of sentinels and queries the server for them. This proposal only supports a bounded number of POR queries after which the server may discover all the embedded sentinels.

Azraoui et al. [15] pursue this approach in [9] with StealthGuard. In this scheme the data owner inserts pseudo-random blocks, called watchdogs, before outsourcing the data to the cloud. To achieve unbounded number of verification queries StealthGuard uses a privacy-preserving word search scheme to verify the existence of watchdogs without disclosing their position to the remote server –hence, reusing them in the future.

2.3.2 Tag-based solutions

Ateniese et al. [8] define the Provable Data Possession (PDP) model, which ensures that a large portion of the outsourced data is stored in storage servers. The authors propose a PDP scheme that uses homomorphic verification tags as check values for each data block. Thanks to the homomorphic property of the tags, the server can combine tags of multiple data blocks into a single value, reducing thus the communication overhead of the verification. This proposal was later extended in [16] where the notion of robust auditing was defined. Their new scheme integrates error-correcting codes to mitigate arbitrarily small file corruptions. Following another direction, Ateniese et al. [17] proposed a symmetric PDP construction to support dynamic writes/updates on stored data. The idea is to precompute challenges and answers as metadata in advance, however, this limits the number of verifications the data owner can conduct. Erway et al. [18] proposed a dynamic PDP scheme that relies on authenticated dictionaries based on rank information to support dynamic data operations. Later on, Chen and Curtmola [19] extended the work in [16] in order to support data updates.

Most proposals for PoR employ homomorphic tags like in [8], in conjunction with erasure codes. Shacham and Waters [10] propose symmetric as well as publicly verifiable PoR schemes that use homomorphic tags to yield compact proofs. This work has been extended by Xu and Chang in [20] where a polynomial commitment protocol combined with homomorphic tags leads to lower communication complexity. Dodis et al [21] generalize the scheme of [9, 10] and introduce the notion of PoR codes, which couples concepts in PoR and hardness amplification. Cash et al. [22] proposed a dynamic PoR scheme that relies on oblivious RAM (ORAM) to perform random access reads and writes in a private manner. Subsequently, Shi et al. [23] proposed a dynamic PoR scheme that considerably improves the performance of [22] by taking advantage of a Merkle tree [24].

2.3.3 PoS with Public Verifiability

Other contributions propose the notion of delegatable verifiability of PoR. For instance, in [25, 26] the authors describe schemes that enable the user to delegate the verification of PoR and to prevent their further re-delegation. Furthermore, Wang et al. [27, 28] proposed

a PoR scheme that supports dynamic operations and public verification on stored data. Their construction uses Merkle trees to support data dynamics. Wang et al. [29] proposed privacy-preserving public auditing for data storage security in cloud computing. Their scheme uses a blinding technique to enable an external auditor to verify the integrity of a file D stored in a server without learning any information about the file contents. Armknecht et al. [30] introduce the notion of outsourced proofs of retrievability, an extension of the PoR model, in which users can task an external auditor to perform and verify PoR on behalf of the data owner. Bowers et al. [31] proposed a theoretical framework of designing PoR protocols. This framework employs two layers of error-correcting codes in order to recover user data from a series of responses.

2.4 Two Proofs of Retrievability Schemes

In this section, we present an overview of two Proofs of Retrievability schemes, namely, Private Compact PoR proposed by Shacham and Waters [10] and StealthGuard proposed by Azraoui et al, [15]. Both schemes serve as building blocks of the protocols presented later in this thesis.

2.4.1 Private Compact PoR

Shacham and Waters [10] proposed two tag-based PoR schemes that rely on linearly-homomorphic tags to minimize the bandwidth required for the transmission of the PoR proof. Thanks to the homomorphic properties of the tags, the cloud storage provider C can aggregate the requested data object symbols and their respective tags into a single symbol-tag pair. The first scheme which is symmetric (privately verifiable), whereas the second scheme is publicly verifiable. In what follows, we briefly describe Private Compact PoR, i.e., the symmetric PoR scheme proposed by Shacham and Waters.

2.4.1.1 Building Blocks

In addition to the erasure code, pseudo-random permutation, and semantically secure encryption (see Section 2.2), Private Compact PoR relies on the following building blocks:

Linearly-homomorphic tags. Linearly-homomorphic tags [8, 10] 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 function (PRF). A pseudo-random function [32] is a function $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{K}$ denotes the set of keys, $\mathcal{X}$ denotes the set of possible inputs, and $\mathcal{Y}$ denotes the set of possible outputs. A pseudo-random function 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 $\mathsf{PRF}(K, X) \in \mathcal{Y}$.
- PRF cannot be distinguished from a random function f, chosen from the uniform distribution of all functions $f : \mathcal{X} \rightarrow \mathcal{Y}$.

2.4.1.2 Protocol Specification

Private Compact PoR is a symmetric PoR scheme, i.e., 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 PoR challenge, and the verification of C's proof. Moreover, in order to reduce the storage overhead inflicted to the cloud storage provider C, this scheme splits the data object $\mathcal{D}$ into n segments each comprising m symbols, and thereafter generates one linearly-homomorphic tag for each segment instead of one tag per symbol.

We now describe the algorithms of Private Compact PoR.

- $\mathsf{SW.KeyGen}\ (1^\lambda) \rightarrow (K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}})$: 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.
- $\mathsf{SW.Encode}\ (K_{\mathsf{u}}, D) \rightarrow (\mathsf{fid}, \mathcal{D})$: 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 $\mathcal{K}$ to permute and encrypt the encoded file, and further outputs

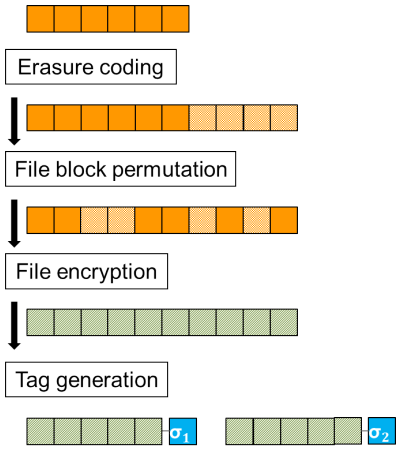

Figure 2.1: Steps of the Encode algorithm of Private Compact PoR

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

$$\mathcal{D} := \big\{\mathsf{fid};\quad \{\hat{d}_{ij}\}_{\substack{1\leq j\leq m\\ 1\leq i\leq n}};\quad \{\sigma_i\}_{1\leq i\leq n}\big\}.$$

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

$$\mathsf{chal} := \big\{(i_c, \nu_c)\big\}_{1\leq c\leq l}.$$

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

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

- 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 $\mathsf{chal} := \{(i_c, \nu_c)\}_{1 \leq c \leq l}$, this algorithm invoked by user U, verifies that the following equation holds:

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

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

2.4.2 StealthGuard

Azraoui et al, [15] proposed StealthGuard: a symmetric watchdog-based PoR scheme. The protocol first splits the file D into equally-sized segments and encodes each segment using an erasure code. After permuting and encrypting D using a semantically secure encryption scheme, a number of pseudo-randomly generated symbols (called watchdogs) are inserted in random positions in each segment. Thanks to the use of semantically secure encryption cloud storage provider C cannot distinguish the watchdogs from the rest of the data object symbols. The PoR challenge chal consists of specifying a subset of the watchdogs and querying them from cloud storage provider C. The watchdogs queries are processed in *privacy-preserving* manner such that cloud storage provider C does not derive any information regarding the positions and values of the watchdogs. V then checks if all the queried watchdogs remain intact. The idea is that if a part of the file is corrupted then, with high probability, some of the queried watchdogs will be corrupted as well.

2.4.2.1 Building Blocks

In order to prepare the verifiable data object $\mathcal{D}$, StealthGuard uses erasure codes, pseudo-random permutation and semantically secure encryption as described in Section 2.2. Furthermore StealthGuard relies on a pseudo-random function (see Section 2.4.1.1) to generate the watchdogs.

Privacy-preserving word search. The Chall and Prove algorithms of StealthGuard leverage a privacy-preserving word search algorithm in order to ensure that the cloud storage provider C cannot determine both the positions of the queried watchdogs and whether these watchdogs are found or not. More specifically, StealthGuard adapts a privacy-preserving word search called PRISM [33] which transforms the search problem into several parallel efficient Private Information Retrieval (PIR) [34] instances. PRISM is defined by three algorithms:

- Algorithm PRISM.Query (invoked by the verifier V) issues a search query for particular watchdog (word) w in a segment.
- Algorithm PRISM.Process (invoked by the cloud storage provider C) processes all the symbols in the segment and constructs C's response. Instead of the watchdog itself, algorithm PRISM.Process retrieves some very short information, called witness, that enables the verifier V to decide about the presence or absence of the queried watchdog in the segment.
- Algorithm PRISM.Analysis (invoked by the verifier V) analyzes C's response and determines whether the queried watchdog w is present in the segment or not.

2.4.2.2 Protocol Specification

StealthGuard is a symmetric PoR scheme. Hence similarly to Private Compact PoR user U and verifier V are the same entity. Thereby, the same user key K_{u} is required for the creation of the data object $\mathcal{D}$, the generation of the PoR challenge, and the verification of C's proof.

We now describe the algorithms of StealthGuard.

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

- SG.Encode $(K_{\mathsf{u}}, D) \rightarrow (\mathsf{fid}, \mathcal{D})$: User U calls this algorithm to prepare her file D for outsourcing to cloud storage provider C. It first divides the encoded file into equally-sized segments $\{D_1, D_2, \ldots, D_n\}$ and applies the erasure code to each segment. Subsequently, algorithm SG.Encode applies a pseudo-random permutation to permute all the symbols in D. Thereafter, SG.Encode encrypts the file D and further uses the secret key K_{u} to determine the value and position of the watchdogs, inserting them within the segments accordingly. Without loss of generality, we denote $\mathcal{D}$ the data object which arises after the insertion of watchdogs.

  Algorithm SG.Encode then picks a unique identifier fid, and terminates its execution and further outsources to cloud storage provider C the verifiable data object $\mathcal{D}$.

- SG.Chall $(K_{\mathsf{u}}, \mathsf{fid}) \rightarrow (\mathsf{chal})$: User U calls this algorithm which chooses γ segments and a watchdog within each of these segments. Thereafter, algorithm SG.Chall uses the underlying PRISM.Query algorithm to issue a privacy preserving search query for each of the selected watchdogs. Algorithm SG.Chall terminates its execution by

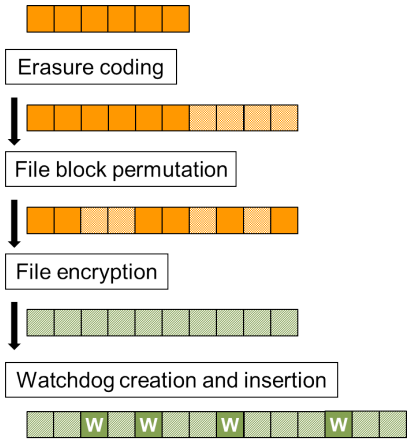

Figure 2.2: Steps of the Encode algorithm of StealthGuard

sending to the cloud storage provider C the challenge chal comprising the γ search queries.

- SG.Prove $(\mathsf{fid}, \mathsf{chal}) \rightarrow (\mathsf{proof})$: Upon receiving the challenge chal the cloud storage provider C invokes this algorithm which uses the underlying PRISM.Process algorithm to construct the response for each of the γ search queries. Thanks to PRISM.Process, cloud storage provider C cannot learn either the content of the search query or the corresponding response. Algorithm SG.Prove terminates its execution by sending to user U the proof proof comprising the responses to γ search queries.

- SG.Verify $(K_{\mathsf{u}}, \mathsf{fid}, \mathsf{chal}, \mathsf{proof}) \rightarrow (\mathsf{dec})$: User U calls this algorithm which uses the underlying PRISM.Analysis algorithm to process all responses included to the proof proof. Algorithm SG.Verify outputs dec := accept if all queried watchdogs are present or dec := reject otherwise.

2.5 Cloud Storage Systems Requirements and PoS

In this section we identify what we believe to be the limitations of current Proof of Storage protocols with respect to two key characteristics of cloud storage systems, namely, *reliable data storage with automatic maintenance*, and *data deduplication*.

2.5.1 Verification of Reliable Data Storage with Automatic Maintenance.

Cloud storage providers currently do not assume any liability for data loss. As a result, users are reluctant to adopt cloud storage services as they lend full control of their data to the cloud provider without the means to verify the correctness of reliable data storage mechanisms deployed by the cloud provider. Hence, technical solutions aiming at establishing users' confidence on the integrity and availability of their data would be highly beneficial both to users' and providers of cloud storage services. As far as the integrity of the outsourced data is concerned, Proof of Storage schemes are fully compatible with reliable data storage mechanisms. The cloud storage system applies its data redundancy and corruption detection mechanisms independently and on top of any PoS processing performed by the user.

Yet, PoS schemes are of limited value when it comes to the audit of reliable data storage mechanisms: indeed, a successful PoS 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. Even in the case of Proof of Retrievability schemes that provide a stronger soundness definition, an unsuccessful verification indicates that the data are not recoverable any more. Detecting that an outsourced file is corrupted is of little help for a client because the latter is no longer retrievable. Despite the fact that PoR schemes rely on erasure codes, the relevant redundancy information is not intended for typical data repair operations: erasure codes assist in realizing the PoR security properties by enabling the recovery of original data from accidental errors that can go undetected from the PoR protocol. However, neither the cloud storage provider C nor the user U can use this redundancy in order to repair corrupted data outsourced to C since, according to the PoR model, C cannot distinguish original data from redundancy information and, U lacks the means to detect any data corruption and repair it.

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 $\mathcal{D}$ and, on the other hand, to guarantee the storage of redundancy the cloud storage provider C 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. 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 verification of reliable data storage criteria. A malicious storage provider can indeed prove the possession of redundancy 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 PoS 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.

2.5.2 Conflict between PoS and deduplication

It seems that the simple composition of Proofs of Retrievability with deduplication is doomed to fail due to some inherent conflict between current PoR and deduplication schemes. The root cause of the conflict is that PoR and deduplication call for diverging objectives: PoR aims at imprinting the data with retrievability guarantees that are unique for each user whereas deduplication tries, whenever feasible, to factor several data segments submitted by different users into a unique copy kept in storage. Indeed, in both watchdog-based and tag-based PoR schemes that insert pseudo-randomly generated watchdogs and append a tag to each data segment (c.f. Section 2.4) respectively, simple composition with deduplication would fail because algorithm Encode of these schemes includes semantically secure encryption by each user, in order to conceal the relation between information and redundancy symbols, that prevents the detection of duplicate data objects. Figure 2.3 depicts the conflict between deduplication and watchdog-based PoR scheme.

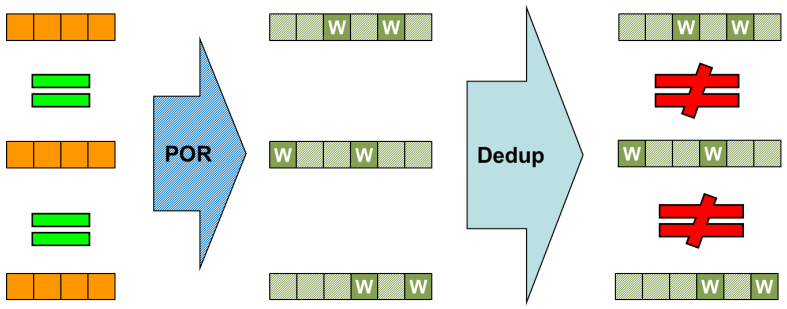

Figure 2.3: Conflict between deduplication and watchdog-based PoR scheme

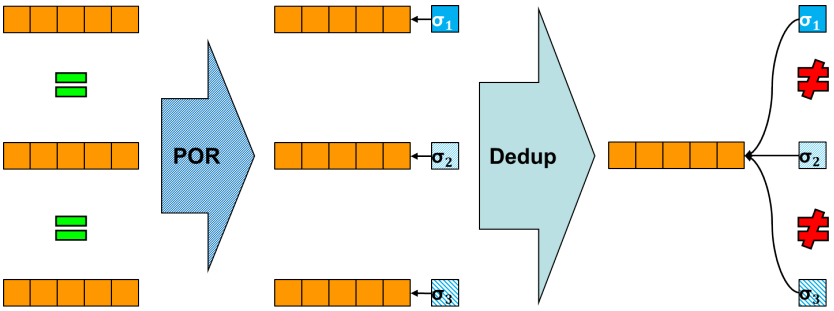

Figure 2.4: Conflict between deduplication and tag-based PoS scheme

Moreover, even other types of tag-based Proof of Storage schemes such as Proofs of Data Possession, where users append a tag to each data symbol are not fully compatible with deduplication (see Figure 2.4). Since the tags are computed under a private key generated by each user, the tags stored with the duplicate copies of identical data objects submitted by different users will still be different; and in case of deduplication, even if storing a single copy for all duplicate data objects would be consistent with respect to basic data management, the PoS scheme would still require that the tags generated by each user for the deduplicated data segment be kept separately in storage. The additional storage resulting from these tags increases very rapidly with the number of users sharing the same data. In the RSA-based PDP scheme described in [8] with segment size 4KB, the storage overhead resulting from PDP tags of 2048 bits each, is 6,25% per user. For example, a 4GB data object uploaded by 50 users would require 312,5% additional storage, that is 12.5GB whereas if tags are to be deduplicated this overhead would remain constant and equal to 256MB only.

2.6 Summary

In this chapter, we presented the concept of Proofs of Storage and a particular flavor of Proofs of Storage schemes named Proofs of Retrievability together with two PoR schemes, namely, Private Compact PoR and StealthGuard, that are uses as building blocks for the protocols we propose later in this thesis. Moreover, we identified what we believe to be the limitations of Proofs of Storage in their current form with respect to two key characteristics of cloud storage systems, namely, reliable data storage with automatic maintenance, and data deduplication.

Now, the reader can either move to Part I where we introduce a comprehensive verifi-

cation scheme that aims to resolve the conflict between reliable data storage verification and automatic maintenance, or Part II where we present as solution that addresses the conflict between Proof of Retrievability schemes and deduplication.

Part I

Proofs of Data Reliability

Chapter 3

Proofs of Data Reliability

In this chapter, 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 of Proofs of Data Reliability and we summarize the state of the art in this field.

3.1 Problem Statement

Data outsourcing raises some unprecedented security issues in the face of potentially misbehaving or malicious cloud service providers. In the case of cloud storage systems, if the misbehavior of a cloud storage provider C goes undetected, nothing prevents a malicious C from "cutting corners" when it comes to reliable data storage mechanisms in order to maximize its returns in terms of storage utilization and thus jeopardizing the integrity and availability of user data. As already discussed in Section 2.5.1, existing verifiable storage solutions like Proofs of Storage suffer from a major shortcoming: they do not take into account a basic feature of storage systems that is *reliable data storage mechanism* implemented by the cloud storage providers. Since data repair functions are part of the basic reliable data storage mechanisms, the assurance of data availability with such services must verify not only the integrity of the data outsourced by the users 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 users' data and the provision by the storage provider for sufficient means to recover from minor failure of storage system components. In addition to the integrity of her 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 user locally

generates the necessary redundancy and subsequently stores the data together with the redundancy on multiple storage nodes. In Section 2.5.1 we established that, in the case of erasure code-based storage systems, the relation between the data and redundancy symbols should stay hidden from the cloud storage provider; otherwise, the latter could store a portion of the encoded data only and compute any missing symbols upon request on-the-fly. Similarly, in the case of replication based storage systems, each storage node should store a different replica; otherwise, the cloud storage provider could simply store a single replica. As a result, the naive approach of designing a Proof of Data Reliability scheme requires some interaction with the user, in order to repair damaged data. Hence these designs are at odds with *automatic maintenance* that is a key feature of cloud storage systems. Thus, in the adversarial setting of cloud storage systems, 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.

3.2 Definition of a Proof of Data Reliability Protocol

In this section, we give a formal definition of a Proof of Data Reliability protocol, inspired by the definition proposed respectively by Chen et al. [35, 36] and Armknecht et. al [37].

3.2.1 Environment

A Proof of Data Reliability scheme is a PoS scheme and hence defines the three following parties:

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.

Similarly to Proofs of Storage, 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$. We consider two different settings for a Proof of Data Reliability scheme: a private one where U is the verifier V and a public one where V can be any third party except C.

3.2.2 Formal Definition

Compared to PoS, a Proof of Data Reliability scheme introduces three new algorithms, namely, Setup, GenR, and Repair. These algorithms allow for the verification of the reliable data storage mechanisms. In particular, algorithm Setup determines the redundancy mechanism as well as some key parameters of the cloud storage system such that the reliability guarantee t is satisfied. Algorithm GenR generates the required redundancy for the data object $\mathcal{D}$ in a manner that the former can be efficiently verified. Lastly, algorithm Repair facilitates the necessary data repair operations upon the detection of data loss or corruption. Additionally, a Proof of Data Reliability scheme consolidates the PoS algorithms KeyGen and Encode into algorithm Store which outputs the verifiable data object $\mathcal{D}$ together with all the necessary keying material.

Definition 4. (Proof of Data Reliability scheme). *A Proof of Data Reliability scheme is defined by seven polynomial time algorithms:*

- 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 redun-*

dancy 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.*

Similarly to a Proof of Storage scheme, a Proof of Data Reliability scheme should be *correct* and *sound*.

3.2.3 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 5. (*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* $\mathsf{chal} \leftarrow \mathsf{Chall}\ (\mathcal{K}, \mathsf{param}_{\mathsf{system}})$*:*

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

3.2.4 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*. Similarly to a PoR scheme, 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.

Extractability. It is essential for any Proof of Data Reliability scheme to ensure that an honest user U can recover her file D with high probability. This guarantee is formalized using the notion of extractability introduced in Section 2.2.2. If 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. We adapt the retrievability game between an adversary $\mathcal{A}$ and an environment (c.f. Section 2.2.2) to the Proof of Data Reliability definition. The environment simulates all honest users and an honest verifier, and it further provides $\mathcal{A}$ with access to oracles $\mathcal{O}_{\mathsf{Store}}$, $\mathcal{O}_{\mathsf{Chall}}$, and $\mathcal{O}_{\mathsf{Verify}}$: 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:

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

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 6. (*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* 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* Extract *recovers D^* from C' with overwhelming probability.*

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

where ϵ is a negligible function in security parameter λ.

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 7. (*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.*

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 [37]. 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 8. (*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).*

3.2.5 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ 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 convince verifier V with high probability that the o 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*.

3.3 State of the Art on Proofs of Data Reliability

In this section, we summarize Proof of Data Reliability solutions that provide mechanisms to enable the verification of reliable data storage.

Replication-based Proofs of Data Reliability. Curtmola et al. proposed a multi-replica Proof of Data Possession protocol (MR-PDP) [38], 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. [8] 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 et al. [39] 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 a 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 et al. [40,41] 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 [39], 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 [39] with the addition of the use of the Merkle hash trees to verify that the proof is computes over the updated data.

Etemad et al. [42] 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. [18] 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 [18] 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 et al. [35,36], 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 [43]: 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 et al. [44] extend the scheme in [36] 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 [37] propose a multi-replica Proof of Retrievability scheme that delegates the replica construction to the cloud storage provider. Similarly to [35], 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. [45] propose HAIL: a 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. [46] 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.

In [47], Chen et al. 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 [45] 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. The work in [48] extends the scheme in [47] by leveraging a more efficient homomorphic tag scheme called SpaceMac [49] 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, the authors in [50] 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. [51] 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 protocol presented [51] 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.

3.3.1 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 [37,51] or "semi-automatic" maintenance by the cloud storage provider [36, 44] 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 [51] enables the cloud storage provider to autonomously repair corrupted codeword symbols, its adversarial model does not guarantee the extractability of an outsourced file.

In Chapters 4 and 5 we propose two Proof of Data Reliability schemes that succeed in verifying reliable data storage mechanisms and at the same time enable the cloud storage provider to autonomously perform automatic maintenance operations.

Chapter 4

POROS: Proof of Data Reliability for Outsourced Storage

In this chapter, we propose POROS, a Proof of Data Reliability scheme, that on the one hand, enables a client 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. We analyze the security of POROS both theoretically and through experiments measuring the time difference between an honest cloud and some malicious adversaries. Finally, we propose an extended version of POROS where clients can run multiple instances of the challenge-response protocol in order to increase their trust in the storage service.

4.1 Introduction of POROS

Our goal is to enable a cloud storage system to provide data reliability guarantees without disrupting its automatic maintenance operations, in spite of the conflict between the latter and Proof of Storage schemes that were described in Section 2.5.1. The root cause of this conflict is the simple fact that the redundancy information is a function of the original data itself: automatic maintenance mechanisms require that the cloud storage system has unobstructed access to redundancy in order to repair corrupted data; unfortunately, as already discussed in Section 2.5.1, this presents a malicious cloud storage provider C the opportunity to gain storage savings by actually not storing the redundancy whilst it positively meets the Proof of Data Reliability criteria (c.f. Section 3.2). Hence, a new approach that treats redundancy separately from the original data, and aims at ensuring users that the redundancy is actually kept in storage, seems to be the right way to resolve this conflict. Such assurance renders harmless the disclosure of the relationship

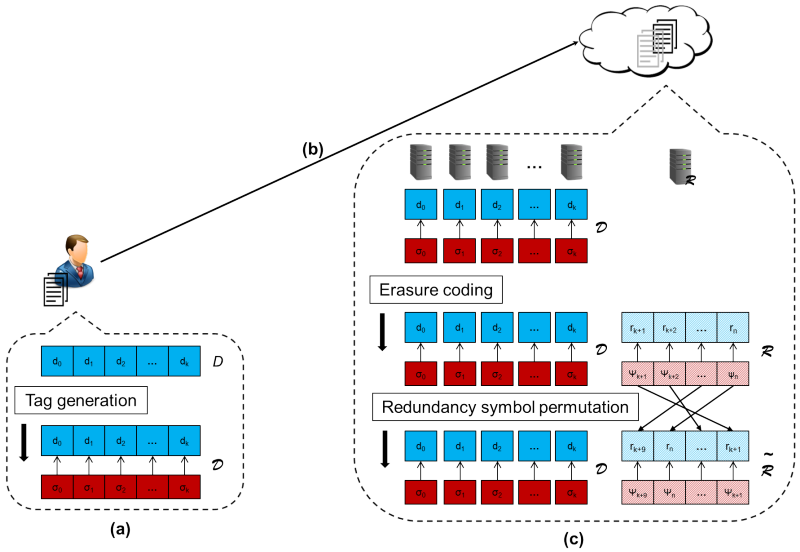

Figure 4.1: Overview of POROS outsourcing process: (a) The user U computes the linearly-homomorphic tags for the original data symbols; (b) U outsources the data object $\mathcal{D}$ to cloud storage provider C; (c) Using $\mathbf{G}$, C applies the systematic erasure code on both data symbols and their tags yielding the redundancy symbols and their corresponding tags; thereafter, C permute all redundancy symbols and derives the redundancy object $\tilde{\mathcal{R}}$.

between original data and redundancy to the cloud storage provider C, thus, allowing for effective automatic maintenance operations without user interaction. Furthermore, the use of reliable data storage mechanisms by C also facilitates the outsourcing of the computation of redundancy information: C is the party that invokes algorithm GenR. As a result, the outsourcing phase of our Proof of Data Reliability scheme becomes lighter for users since they no longer have to perform this operation. However, a new security concern arises as users now require some means to verify the correct computation of the redundancy information (c.f. *Req 2* in Section 3.2.4) by this untrusted C.

With respect to the integrity verification of the original data (c.f. *Req 1* in Section 3.2.4), our scheme leverages the *linearly-homomorphic tags* used in the Private Compact PoR scheme proposed by Shacham and Waters (c.f. Section 2.4.1) 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 4.1(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 4.1(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 4.1(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 3.2.4). 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 $\mathsf{T_{thr}}$, such that whenever C receives a Proof of Data Reliability challenge, it is compelled to generate and deliver the proof before the time-threshold $\mathsf{T_{thr}}$ 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 4.1(c)). Similarly to the permutation-based hourglass scheme in [43], 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 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 $\mathsf{T_{thr}}$ is thus defined as a function of time $\mathsf{T_h}$ that an *honest* cloud storage provider C takes to generate the Proof of Reliability and, time $\mathsf{T_m}$ 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 $\mathsf{T_{thr}}$ should satisfy the inequality $\mathsf{T_h} < \mathsf{T_{thr}} \ll \mathsf{T_m}$, implying that the time required for the proof generation is much shorter than the time needed to compute the redundancy.

Organization. The rest of this chapter is organized as follows. In Section 4.2, we provide the specification of POROS. We analyze its security in section Section 4.3, and in

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

Section 4.4 we describe an extended version of our protocol that allows users to challenge the cloud storage provider multiple times while preventing the latter to parallelize its responses.

4.2 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.

4.2.1 Building Blocks

POROS relies on the following building blocks.

MDS codes. *Maximum distance separable* (MDS) codes [12, 52] 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 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 [12] 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 [53].

Linearly–homomorphic tags. POROS's integrity guarantee (c.f. *Req 1* in Section 3.2.4) derives from the use of the linearly-homomorphic tags proposed by Shacham and Waters, for the design of the Private Compact PoR scheme (c.f. Section 2.4.1). This scheme makes use of a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{Z}_p$, 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 $\big(d^{(1)}, \ldots, d^{(k)} \mid r^{(k+1)}, \ldots, r^{(n)}\big)$, where $d^{(j)}$ (for $1 \leq j \leq k$), denote the original data symbols and $r^{(j)}$ (for $k+1 \leq j \leq n$), 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:

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

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

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

The verifier V checks that the following equation holds:

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

As regards to the verification of the redundancy symbols, we observe that $r^{(j)} := \mathbf{d} \cdot \mathbf{G}^{(j)}$, where vector $\mathbf{d} := \big(d^{(1)}, \ldots, d^{(k)}\big)$ denotes the vector of data symbols and $\mathbf{G}^{(j)}$ denotes the j^{th} column of generator matrix $\mathbf{G}$, for $k+1 \leq j \leq n$. 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 $\mathsf{chal} := \big\{(j_c, \nu_c)\big\}_{1 \leq c \leq l}$ as follows:

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

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

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

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

Pseudo-random permutation PRP. POROS uses a pseudo-random permutation (c.f. Section 2.2) in order to permute the redundancy symbols and construct the redundancy object $\tilde{\mathcal{R}}$.

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 3.2.4).

4.2.2 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.

- $\mathsf{Setup}\ (1^\lambda, t) \rightarrow \big(\big\{\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}, \mathcal{S}_{\mathcal{R}}\big\}, \mathsf{param}_{\mathsf{system}}\big)$: 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 $t \leq n - k + 1$. In addition, algorithm Setup chooses k storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}$ that are going to store the data object $\mathcal{D}$ and one storage node $\mathcal{S}_{\mathcal{R}}$ 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 $\big\{\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}, \mathcal{S}_{\mathcal{R}}\big\}$.

- $\mathsf{U.Store}\ (1^\lambda, D, \mathsf{param}_{\mathsf{system}}) \rightarrow (K_{\mathsf{u}}, \mathcal{D}, \mathsf{param}_{\mathcal{D}})$: 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 $1 \leq i \leq s$ and $1 \leq j \leq k$.

  Algorithm U.Store also picks a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{Z}_p$, together with its pseudo-randomly generated key $\mathrm{k}_{\mathsf{prf}} \in \{0,1\}^\lambda$, and a non-

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
$\alpha, \mathrm{k}_{\mathsf{prf}}$	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
$\mathtt{T}_{\mathsf{thr}}$	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 4.1: Notation used in the description of POROS.

zero element $\alpha \xleftarrow{R} \mathbb{Z}_p$. Hereafter, U.Store computes for each data symbol a *linearly homomorphic* MAC as follows:

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

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

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

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

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

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

and the data object parameters

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

- $\mathsf{C.GenR}\,(\mathcal{D}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\tilde{\mathcal{R}})$: Upon reception of data object $\mathcal{D}$, algorithm $\mathsf{C.GenR}$ starts computing the redundancy symbols $\{r_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ 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 $\{\psi_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ are the *linearly-homomorphic* authenticators of $\{r_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$.

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 $\mathrm{k}_{\mathsf{prp}}$ in order to permute both the redundancy symbols $\{r_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ and the corresponding homomorphic-tags $\{\psi_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$ yielding the redundancy object

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

More precisely, if we denote $(\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_{(n-k)s})$ the vector of the permuted redundancy symbols $\{\tilde{r}_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$, then the redundancy symbol $r_i^{(j)}$ is mapped to the position $\mathsf{PRP}(\mathrm{k}_{\mathsf{prp}}, (i-1)(n-k)+j)$ 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 $\mathsf{PRP}(\mathrm{k}_{\mathsf{prp}}, (i-1)(n-k)+j)$.

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

- $\mathsf{U.Chall}\ (\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{chal})$: Provided with the object identifier fid, the secret key K_{u}, and the system parameters $\mathsf{param}_{\mathsf{system}}$, algorithm $\mathsf{U.Chall}$ generates a vector $\boldsymbol{\nu} := (\nu_c)_{c=1}^{l}$ 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 $\{d_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}}$, and picks one random index $1 \le c_r \le (n-k)s - l$. Then, algorithm $\mathsf{U.Chall}$ terminates by sending C the challenge

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

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

  Thereupon, algorithm $\mathsf{C.Prove}$ 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

$$\mu = \hat{\mathbf{d}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \hat{d}_c \nu_c \tag{4.1}$$

$$\tau = \hat{\boldsymbol{\sigma}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \hat{\sigma}_c \nu_c \tag{4.2}$$

  In the same manner, algorithm $\mathsf{C.Prove}$ 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

$$\tilde{\mu} = \tilde{\mathbf{r}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{r}_{(c_r+c-1)} \nu_c \tag{4.3}$$

$$\tilde{\tau} = \tilde{\boldsymbol{\psi}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{\psi}_{(c_r+c-1)} \nu_c \tag{4.4}$$

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 $\mathsf{chal} = (\mathsf{fid},\ (\boldsymbol{i_d},\ i_r,\ \boldsymbol{\nu}))$, 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 $\mathsf{T}_{\mathsf{thr}}$. If not algorithm U.Verify outputs reject; otherwise it executes the next step.
  - *Data possession verification:* Given vector $\boldsymbol{\nu} = (\nu_1, ..., \nu_l)$ and vector $\boldsymbol{c_d} := (i_c, j_c)_{c=1}^{l}$ algorithm U.Verify verifies whether

$$\tau = \alpha\mu + \sum_{c=1}^{l} \nu_c \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, (i_c - 1)k + j_c) \tag{4.5}$$

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

$$\tilde{\tau} = \alpha\tilde{\mu} + \sum_{c=1}^{l} \nu_c \mathbf{P}^{y_c} \cdot \mathsf{prf}_{(x_c)} \tag{4.6}$$

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

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

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

- C.Repair $({}^*\mathcal{D}, J_{\mathsf{f}}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}, \mathsf{maskGen}) \rightarrow (\mathcal{D}, \perp)$: On input of a corrupted data object ${}^*\mathcal{D}$ and a set of failed storage node indices $J_{\mathsf{f}} \subseteq [1, k]$, algorithm 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 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 $\mathcal{D}$.

4.3 Security Evaluation

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

4.3.1 Correctness

We now show that verification Equations 4.5 and 4.6 always hold when algorithm $\mathsf{C.Prove}$ is executed correctly. Subsequently we argue that if time threshold $\mathsf{T_{thr}}$ 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:

$$\hat{\sigma}_c = \alpha\hat{d}_c + \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, (i_c-1)k + j_c),\ \forall\ 1 \le c \le l \tag{4.7}$$

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 $1 \le c \le l$ 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 $(x_c, y_c) = \mathsf{PRP}^{-1}(\mathrm{k}_{\mathsf{prf}}, c_r + c - 1)$ and $\mathbf{P}^{y_c}$ is the ${y_c}^{\text{th}}$ column of the linear code matrix $\mathbf{P}$. Therefore, the following equality always holds.

$$\tilde{\psi}_{c_r+c-1} = \mathbf{P}^{y_c} \cdot (\alpha\mathbf{d}_{(x_c)} + \mathsf{prf}_{(x_c)}) = \alpha\tilde{r}_{c_r+c-1} + \mathbf{P}^{y_c} \cdot \mathsf{prf}_{(x_c)} \tag{4.8}$$

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

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

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

Where $\boldsymbol{\nu} = (\nu_1, \nu_2, \ldots, \nu_l)$ is the random vector generated by the client and transmitted in the challenge message chal.

By plugging Equations 4.7 and 4.8 in the inner products, we derive the following equalities:

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

We can easily see that the above equations are the same as Equations 4.5 and 4.6. This means that if the cloud server executes algorithm C.Prove correctly, then it will pass the verification so long as *its response time is smaller than time threshold* $\mathsf{T_{thr}}$.

4.3.2 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 (c.f. Section 2.4.1) 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 of the Proof of Retrievability model (c.f. Section 2.2), 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 [10] 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 $\mathsf{Extract}$ 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 computation.** From the definition of linearly-homomorphic tags (c.f. Section 2.4.1.1), 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

$$\tilde{\mu} = \tilde{\mathbf{r}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{r}_{(c_r+c-1)} \nu_c \tag{4.9}$$

$$\tilde{\tau} = \tilde{\boldsymbol{\psi}} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{\psi}_{(c_r+c-1)} \nu_c \tag{4.10}$$

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 $\mathsf{C.GenR}$.

Req 3 : **Storage allocation commitment.** Similarly to [43], storage allocation commitment is met as long as $\mathsf{T_{thr}} \ll \mathsf{T_m}$, where $\mathsf{T_m} = min(\mathsf{T_{m_1}}, \mathsf{T_{m_2}})$ is the response time of a malicious C where $\mathsf{T_m}$ is defined as the minimum of the following values:

- $\mathsf{T_{m_1}}$: the response time of a malicious C who stores the redundancy information in its original order (i.e. without permutation).

$$\mathsf{T_{m_1}} = l\mathsf{T_{Seek}} + l\mathsf{T_{SeqRead}}(1)$$

- $\mathsf{T_{m_2}}$: the response time of a malicious C who stores the data object $\mathcal{D}$, only.

$$\mathsf{T_{m_2}} = l\mathsf{T_{Seek}} + l\mathsf{T_{SeqRead}}(k) + l\mathsf{T_{Encode}}$$

In the above equations, l is the number of sequential redundancy blocks $\tilde{r}$ requested in a POROS challenge, $\mathsf{T_{Seek}}$ is the time required for a seek operation on the hard drive, $\mathsf{T_{SeqRead}}(n)$ is the required time to read n data blocks sequentially from the hard disk, $\mathsf{T_{Encode}}$ 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, $\mathsf{T_{m_1}}$ should be less than $\mathsf{T_{m_2}}$.

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

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

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

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

Section 4.3.3 provides some hints on the order of $\mathsf{T}_{\mathsf{thr}}$ through an experimental study.

4.3.3 Evaluation

We have performed an experimental evaluation of POROS' $\mathsf{C.Prove}$ 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 $\mathbf{G}$ 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

[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++/

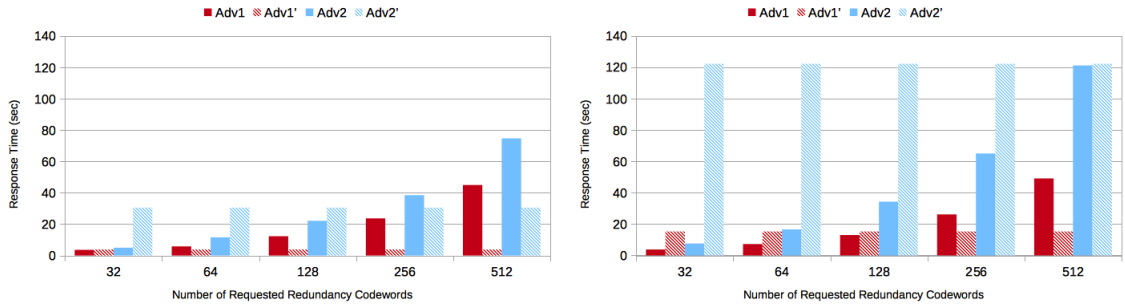

Figure 4.2: Response times of adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ for different challenge sizes l; data object size of 4GB (left) vs. 16GB (right).

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

$\mathbf{r}_i$ 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:

- $\mathcal{A}'_1$ 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 4.2 and Tables 4.2 and 4.3 are the median of 20 independent measurements of the cloud response time; before each measurement we flushed all file system caches.

Figure 4.2 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 4.2 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

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

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 [43], 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 4.3 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, $\mathsf{T}_{\mathsf{thr}}$ 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.

4.4 Multiple-challenge POROS

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

To address these shortcomings, we propose that U 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 4.2). Subsequent challenges however will be produced as a function of the proofs to preceding challenges. More specifically, the challenge in iteration $\mathsf{t}+1$, 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 U and C minimal (i.e. there are only two rounds of communication);

(ii) and induces C to generate the proofs iteratively.

4.4.1 Description

Multiple-challenge POROS runs as following:

- $\mathsf{Setup}\ (1^\lambda, t) \rightarrow (\{\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}, \mathcal{S}_\mathcal{R}\}, \mathsf{param}_{\mathsf{system}})$: 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 $t \leq n-k+1$. In addition, algorithm Setup chooses k storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}$ that are going to store the data object $\mathcal{D}$ and one storage node $\mathcal{S}_\mathcal{R}$ 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 $\{\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}, \mathcal{S}_\mathcal{R}\}$.

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

  Algorithm $\mathsf{U.Store}$ also picks a pseudo-random function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{Z}_p$, together with its pseudo-randomly generated key $\mathrm{k}_\mathsf{prf} \in \{0,1\}^\lambda$, 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:

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

  In addition, algorithm $\mathsf{U.Store}$ produces a pseudo-random permutation $\mathsf{PRP} : \{0,1\}^\lambda \times [(n-k)s] \rightarrow [(n-k)s]$, together with its pseudo-randomly generated key $\mathrm{k}_\mathsf{prp} \in \{0,1\}^\lambda$. Algorithm $\mathsf{U.Store}$, also picks a pseudo-random function $\mathsf{PRF_{chal}} : \mathbb{Z}_p \times \mathbb{Z}_p \rightarrow \{0,1\}^\lambda$ and a pair of pseudo-random generators ($\mathsf{PRG_i} : \{0,1\}^\lambda \rightarrow [s]^l$, $\mathsf{PRG_j} : \{0,1\}^\lambda \rightarrow [k]^l$). Finally, algorithm $\mathsf{U.Store}$ picks a unique identifier fid.

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

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

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

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

  and the data object parameters

$$\mathsf{param}_\mathcal{D} := ((\mathsf{PRP}, \mathrm{k}_\mathsf{prp}), \mathsf{PRF_{chal}}, (\mathsf{PRG_i}, \mathsf{PRG_j}))\,.$$

- $\mathsf{C.GenR}\ (\mathcal{D}, \mathsf{param_{system}}, \mathsf{param}_\mathcal{D}) \rightarrow (\tilde{\mathcal{R}})$: 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 $\mathbf{d}_i := \big(d_i^{(1)}, \ldots, d_i^{(k)}\big)$ with the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

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

  Similarly, algorithm $\mathsf{C.GenR}$ multiplies the vector of linearly-homomorphic tags $\boldsymbol{\sigma}_i :=$

$\big(\sigma_i^{(1)}, \ldots, \sigma_i^{(k)}\big)$ with $\mathbf{G}$:

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

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

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

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

More precisely, if we denote $(\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_{(n-k)s})$ the vector of the permuted redundancy symbols $\{\tilde{r}_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}$, then the redundancy symbol $r_i^{(j)}$ is mapped to the position $\mathsf{PRP}(\mathsf{k_{prp}}, (i-1)(n-k)+j)$ 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 $\mathsf{PRP}(\mathsf{k_{prp}}, (i-1)(n-k)+j)$.

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

- $\mathsf{U.Chall}\ (\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param_{system}}) \to (\mathsf{chal})$: Provided with the object identifier fid, the secret key K_{u}, and the system parameters $\mathsf{param_{system}}$, algorithm $\mathsf{U.Chall}$ generates a vector $\boldsymbol{\nu} := (\nu_c)_{c=1}^{l}$ of l random elements in $\mathbb{Z}_p$, selects an integer v which specifies the number of iterations that C is required to go through,generates a random seed $\eta^{(1)} \in \{0,1\}^{\lambda}$, and picks one random index $1 \le c_r^{(1)} \le (n-k)s - l$. Then, algorithm $\mathsf{U.Chall}$ terminates by sending C the challenge

$$\mathsf{chal} := \big(\mathsf{fid}, \big(\mathsf{v},\ \eta^{(1)},\ c_r^{(1)},\ \boldsymbol{\nu}\big)\big).$$

- $\mathsf{C.Prove}\ (\mathsf{chal}, \tilde{\mathcal{D}}, \mathsf{param}_{\mathcal{D}}) \to (\mathsf{proof})$: On receiving challenge $\mathsf{chal} = \big(\mathsf{fid},\ (\mathsf{v},\ \eta^{(1)},\ c_r^{(1)},\ \boldsymbol{\nu})\big)$, algorithm $\mathsf{C.Prove}$ first retrieves the authenticated data object $\mathcal{D}$ and the corresponding authenticated redundancy $\tilde{\mathcal{R}}$ that match identifier fid.

  Thereupon, algorithm $\mathsf{C.Prove}$ processes $\mathcal{D}$ and $\tilde{\mathcal{R}}$ by executing the following operations v times. For ease of exposition, we assume that algorithm $\mathsf{C.Prove}$ is at the t^{th} iteration:

1. Algorithm $\mathsf{C.Prove}$ first derives the indices of the requested data symbols and their respective tags $\boldsymbol{c_d}^{(\mathsf{t})} := (\mathsf{PRG_i}(\eta^{(\mathsf{t})}), \mathsf{PRG_j}(\eta^{(\mathsf{t})})$.

2. It reads the l requested blocks defined by $\boldsymbol{c_d}^{(\mathsf{t})}$. Without loss of generality, we denote these blocks $\hat{\mathbf{d}}^{(\mathsf{t})} := (\hat{d}_1^{(\mathsf{t})}, \hat{d}_2^{(\mathsf{t})}, \ldots, \hat{d}_l^{(\mathsf{t})})$.

3. It reads the l tags associated with blocks $\hat{\mathbf{d}}$. We denote these tags $\hat{\boldsymbol{\sigma}}^{(\mathsf{t})} := (\hat{\sigma}_1^{(\mathsf{t})}, \hat{\sigma}_2^{(\mathsf{t})}, \ldots, \hat{\sigma}_l)^{(\mathsf{t})}$.

4. It computes the inner products

$$\mu^{(\mathsf{t})} = \hat{\mathbf{d}}^{(\mathsf{t})} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \hat{d}_c^{(\mathsf{t})} \nu_c \tag{4.11}$$

$$\tau^{(\mathsf{t})} = \hat{\boldsymbol{\sigma}}^{(\mathsf{t})} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \hat{\sigma}_c^{(\mathsf{t})} \nu_c \tag{4.12}$$

5. It reads l consecutive redundancy blocks starting from block $\tilde{r}_{c_r^{(\mathsf{t})}}$. Let $\tilde{\mathbf{r}}^{(\mathsf{t})}$ denote the l consecutive redundancy blocks $(\tilde{r}_{c_r^{(\mathsf{t})}}, ..., \tilde{r}_{(c_r^{(\mathsf{t})}+l-1)})$.

6. It reads the l consecutive homomorphic MACs associated with redundancy blocks $\tilde{\mathbf{r}}^{(\mathsf{t})}$. Let $\tilde{\boldsymbol{\psi}}^{(\mathsf{t})} := (\tilde{\psi}_{c_r^{(\mathsf{t})}}, ..., \tilde{\psi}_{(c_r^{(\mathsf{t})}+l-1)})$ denote these MACs.

7. It computes the inner products

$$\tilde{\mu}^{(\mathsf{t})} = \tilde{\mathbf{r}}^{(\mathsf{t})} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{r}_{(c_r^{(\mathsf{t})}+c-1)} \nu_c \tag{4.13}$$

$$\tilde{\tau}^{(\mathsf{t})} = \tilde{\boldsymbol{\psi}}^{(\mathsf{t})} \cdot \boldsymbol{\nu} = \sum_{c=1}^{l} \tilde{\psi}_{(c_r^{(\mathsf{t})}+c-1)} \nu_c \tag{4.14}$$

8. It computes the seed for the next iteration $\eta^{(\mathsf{t}+1)} := \mathsf{PRF_{chal}}(\mu^{(\mathsf{t})}, \tau^{(\mathsf{t})})$. It also computes the random value $I_{c_r^{(\mathsf{t}+1)}} := \mathsf{PRF_{chal}}(\tilde{\mu}^{(\mathsf{t})}, \tilde{\tau}^{(\mathsf{t})})$ and sets the new index $c_r^{(\mathsf{t}+1)}$ for the next challenge to the first $\log((n-k)s - l)$ bits of $I_{c_r^{(\mathsf{t}+1)}}$.

9. It goes back to step 1.

Finally, algorithm $\mathsf{C.Prove}$ terminates its execution by returning the proof

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

such that:

$$\boldsymbol{\mu} = (\mu^{(1)}, \mu^{(2)}, \ldots, \mu^{(\mathsf{v})}) \; ; \; \boldsymbol{\tau} = (\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(\mathsf{v})})$$
$$\tilde{\boldsymbol{\mu}} = (\tilde{\mu}^{(1)}, \tilde{\mu}^{(2)}, \ldots, \tilde{\mu}^{(\mathsf{v})}) \; ; \; \tilde{\boldsymbol{\tau}} = (\tilde{\tau}^{(1)}, \tilde{\tau}^{(2)}, \ldots, \tilde{\tau}^{(\mathsf{v})})$$

- $\mathsf{U.Verify}\ (K_{\mathsf{u}}, \mathsf{chal}, \mathsf{proof}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{dec})$: On input of user key $K_{\mathsf{u}} = (\alpha, \mathrm{k}_{\mathsf{prf}})$, challenge $\mathsf{chal} = \Big(\mathsf{fid},\ (\mathsf{v},\ \eta^{(1)},\ c_r^{(1)},\ \boldsymbol{\nu})\Big)$, proof $\mathsf{proof} := \{(\boldsymbol{\mu}, \boldsymbol{\tau}),\ (\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\tau}})\}$, and data object parameters $\mathsf{param}_{\mathcal{D}}$ algorithm $\mathsf{U.Verify}$ performs the following checks:
  - *Response time verification:* It first checks whether the response time of the server was under time threshold $\mathsf{T}_{\mathsf{thr}}$. If not algorithm $\mathsf{U.Verify}$ outputs reject; otherwise it executes the next step.
  - *Data possession verification.* Given vectors $\boldsymbol{\mu} = (\mu^{(1)}, \mu^{(2)}, \ldots, \mu^{(\mathsf{v})})$, and $\boldsymbol{\tau} = (\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(\mathsf{v})})$, algorithm $\mathsf{U.Verify}$ executes the subsequent steps v times. We assume here that algorithm $\mathsf{U.Verify}$ is at the t^{th} iteration:
    1. Given vector $\boldsymbol{\nu} = (\nu_1, ..., \nu_l)$ and vector $\boldsymbol{c_d}^{(\mathsf{t})} := (i_c^{(\mathsf{t})}, j_c^{(\mathsf{t})})_{c=1}^{l}$ algorithm $\mathsf{U.Verify}$ verifies whether
    $$\tau^{(\mathsf{t})} = \alpha\mu^{(\mathsf{t})} + \sum_{c=1}^{l} \nu_c \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, (i_c^{(\mathsf{t})} - 1)k + j_c^{(\mathsf{t})}) \tag{4.15}$$
    If it is not the case, algorithm $\mathsf{U.Verify}$ returns reject; otherwise it moves onto verifying the integrity of the redundancy.
    2. Otherwise, algorithm $\mathsf{U.Verify}$ generates the seed $\eta^{(\mathsf{t}+1)} := \mathsf{PRF}_{\mathsf{chal}}(\mu^{(\mathsf{t})}, \tau^{(\mathsf{t})})$, sets the indices for the next iteration to $\boldsymbol{c_d}^{(\mathsf{t}+1)} := (\mathsf{PRG}_{\mathsf{i}}(\eta^{(\mathsf{t}+1)}), \mathsf{PRG}_{\mathsf{j}}(\eta^{(\mathsf{t}+1)})$, and goes to step 1.

  If algorithm $\mathsf{U.Verify}$ does not return reject, it proceeds with verifying the integrity of the redundancy.
  - *Redundancy possession verification.* Given the vectors $\tilde{\boldsymbol{\mu}} = (\tilde{\mu}^{(1)}, \tilde{\mu}^{(2)}, \ldots, \tilde{\mu}^{(\mathsf{v})})$ and $\tilde{\boldsymbol{\tau}} = (\tilde{\tau}^{(1)}, \tilde{\tau}^{(2)}, \ldots, \tilde{\tau}^{(\mathsf{v})})$ it performs the following operations v times.
    1. Given index $c_r^{(\mathsf{t})}$, algorithm $\mathsf{U.Verify}$ finds the shuffling function preimage $(x_c^{(\mathsf{t})}, y_c^{(\mathsf{t})}) = \mathsf{PRP}^{-1}(c_r^{(\mathsf{t})} + c - 1)$ for all $1 \leq c \leq l$.
    2. Given 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 equality holds:
    $$\tilde{\tau}^{(\mathsf{t})} = \alpha\tilde{\mu}^{(\mathsf{t})} + \sum_{c=1}^{l} \nu_c \mathbf{P}^{y_c^{(\mathsf{t})}} \cdot \mathsf{prf}_{(x_c^{(\mathsf{t})})}$$

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

$$\mathsf{prf}_{(x_c^{(\mathsf{t})})} = \Big(\mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, (x_c^{(\mathsf{t})} - 1)k + 1), \ldots, \mathsf{PRF}(\mathrm{k}_{\mathsf{prf}}, x_c^{(\mathsf{t})}k)\Big)$$

If the equality is not satisfied, then algorithm U.Verify returns reject.

3. Otherwise, it computes $I_{c_r^{(\mathsf{t}+1)}} := \mathsf{PRF}_{\mathsf{chal}}(\tilde{\mu}^{(\mathsf{t})}, \tilde{\tau}^{(\mathsf{t})})$, defines the new index $c_r^{(\mathsf{t}+1)}$ for the next iteration by truncating the first $\log((n-k)s - l)$ bits of $I_{c_r^{(\mathsf{t}+1)}}$, and goes to step 1.

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

- C.Repair $({}^*\mathcal{D}, J_{\mathsf{f}}, \mathsf{param}_{\mathsf{system}}, \mathsf{param}_{\mathcal{D}}, \mathsf{maskGen}) \rightarrow (\mathcal{D}, \perp)$: On input of a corrupted data object ${}^*\mathcal{D}$ and a set of failed storage node indices $J_{\mathsf{f}} \subseteq [1, k]$, algorithm 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 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 $\mathcal{D}$.

4.5 Conclusion

In this section, we have introduced a new Proof of Data Reliability solution named POROS that 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. Running the POROS protocol, a client is assured that the cloud server actually stores both the original data and the corresponding redundancy information. 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.

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 the next section we introduce a new Proof of Data Reliability scheme which neither makes any assumptions regarding the underlying storage medium technology nor deviates from the current distributed storage architecture.

Chapter 5

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

In this chapter, 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 (c.f. Chapter 4), without the shortcomings of the latter. We analyze the security of the protocol and we show that PORTOS is secure against a *rational* adversary. Moreover, we evaluate the performance of PORTOS in terms of storage, communication, and verification cost. Finally, we propose 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.

5.1 Introduction of PORTOS

PORTOS is a Proof of Data Reliability scheme designed for distributed cloud storage systems. 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.

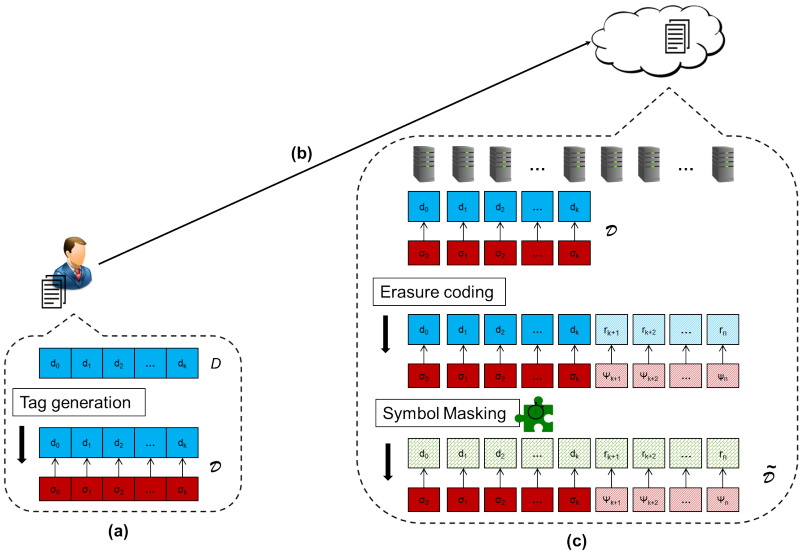

Figure 5.1: Overview of PORTOS outsourcing process: (a) The user U computes the linearly-homomorphic tags for the original data symbols; (b) U outsources the data object $\mathcal{D}$ to cloud storage provider C; (c) Using G, C applies the systematic erasure code on both data symbols and their tags yielding the redundancy symbols and their corresponding tags; thereafter, C derives the masking coefficients and masks all data and redundancy symbols.

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 4.2.1). More specifically, this PDP scheme relies on linearly-homomorphic tags of Private Compact PoR (c.f. Section 2.4.1) 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 5.1 depicts the outsourcing process of PORTOS.

Organization. The remaining of this chapter is organized as follows: In Section 5.2, we provide the specification of PORTOS. We analyze its security in Section 5.3 and we evaluate its performance in Section 5.4. Finally, in Section 5.5 we describe a more efficient version of our protocol.

5.2 PORTOS

In this section, we introduce a new Proof of Data Reliability solution named PORTOS. We first present its main building blocks and further provide a complete description of the scheme.

5.2.1 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 4.2.1). 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 $\{\mathcal{S}^{(j)}\}_{1 \leq j \leq n}$.

Linearly–homomorphic tags. Similarly to POROS (c.f. Section 4.2.1), 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. [54] 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[1] that remain secret, and $\mathcal{T}$ is

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

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 $\beta^e \ (mod\ N)$.

5.2.2 Overview of PORTOS's Masking Mechanism

PORTOS leverages the cryptographic puzzle of Rivest et al. [54] 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:

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

- $\mathsf{maskGenFast}((i,j),\ (p',q'),\ \mathsf{param_m}) \rightarrow m_i^{(j)}$: 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:

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

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 $\mathsf{maskGen}$ 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.

5.2.3 Protocol specification

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.

- $\mathsf{Setup}\ (1^\lambda, t) \rightarrow (\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}, \mathsf{param_{system}})$: 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ and the system parameters $\mathsf{param_{system}} := (k, n, \mathbf{G}, q)$.

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

  Algorithm $\mathsf{U.Store}$ 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 $\mathrm{k}_{\mathsf{prf}} \in \{0,1\}^\lambda$, and a non-zero element $\alpha \xleftarrow{R} \mathbb{Z}_q$. Hereafter, $\mathsf{U.Store}$ computes for each data symbol a *linearly homomorphic* MAC as follows:

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

  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 $\mathsf{T_{thr}}$. Thereafter, algorithm $\mathsf{U.Store}$ picks a pseudo-random function $\mathsf{PRF_{mask}} : \mathbb{Z}_N \times$

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
$\alpha, \mathrm{k}_{\mathsf{prf}}$	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 5.1: Notation used in the description of PORTOS.

$\{0,1\}^* \to \mathbb{Z}_N$ (such that its output is guaranteed to have a large order modulo N) together with a random seed $\eta_{\mathsf{m}} \xleftarrow{R} \mathbb{Z}_N$, and constructs the functions $\mathsf{maskGen}$ and $\mathsf{maskGenFast}$ as described in Section 5.2.1

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

Finally, algorithm $\mathsf{U.Store}$ picks a pseudo-random generator $\mathsf{PRG}_{\mathsf{chal}} : \{0,1\}^\lambda \to [1,s]^l$ [13] and a unique identifier fid.

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

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

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

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

and the data object parameters:

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

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

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

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

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

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

$$\{\tilde{d}_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}} \leftarrow \{d_i^{(j)} + m_i^{(j)}\}_{\substack{1\le j\le k\\ 1\le i\le s}}, \tag{5.1}$$

$$\{\tilde{r}_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}} \leftarrow \{r_i^{(j)} + m_i^{(j)}\}_{\substack{k+1\le j\le n\\ 1\le i\le s}}. \tag{5.2}$$

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

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

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

- $\mathsf{U.Chall}\ (\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{chal})$: Provided with the object identifier fid, the secret key K_{u}, and the system parameters $\mathsf{param}_{\mathsf{system}}$, algorithm $\mathsf{U.Chall}$ 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 $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ the challenge

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

- $\mathsf{C.Prove}\ (\mathsf{chal}, \tilde{\mathcal{D}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{proof})$: 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

$$\tilde{\mu}^{(j)} \ \leftarrow\ \begin{cases} \sum_{c=1}^{l} \nu_c\ \tilde{d}^{(j)}_{i_c^{(j)}}, & \text{if}\ \ 1 \le j \le k \\ \sum_{c=1}^{l} \nu_c\ \tilde{r}^{(j)}_{i_c^{(j)}}, & \text{if}\ \ k+1 \le j \le n, \end{cases} \tag{5.3}$$

$$\tau^{(j)} \ \leftarrow\ \begin{cases} \sum_{c=1}^{l} \nu_c\ \sigma^{(j)}_{i_c^{(j)}}, & \text{if}\ \ 1 \le j \le k \\ \sum_{c=1}^{l} \nu_c\ \psi^{(j)}_{i_c^{(j)}}, & \text{if}\ \ k+1 \le j \le n. \end{cases} \tag{5.4}$$

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

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

- $\mathsf{U.Verify}\ (K_{\mathsf{u}}, \mathsf{chal}, \mathsf{proof}, \mathsf{maskGenFast}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{dec})$: 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 $\mathsf{param}_{\mathcal{D}} := (\mathsf{PRG}_{\mathsf{chal}},$

$\mathsf{maskGen}, \mathsf{param_m}$), this algorithm first checks if the response time of all storage nodes $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ is shorter than the time threshold $\mathsf{T_{thr}}$. If not algorithm U.Verify terminates by outputting dec := 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

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

and it generates the corresponding masking coefficients

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

Subsequently, it computes

$$\tilde{\tau}^{(j)} := \tau^{(j)} + \alpha \cdot \sum_{c=1}^{l} \nu_c \; m_{i_c^{(j)}}^{(j)}, \tag{5.5}$$

and then it verifies that the following equations hold

$$\tilde{\tau}^{(j)} \stackrel{?}{=} \begin{cases} \alpha\tilde{\mu}^{(j)} + \sum_{c=1}^{l} \nu_c \; \mathsf{PRF}(\mathrm{k_{prf}}, i_c^{(j)} \| \; j) & \text{if } 1 \leq j \leq k, \\ \alpha\tilde{\mu}^{(j)} + \sum_{c=1}^{l} \nu_c \; \mathsf{prf}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)} & \text{if } k+1 \leq j \leq n, \end{cases} \tag{5.6}$$

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

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

- C.Repair $({}^*\tilde{\mathcal{D}}, J_\mathsf{f}, \mathsf{param_{system}}, \mathsf{param_{\mathcal{D}}}, \mathsf{maskGen}) \rightarrow (\tilde{\mathcal{D}})$: 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$. In this case, algorithm C.Repair terminates outputting $\perp$; otherwise, it picks a set of k surviving storage nodes $\{\mathcal{S}^{(j)}\}_{j\in J_\mathsf{r}}$, where $J_\mathsf{r} \subseteq [1, n] \backslash J_\mathsf{f}$ and, computes the masking coefficients $\{m_i^{(j)}\}_{\substack{j\in J_\mathsf{r}\\ 1\leq i\leq s}}$ and $\{m_i^{(j)}\}_{\substack{j\in J_\mathsf{f}\\ 1\leq i\leq s}}$, 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 $\{\mathcal{S}^{(j)}\}_{j\in J_\mathsf{r}}$ and recon-

structs the original data object $\mathcal{D}$ using $\mathbf{H}$. Finally, algorithm $\mathsf{C.Repair}$ 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 $\{\mathcal{S}^{(j)}\}_{j\in J_\mathsf{f}}$.

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

5.3 Security Analysis

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

5.3.1 Correctness

We now show that the verification Equation 5.6 must hold if algorithm $\mathsf{C.Prove}$ is executed correctly. In particular, Equation 5.6 consists of two parts: the first one defines the verification of the proofs $\{(\tilde{\mu}^{(j)}, \tau^{(j)})\}_{1\le j\le k}$ generated by the data storage nodes $\{\mathcal{S}^{(j)}\}_{1\le j\le k}$; and the second part corresponds to the proofs $\{(\tilde{\mu}^{(j)}, \tau^{(j)})\}_{k+1\le j\le n}$ generated by the redundancy storage nodes $\{\mathcal{S}^{(j)}\}_{k+1\le j\le n}$. By definition the following equality holds:

$$\sigma_i^{(j)} = \alpha d_i^{(j)} + \mathsf{PRF}(\mathrm{k}_\mathsf{prf}, i \parallel j),\ \forall 1 \le i \le s, 1 \le j \le k \tag{5.7}$$

We begin with the first part of Equation 5.6. By plugging Equations 5.5 and 5.4 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 5.7 we get

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

Finally, by plugging Equations 5.1 and 5.3 to Equation 5.7 we get

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

As regards to the second part of Equation 5.6 that defines the verification of the proofs $\{(\tilde{\mu}^{(j)}, \tau^{(j)})\}_{k+1 \leq j \leq n}$ generated by the redundancy storage nodes $\{\mathcal{S}^{(j)}\}_{k+1 \leq j \leq n}$, we observe that for all $c \in [1, l]$ it holds that redundancy symbols $r^{(j)}_{i_c^{(j)}} = \boldsymbol{d}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)}$ and tags $\psi^{(j)}_{i_c^{(j)}} = \boldsymbol{\sigma}_{i_c^{(j)}} \cdot \mathbf{G}^{(j)}$, whereby $\mathbf{G}^{(j)}$ is the j^{th} column of generator matrix $\mathbf{G}$, $\boldsymbol{d}_{i_c^{(j)}} := (d^{(1)}_{i_c^{(j)}}, \ldots, d^{(k)}_{i_c^{(j)}})$ is the vector of data symbols for segment i_c, and $\boldsymbol{\sigma}_{i_c^{(j)}} := (\sigma^{(1)}_{i_c^{(j)}}, \ldots, \sigma^{(k)}_{i_c^{(j)}})$ is the corresponding vector of linearly homomorphic tags. Hence, by Equation 5.7 the following equality always holds:

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

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

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

and this proves correctness.

5.3.2 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 C.Prove and U.Verify can be seen as a distributed version of the algorithms SW.Prove and SW.Verify of the private PoR scheme in Section 2.4.1 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 of the Proof of Retrievability model (c.f. Section 2.2), 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

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

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 [10] 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-homomorphic tags (c.f. Section 2.4.1.1), 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

$$\tilde{\mu}^{(j)} \leftarrow \sum_{c=1}^{l} \nu_c \, \tilde{r}^{(j)}_{i_c^{(j)}}, \quad \text{for } k+1 \leq j \leq n, \tag{5.8}$$

$$\tau^{(j)} \leftarrow \sum_{c=1}^{l} \nu_c \, \psi^{(j)}_{i_c^{(j)}}, \quad \text{for } k+1 \leq j \leq n. \tag{5.9}$$

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 $\mathsf{T_{thr}}$ 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 $\mathsf{T_{resp}}$ and $\mathsf{T'_{resp}}$, we can determine the lower and upper bounds for the time threshold $\mathsf{T_{thr}}$.

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 $n-k+1$ 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 $\{m^{(j)}_{i_c^{(j)}}\}_{\substack{1\leq j\leq n\\ 1\leq c\leq l}}$.

(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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$, ensuring that it preserves at least k symbols for each data segment.

In order to determine the lower bound for the time threshold $\mathsf{T_{thr}}$ we evaluate the response time $\mathsf{T_{resp}}$ of an honest cloud storage provider C. Additionally, for each type of

Scenario	**Proof generation complexity for one $\mathcal{S}$**	**$\mathcal{S}$'s Response Time**
Honest C:	$2l$ mult $+ 2(l+1)$ add	$\mathsf{T}_{\text{resp}} = \lceil \frac{2l}{\Pi} \rceil \ \mathsf{T}_{\text{mult}} + \lceil \frac{2(l-1)}{\Pi} \rceil \ \mathsf{T}_{\text{add}} + \text{RTT}$
C' stores $\mathcal{D}$ unmasked:	$l\mathcal{T}$ exp $+2l$ mult $+ (3l-2)$ add	$\mathsf{T}'_{\text{resp}_1} = \lceil \frac{l}{\Pi} \rceil \mathsf{T}_{\text{puzzle}} + \lceil \frac{2l}{\Pi} \rceil \ \mathsf{T}_{\text{mult}} + \lceil \frac{3l-2}{\Pi} \rceil \mathsf{T}_{\text{add}} + \text{RTT}$
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	$\mathsf{T}'_{\text{resp}_2} = \mathsf{T}_{\text{puzzle}} + \lceil \frac{2l+1}{\Pi} \rceil \ \mathsf{T}_{\text{mult}} + \lceil \frac{2l-1}{\Pi} \rceil \ \mathsf{T}_{\text{add}} + \text{RTT}$

Table 5.2: 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. 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 $\mathsf{T}_{\text{puzzle}} := \mathcal{T} \cdot \mathsf{T}_{\text{exp}}$ is the time required by $\mathcal{S}$ to generate one masking coefficient.

C''s malicious behavior we evaluate its response time, and determine the upper bound for the time threshold T_{thr} as $\mathsf{T}'_{\text{resp}} = \min(\mathsf{T}'_{\text{resp}_1}, \mathsf{T}'_{\text{resp}_2})$, where $\mathsf{T}'_{\text{resp}_1}$ and $\mathsf{T}'_{\text{resp}_2}$ 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 $\mathsf{T}'_{\text{resp}_2}$, we consider the most favorable scenario for C' where it has to generate only one missing symbol for a PORTOS challenge. Table 5.2 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 $\{\mathcal{S}^{(j)}\}_{1 \leq j \leq n}$ have a bounded capacity of Π concurrent threads of execution, that computations – exponentiations, multiplications, additions, etc. – require a minimum execution time, and that $\mathsf{T}_{\text{add}} \ll \mathsf{T}_{\text{exp}}$ and $\mathsf{T}_{\text{add}} \ll \mathsf{T}_{\text{mult}}$. Furthermore, we assume that $\{\mathcal{S}^{(j)}\}_{1 \leq j \leq n}$ are connected with premium network connections (low latency and high bandwidth), and hence the communication among them has negligible impact on C' response time. As given in Table 5.2, *Req 3* is met as long as the time threshold T_{thr} is tuned such that it fulfills the following relations:

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

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

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

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 5.3: Performance analysis of PORTOS.

carefully setting the puzzle difficulty coefficient $\mathcal{T}$, we can guarantee that $\mathsf{T}_{\mathsf{resp}} \ll \mathsf{T}'_{\mathsf{resp}}$ and $\mathtt{RTT}_{\mathsf{max}} \ll \mathsf{T}'_{\mathsf{resp}} - \mathsf{T}_{\mathsf{resp}}$, 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.

5.4 Performance Analysis

Table 5.3 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 $\{\mathcal{S}^{(j)}\}_{1 \leq j \leq n}$) – 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 $\mathsf{proof} := \left\{(\tilde{\mu}^{(j)}, \tau^{(j)})\right\}_{1 \leq j \leq n}$ generated by cloud storage provider C is $2 \cdot n$ symbols: a pair of aggregated symbols and tags for each storage node $\{\mathcal{S}^{(j)}\}_{1 \leq j \leq n}$.

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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$. 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 $\{\mathcal{S}^{(j)}\}_{k+1\leq j\leq n}$ is much higher than the effort required to verify the response of a data node $\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}$. 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.

5.5 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 presented in Section 2.4.1. 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 3.2.4): the proof of reliability verification is done at the segment level and there is no redundancy among the segments of $\tilde{\mathcal{D}}$, 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 dec := 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 Section 2.2, 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$. 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq k}$ 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 $\{\mathcal{S}^{(j)}\}_{k+1\leq j\leq n}$.

5.5.1 Description

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

- $\mathsf{Setup}\ (1^\lambda, t) \rightarrow (\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}, \mathsf{param_{system}})$: 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$ 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 $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$, and the system parameters $\mathsf{param_{system}} := (k, n, \mathbf{G}, q)$.

- $\mathsf{U.Store}\ (1^\lambda, D, \mathsf{param_{system}}) \rightarrow (K_\mathsf{u}, \mathcal{D}, \mathsf{param}_\mathcal{D}, \mathsf{maskGenFast})$:

  On input security parameter λ, file $D \in \{0,1\}^*$, and system parameters $\mathsf{param_{system}}$, 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 $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{Z}_q$, together with its pseudo-randomly generated key $\mathrm{k}_\mathsf{prf} \in \{0,1\}^\lambda$, and k non-zero elements $\{\alpha^{(j)}\}_{1\leq j\leq k} \xleftarrow{R} \mathbb{Z}_q$. Hereafter, $\mathsf{U.Store}$ computes for each data symbol a *linearly homomorphic* MAC as follows:

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

  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 U.Store picks a pseudo-random generator $\mathsf{PRF}_{\mathsf{mask}} : \mathbb{Z}_N \times \{0,1\}^* \rightarrow \mathbb{Z}_N$[2] together with a random seed $\eta_{\mathsf{m}} \xleftarrow{R} \mathbb{Z}_N$, and constructs the functions maskGen and maskGenFast as described in Section 5.2.1

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

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

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

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

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

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

and the data object parameters

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

- 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 $\{r_i^{(j)}\}_{\substack{k+1\leq j\leq n\\ 1\leq i\leq s}}$ by multiplying each segment $\mathbf{d}_i := \big(d_i^{(1)}, d_i^{(2)}, \ldots, d_i^{(k)}\big)$ with the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

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

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

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

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

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

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

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 $\{\sigma_i\}_{1\le i\le s}$.

- $\mathsf{U.Chall}\ (\mathsf{fid}, K_{\mathsf{u}}, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{chal})$: 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 $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ the challenge

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

- $\mathsf{C.Prove}\ (\mathsf{chal}, \tilde{\mathcal{D}}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{proof})$: 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 $\{\mathcal{S}^{(j)}\}_{1\le j\le n}$ 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 C.Prove the linear combination of tags

$$\tau \leftarrow \sum_{c=1}^{l} \nu_c \, \sigma_{i_c}, \tag{5.10}$$

and terminates its execution by returning the proof:

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

- U.Verify $(K_{\mathsf{u}}, \mathsf{chal}, \mathsf{proof}, \mathsf{maskGenFast}, \mathsf{param}_{\mathcal{D}}) \rightarrow (\mathsf{dec})$: On input of user key $K_{\mathsf{u}} := (\{\alpha^{(j)}\}_{1\le j\le k},\ \mathrm{k}_{\mathsf{prf}},\ p',\ q')$, 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 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 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:

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

  and subsequently, it multiplies $\boldsymbol{\mu} := (\mu^{(1)}, \mu^{(2)}, \ldots, \mu^{(k)})$ with the generator matrix

$\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$:

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

Thereafter algorithm $\mathsf{U.Verify}$ verify the redundancy node responses are well formed as follows:

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

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

- $\mathsf{C.Repair}\ ({}^*\tilde{\mathcal{D}}, J_\mathsf{f}, \mathsf{param}_\mathsf{system}, \mathsf{param}_\mathcal{D}, \mathsf{maskGen}) \rightarrow (\tilde{\mathcal{D}})$: 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 $\{\mathcal{S}^{(j)}\}_{1 \leq j \leq n}$. 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 $\{m_i^{(j)}\}_{\substack{j \in J_\mathsf{r} \\ 1 \leq i \leq s}}$ and $\{m_i^{(j)}\}_{\substack{j \in J_\mathsf{f} \\ 1 \leq i \leq s}}$, using the function $\mathsf{maskGen}$, together with the parity check matrix $\mathbf{H} = [-\mathbf{P}^\top \mid \mathbf{I}_{n-k}]$.

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

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

5.5.2 Performance Analysis

Table 5.4 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 $\mathsf{U.Store}$. 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 3.2.4) as long as the time threshold $\mathsf{T_{thr}}$ is tuned such that it fulfills the following relation:

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

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

Table 5.4: Performance analysis of PORTOS with storage efficient tags.

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

5.6 Summary

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

Part II

Verifiable Storage with Data Reduction

Chapter 6

Verifiable Storage with Secure Deduplication

In this chapter, we address the conflict between Proofs of Retrievability and deduplication. More precisely, inspired by previous attempts in solving the problem of duplicating encrypted data, we propose a straightforward solution in combining PoR and deduplication. In addition we propose a novel massage-locked key generation protocol which is more resilient against off-line dictionary attacks compared to existing solutions.

6.1 Introduction

While existing Proof of Retrievability (PoR) schemes mainly look for means to optimize the performance at the user side, they usually assume that cloud storage providers have potentially infinite resources to store data and compute proofs of retrievability. Such an assumption unfortunately becomes too strong with the explosion of digital content. Cloud storage providers, nowadays, look for different data reduction techniques including data deduplication [55] to optimize their storage capacity: by eliminating duplicate copies of data, cloud servers achieve high storage space savings [56]. Unfortunately, current PoR solutions are incompatible with data deduplication as the integrity values resulting from the encoding algorithm are generated using a secret key that is only known to the owner of the file, and thus unique. Therefore, the encoding of a given file by two different users results in two different outputs.

In this chapter, we present a generic approach that solves the conflict between PoR and deduplication (c.f. Section 2.5.2) paving the way for a simple integration of PoR and deduplication within the same cloud storage system. The root cause of the conflict is the difference in the way duplicate data segments submitted by different users are handled

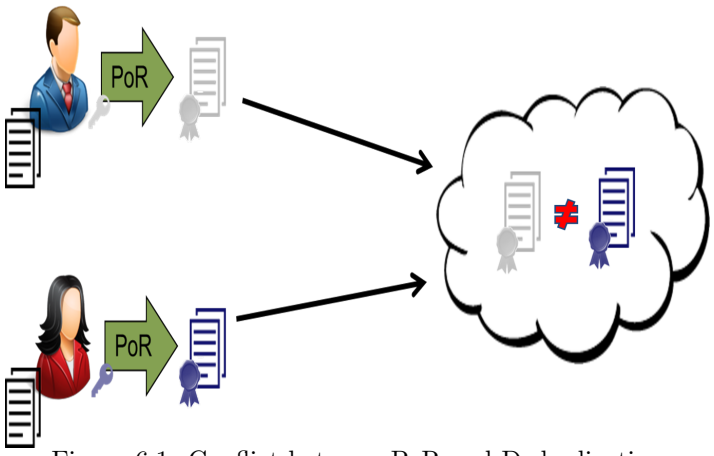

Figure 6.1: Conflict between PoR and Deduplication

by PoR and deduplication, namely, the fact that deduplication summarizes all duplicates into a single copy whereas PoR requires that every duplicate segment be kept separately in order to preserve the user-specific effect of PoR on each duplicate. Consequently, a new approach aiming at achieving identical PoR effects on duplicates submitted by different users seems to be the right solution for a simple composition of PoR with deduplication. Further along the same direction, the new approach should assure that PoR's effect on each data segment depends on the value of the data segment regardless of the difference in the identity of the users submitting the data segments, in other words, these PoR operations should be a function of the data segment's value independently of the user's identity. We thus define such PoR schemes that would be compatible with deduplication as *message-locked proofs of retrievability*.

A straightforward technique to implement *message-locked* PoR (ML.PoR) consists of defining the PoR operation on the data segment as a one-way function of the data segment. Without loss of generality existing PoR schemes can be represented by $\mathcal{P}(d, K)$ as a function $\mathcal{P}$ of the data segment d and a key K that is determined by each user performing the PoR processing on d. In the case of tag-based PoR schemes K is the secret key used to produce the tags, whereas in PoR schemes relying on watchdogs K is the secret key used to generate the watchdogs and encrypt the data. Given a one-way hash function f, existing PoR schemes can be transformed to become *message-locked* by simply substituting K with $f(d)$ in the existing PoR function $\mathcal{P}(d, K)$. Based on this technique any existing symmetric PoR scheme can be transformed into a *message-locked* PoR and as such be smoothly integrated with a deduplication function. Yet a new concern arises about the substitution of the secret key K with the result of a one-way hash function of the data segment. A similar question has been tackled when dealing with the problem of deduplication of

encrypted data and several solutions ranging from convergent encryption [55] to schemes exploiting the popularity of data segments [57, 58] through server-based protocols [59, 60] have been suggested. Like *message-locked* PoR, these solutions also rely on a key derived from the data segment using some one-way function often called a *message-locked* key. Unfortunately, most *message-locked* key generation techniques suffer from various security exposures and the design of a secure *message-locked* key generation technique is subject to several additional constraints. The *message-locked* PoR transformation we suggest can be based on any secure *message-locked* key generation technique.

Organization. The rest of this chapter is organized as follows. Section 6.2 provides some background on secure deduplication and reviews related work on message-locked key generation protocols, and Proofs of Storage with deduplication. The idea of the underlying solution is presented in Section 6.3. Section 6.4 describes the newly proposed server-aided message-locked key generation technique. Section 6.5 presents the entire solution and specifies two instantiations building upon two different PoR schemes. The security and performance of the solution are analyzed in Sections 6.6 and 6.7, respectively.

6.2 Background

In this section, we present the problem of *secure deduplication* that is the source of inspiration for our *message-locked PoR* solution. Thereafter, we review previous work on message-locked key generation protocols that were tailored for secure deduplication and the state of the art on Proofs of Storage with deduplication.

6.2.1 Secure Deduplication

The term *secure deduplication* describes the techniques that enable a cloud storage system to meet the inherently conflicting requirements of *data confidentiality* and *data deduplication*, i.e., securely and efficiently combining encryption with storage deduplication. The root cause of the conflict is that encryption of a same data segment performed by different users ensues completely different results for the same original copies.

Single-user vs cross-user deduplication. In addition to the differentiation according to the granularity level of deduplication (*file-level* or *block-level* deduplication) as well as, where deduplication occurs (*server-side* or *client-side* deduplication) that we presented in Section 1.1, secure deduplication methods can also be classified according to their scope:

- *Single-user deduplication*, where the cloud storage system performs deduplication only on the data of a single user. In this case, encryption can be safely added without impeding deduplication, as long as the same key is used to encrypt all data segments.

- *Cross-user deduplication*, where the cloud storage system performs deduplication across the data of multiple users. This type of deduplication raises the issue of implementing a secure *message-locked* key generation mechanism that enables users to derive the same encryption key for a given data segment without compromising confidentiality.

Regarding the performance of the two methods, *cross-user deduplication* is clearly the most effective method: enabling deduplication across the data of all users in a cloud storage system significantly increases the probability of finding redundant data and as a result achieve grater gains in terms of storage space.

Client-side deduplication and data confidentiality. Client-side deduplication brings the benefit of reducing bandwidth consumption as only data that have not previously uploaded to the cloud storage system are transfered over the network. Unfortunately, this setting enables a malicious adversary to discover whether a particular piece of data is already stored in the cloud storage system [61,62]. In some contexts, this weakness may pose a serious confidentiality threat as the adversary can exploit that deduplication functionality in order to obtain confidential information. To circumvent such attacks, solutions in the literature [63, 64] propose to combine client-side deduplication mechanisms with proofs of ownership (PoW) [65] which help the cloud storage provider to verify that a user actually *owns* a file without the need to upload it.

6.2.2 State of the Art

Message-locked key generation. To show that deduplication and encryption can coexist, authors in [55] introduce the concept of convergent encryption whereby all existing copies of a file is encrypted with the same encryption key: the encryption key simply is the hash of the file. While this initial approach where the encryption key is derived from the file itself seems ideal to achieve storage efficiency and data confidentiality at the same time, it unfortunately suffers from several weaknesses including dictionary attacks during which a cloud storage provider C can try to guess the file. A curious C can compute the hash of potential candidates of a given file, derive an encryption key, and check whether the encryption of the file with this key is actually stored at its premises.

To thwart this type of attacks, [59] introduces a key server KS. The idea is that every time a user wants to upload a file it will engage in an instance of an oblivious pseudo-random function (OPRF) protocol [66] with KS to generate a secret key with which the file to be uploaded is going to be encrypted. In order to ensure that users owning the same file agree on the same key, the user's input to the OPRF will depend on the file. As a result, the proposed scheme puts an end to offline attacks and forces the cloud storage provider C (or a user) to contact KS whenever it makes a guess for an uploaded file. This allows KS to implement rate-limiting measures to restrict the number of queries C can issue, and as a by product limit the number of dictionary attacks C can conduct in a given period of time.

Building on the same idea, the authors of [64] propose ClearBox which is a transparent storage service that provides privacy-preserving deduplication while making sure that users are charged only for the storage they actually use. The main contributions of ClearBox are twofold: Replace OPRF with a BLS blind signature [67] to reduce the communication and the computation cost of the key generation protocol; and combine Merkle trees and time-dependent randomness (obtained by leveraging some bitcoin functionalities) to ensure that the amount of money users pay is proportional to the rate of deduplication of their uploaded files.

While the work of [59, 64] succeeds in circumventing offline dictionary attacks by C, they are both prone to offline dictionary attacks by KS. To address this issue, Liu et al. [68] devised a solution that allows users to agree on a secret key without connecting to a key server. The proposed solution relies on additively homomorphic encryption, password-authenticated key exchange (PAKE) and low-entropy hash to empower users uploading the same file to derive the same encryption key. Furthermore by using low-entropy hash, the scheme in [68] is able to have a rate-limiting strategy per file, which offers better security guarantees than [59, 64] that deploy rather a rate-limiting strategy per user.

Any of these solutions can be used as a building block to achieve message-locked PoR, nevertheless, we propose a different approach that does not rely on a single key server, but without the complexity of peer-to-peer systems. The idea is to have the user U interact with both cloud server C and key server KS to generate the message-locked secret key that will later be used as input to the message-locked PoR. The new solution will be described in Section 6.4.

Proofs of storage with deduplication. Chen et al. [69] present BL-MLE, a message-locked encryption scheme that supports both file-level and block-level secure data deduplication. The main contributions of this scheme are the small-size metadata that allow for fine-grained dual-level deduplication with minimal overhead and the use of a Proof of

Ownership (PoW) scheme in order to support for client-side deduplication. Additionally, the authors claim that BL-MLE can be extended to use the PoR scheme in [28] in order to support support secure deduplication with data integrity without providing integration details.

Shin et a;. [70] proposed a proof of storage with deduplication (POSD) whereby thanks to a publicly verifiable proof of data possession scheme, users can verify the correct storage of deduplicated data using the public key of the first user actually storing the data. This solution has been proved insecure in [71] as it does not prevent the cloud storage provider from cheating.

Armknecht et al. [72] introduce a multi-tenant PoR framework that marries well with deduplication. The proposed solution relies on shared aggregated tags based on BLS signatures [10] that incorporate the secret keys of all users. The scheme considers a security model in which users can be corrupted by a malicious cloud storage provider: obtaining the secret key from a user does not help the malicious cloud storage provider to reconstruct a deleted symbol tag. This however comes at a high cost at the user side in terms of bandwidth and computation: the verification complexity of the storage of a file grows linearly with the number of users outsourcing that file.

Leontiadis et al. [44] extend the Proof of Data Reliability scheme in [36] in order to propose a 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.

6.3 Message-Locked Proofs of Retrievability

We consider a cloud storage model that comprises a number of affiliated users who are interested in securely outsourcing their files to a cloud storage provider C. Moreover, these users wish to take advantage of the benefits of *cross-user file-level deduplication performed* by C (e.g., reduced storage costs) whilst still being assured of the integrity of their outsourced data. The latter goal is achieved through Proofs of Retrievability (PoR) which, as discussed earlier, offer a user U cryptographic guarantees on the correct storage of her outsourced data at the cost of a pre-processing (setup) phase during which algorithms KeyGen and Encode are executed to prepare U's files for upload. As we have discussed in Section 2.2, the Encode algorithm consists of combining erasure codes with cryptographic primitives such as encryption and tags or watchdogs to build a verifiable data object that is uploaded to C.

This entails that in order to allow cloud storage provider C to deduplicate the out-

sourced files, users outsourcing the same file D should provide algorithm Encode with the same input. Notably, the users should agree on the secret keys of the cryptographic functions. To that effect, we propose a new ML.KeyGen algorithm that allows a user U with some file D to generate a key K_D by communicating with a key server KS and cloud storage provider C such that K_D is generated using a one-way function of file D, and the secret keys of KS and C (c.f. Section 6.4). Thanks to ML.KeyGen, we can transform any PoR scheme into a message-locked PoR by applying minor changes to the Encode algorithm. This provides users with secure means to verify the integrity of their outsourced files while at the same time saving storage (via deduplication) at the cloud storage provider. As to what type of deduplication to use, ML-PoR goes with server-side deduplication. We recall that in server-side deduplication (c.f. Section 2.5), the users upload their files to cloud storage provider C, which in turn performs the deduplication.

Threat model. Since the solution uses a server aided key generation solution for deduplication, similarly to all previously proposed server-aided encryption solutions, it assumes that the cloud storage provider C does not collude with the key server KS. Furthermore, cloud users are assumed not to collude either with the cloud server or with the key server. Therefore, the only information the cloud storage provider C can have access to is users' verifiable data objects and the messages exchanged during the message-locked key generation protocol.

6.4 ML.KeyGen: Server-aided message-locked key generation

In this section, we describe a new server-aided message-locked key generation protocol, ML.KeyGen, that will be used by ML.PoR to generate the keying material for PoR. Compared to related work (c.f. Section 6.2), this new solution offers better security guarantees, as it protects against dictionary attacks that could be launched by the key server as well. ML.KeyGen extends the solution in [64] by generating the *message-locked* key using two secret keys, each of them generated by the key server KS and cloud storage provider C respectively. Thanks to this solution, neither KS nor C can solely mount dictionary attacks.

6.4.1 Building Blocks

ML.KeyGen relies on the following building blocks.

Bilinear pairings. Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be three cyclic groups of prime order p. A bilinear pairing [73, 74] is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ with the following properties:

- e is *bilinear*: $e(g_1^{\alpha}, g_2^{\beta}) = e(g_1, g_2)^{\alpha\beta}$, for all $g_1 \in \mathbb{G}_1$, $g_1 \in \mathbb{G}_2$, and $\alpha, \beta \in \mathbb{Z}_p$.
- e is *computable*: there exists an *efficient* algorithm that computes $e(g_1, g_2)$ for any $(g_1, g_2) \in \mathbb{G}_1 \times \mathbb{G}_2$.
- e is *non-degenerate*: if g_1, g_2 are generators of $\mathbb{G}_1$ ans $\mathbb{G}_2$ respectively, then $e(g_1, g_2)$ is a generator of $\mathbb{G}_T$.

Bilinear pairings are classified in three types with respect to the characteristics of the underlying groups.

- If $\mathbb{G}_1 = \mathbb{G}_2$, then the pairing is symmetric or of *Type 1*.
- If $\mathbb{G}_1 \neq \mathbb{G}_2$ and there exists an *efficiently* computable map $\phi : \mathbb{G}_2 \to \mathbb{G}_1$, then the pairing is of *Type 2*.
- If $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is no *efficiently* computable map between $\mathbb{G}_1$ and $\mathbb{G}_1$, then the pairing is of *Type 3*.

BLS signatures. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a *type 2* bilinear pairing and g_1, g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Moreover, let $H^* : \{0,1\}^* \to \mathbb{G}_1$ a cryptographic hash function. The the BLS signature scheme [67] is defined as follows:

- $\mathsf{KeyGen}\ (1^{\lambda}) \to (sk, pk)$: Algorithm KeyGen picks a random number $\alpha \in \mathbb{Z}_p$, and sets $sk := \alpha$ and $pk := g_2^{\alpha} \in \mathbb{G}_2$.
- $\mathsf{Sign}\ (sk, m) \to (\sigma)$: Algorithm Sign outputs the signature $\sigma := H^*(m)^{\alpha} \in \mathbb{G}_1$.
- $\mathsf{Verify}\ (pk, m, \sigma) \to (\mathsf{dec} \in \{\mathsf{accept}, \mathsf{reject}\})$: If $e(g_2, \sigma) = e(pk, H^*(m))$ algorithm Verify outputs $\mathsf{dec} := \mathsf{accept}$, otherwise it outputs $\mathsf{dec} := \mathsf{reject}$.

6.4.2 Description of ML.KeyGen

The proposed protocol is executed among a user U, the key server KS and the cloud storage provider C. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two groups of prime order p with g_1 and g_2 as their respective generators, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing. We also define two cryptographic hash functions $H^* : \{0,1\}^* \to \mathbb{G}_1$ and $H : \mathbb{G}_1 \to \{0,1\}^{\lambda}$ with λ being a security parameter. During a setup phase, KS and C, respectively choose a private key $\kappa \in \mathbb{Z}_p^*$ and $\gamma \in \mathbb{Z}_p^*$ and publish their corresponding public keys $(y_{\mathsf{KS},1} = g_1^{\kappa}, y_{\mathsf{KS},2} = g_2^{\kappa})$ and $(y_{\mathsf{CS},1} = g_1^{\gamma}, y_{\mathsf{CS},2} = g_2^{\gamma})$. As depicted in Figure 6.2:

- User U computes an initial *message-locked* key h for file D by simply computing the hash of D: $h \leftarrow H^*(D)$.

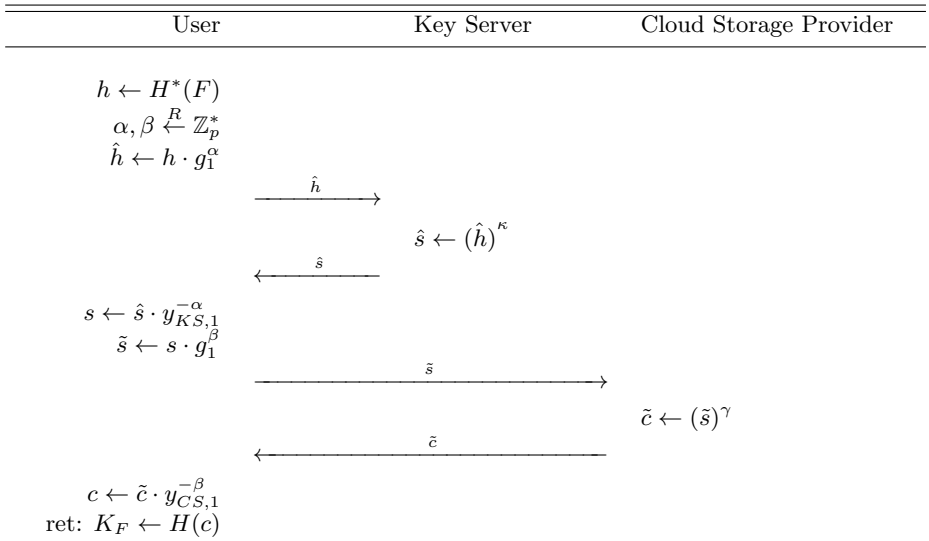

Figure 6.2: ML.KeyGen- Protocol Description

- This key is further blinded using a pseudo randomly generated value $\alpha \in \mathbb{Z}_p^*$ and the result denoted by $\hat{h}$ is sent to key server KS: $\hat{h} \leftarrow h \cdot g_1^{\alpha}$
- Thanks to the underlying blinded signature, key server KS computes the signature of $\hat{h}$ using its private key κ: $\hat{s} \leftarrow (\hat{h})^{\kappa}$.
- Upon reception of this blinded signature, user U unblinds this value to derive the signature of h. Hence: $s = h^{\kappa} g_1^{\alpha\kappa} g_1^{-\alpha\kappa}$. Subsequently, s is verified by checking if $e(s, g_2) = e(h, y_{\mathsf{KS},2})$.
- User U blinds s using a second random value $\beta \in \mathbb{Z}_p^*$ and sends the result $\tilde{s}$, this time, to cloud storage provider C: $\tilde{s} \leftarrow s \cdot g_1^{\beta}$.
- Similarly to key serverKS, cloud storage provider C signs $\tilde{s}$ and returns this signature to user U: $\tilde{c} \leftarrow (\tilde{s})^{\gamma}$.
- User U in turn, unblinds $\tilde{c}$ to derive the signature $c = h^{\kappa\gamma} g_1^{\beta\gamma} g_1^{-\beta\gamma}$ and verifies the signature using $y_{CS,2}$.
- If the verification succeeds, the *message-locked* key K_D equals $H(c) = H(h^{\kappa\gamma})$.

Thanks to this new key generation solution neither party can perform offline dictionary attacks.

6.4.3 Security analysis of ML.KeyGen

The work of [59, 64] on secure deduplication relies on the assumption that the key server KS is not interested in learning the content of the files kept at the cloud storage provider C; and thus the key generation protocol only involves the user – having as input the file to be uploaded – and KS – having as input its secret key. Yet, it is easy to see that there is no real countermeasure to deter key server KS from running offline dictionary attacks to compromise the confidentiality of the uploaded files. Our solution addresses this problem by having cloud storage provider C take part in ML.KeyGen forcing as a result KS to go online and connect to C whenever it makes a guess for an outsourced file. Luckily, such online attacks can be obstructed using rate-limiting measures that bound the number of ML.KeyGen runs (and therewith the number of online attacks) that a given user can initiate within a given time period. Hence, as long as cloud storage provider C and key server KS do not collude, none of them can perform offline dictionary attacks.

6.5 ML.PoR: Protocol Description

ML.Encode: Message-Locked PoR Encoding. Generally speaking, the preprocessing step of a Proof of Retrievability scheme consists of the following operations (c.f. Section 2.2):

(i) applying an erasure code to the file to be uploaded so as to allow user U to recover her file in the face of accidental errors;

(ii) encrypting and permuting the file to hide the dependencies between data blocks and redundancy symbols;

(iii) incorporating some integrity values (tags or watchdogs) which are later used to verify the integrity of the outsourced file.

It follows that for message-locked PoR to work, users should not only generate the same secret for the same file, but also agree on the the erasure code parameters, the encryption and the permutation algorithms and the integrity mechanisms. Accordingly in our scheme, we let the key server KS choose and advertise these parameters and algorithms before any user U joins the system. In this manner, we ensure that the encoding operation yields the same output for a fixed input file regardless of the user carrying it out.

To summarize, ML.Encode runs in the same way as a regular Encode, except for the following:

(i) The secret key K_D is generated using algorithm ML.KeyGen; and

(ii) all the parameters related to PoR (i.e. erasure code algorithm and cryptographic functions) are provided by key server KS.

To illustrate the feasibility of *message-locked* PoRs, we describe in what follows two instantiations that build upon the Privet Compact PoR and StealthGuard (c.f. Section 2.4). Before the detailed description of our instantiations, we note that like a regular PoR a message locked PoR comprises five algorithms: ML.KeyGen, ML.Encode, ML.Chall, ML.Prove and ML.Verify. We note that algorithms ML.Chall, ML.Prove and ML.Verify are executed in the same way as their counterparts in the original protocols.

6.5.1 ML.Private-Compact-PoR: A message-locked PoR scheme based on linearly-homomorphic tags

This section describes a *message-locked* PoR that extends an existing tag-based solution named Private Compact PoR (c.f Section 2.4.1). Tag-based PoR schemes need to be adapted in order to be compatible with secure deduplication.

Overview of Private Compact PoR. A user U wishing to outsource a file D proceeds as follows:

- U calls algorithm KeyGen in order to generate a secret key K that will be used to prepare the verifiable data object $\mathcal{D}$ for upload and to verify its retrievability later.
- U then calls algorithm Encode, which is in charge of preparing data object $\mathcal{D}$. Accordingly, algorithm Encode applies an erasure code to D and then uses the secret key K to encrypt, shuffle and authenticate D. The authentication consists of generating a linearly-homomorphic tag for each segment within the $\mathcal{D}$.

At any time user U wishes to check the retrievability of file $\mathcal{D}$, she calls algorithm Chall to generate a query chal that includes the indices of randomly-chosen segments together with some randomly-generated coefficients. On receiving the challenge chal, cloud storage provider C calls algorithm Prove to generate a proof of retrievability proof. Such a proof consists of a linear combination of the randomly-chosen segments using the randomly-generated coefficients and the corresponding tags. The latter is computed by applying the same linear combination over each segment's linearly-homomorphic tag.

User U decides that D is retrievable if the proof produced by cloud storage provider C is correctly formed: She uses her secret key K to check if the tag in the proof successfully authenticates the linear combination of the file splits.

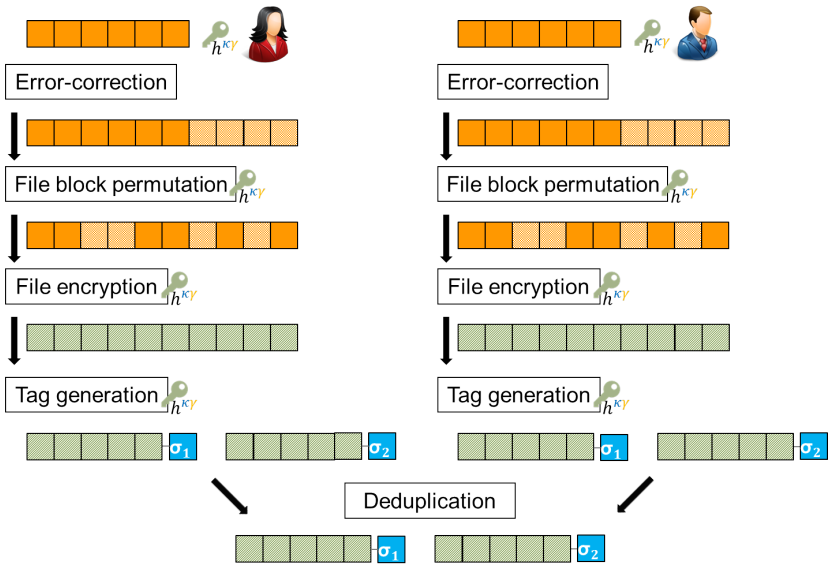

Figure 6.3: Steps of the ML.Encode algorithm of ML.Private-Compact-PoR.

ML.Private-Compact-PoR. In order for Private Compact PoR to become compatible with deduplication, it needs to be slightly modified such that all users who outsource the same file D to cloud storage provider C compute *the same* verifiable data object $\mathcal{D}$. In particular, we make Private Compact PoR *message-locked* by leveraging the newly proposed algorithm ML.KeyGen. Additionally, key server KS publishes some parameters which should be common for all users in the system in order for algorithm ML.Encode to be deterministic and return the same output for each of its execution over the same file D.

Assume that user U intends to outsource a file D. Accordingly, U prepares data object $\mathcal{D}$ for upload as follows:

- ML.KeyGen $(1^\lambda, D) \rightarrow (K_D, \mathsf{param}_{\mathsf{system}})$: User U calls this algorithm in order to generate a secret key K_D and a set of public parameters $\mathsf{param}_{\mathsf{system}}$ that will be used to prepare D for upload and to verify its retrievability later. These public parameters are listed in Table 6.1.

- ML.Encode $(K_D, D, \mathsf{param}_{\mathsf{system}}) \rightarrow (\mathsf{fid}, \mathcal{D})$: Given secret key K_D and the public parameters $\mathsf{param}_{\mathsf{system}}$ advertised by key sever KS, User U calls algorithm ML.Encode that performs the following operations:

  1. *Erasure coding*: Algorithm ML.Encode applies the erasure code published by key server KS to file D. This yields file $\dot{D}$.

Public Parameter	Description
λ	security parameter of Compact PoR
b	size of a symbol in bits
m	number of symbols in a segment D_i
ρ	code rate of erasure code, $\rho = \frac{m}{m+d}$
$[m, k, d]$–Erasure code	Erasure code correcting up to $\frac{d}{2}$ errors per segment
$\mathsf{Enc} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$	encryption algorithm
$\mathsf{PRP}_D : \{0,1\}^\lambda \times [\![1, n \cdot D]\!] \to [\![1, n \cdot D]\!]$	pseudo-random file-level permutation
$\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \to \mathbb{Z}_p$	pseudo-random function
$H_{\mathsf{perm}} : \{0,1\}^* \to \{0,1\}^\lambda$	file-level permutation key generator
$H_{\mathsf{enc}} : \{0,1\}^* \to \{0,1\}^\lambda$	encryption key generator
$H_{\mathsf{prf}} : \{0,1\}^* \to \{0,1\}^\lambda$	PRF key generator
$H_\alpha : \{0,1\}^* \times [\![1, m]\!] \to \mathbb{Z}_p$	tag coefficient generator

Table 6.1: ML.Private-Compact-PoR's Public Parameters.

2. *File block permutation*: At this step, ML.Encode computes a permutation key $K_{\mathsf{perm}} = H_{\mathsf{perm}}(K_D)$ which together with the published pseudo-random permutation PRP_D is used to permute all the blocks in file $\dot{D}$. Without loss of generality, we denote the resulting permuted file $\ddot{D}$.
3. *File encryption*: Having K_D, ML.Encode derives an encryption key $K_{\mathsf{enc}} = H_{\mathsf{enc}}(K_D)$, and uses this encryption key and the semantically secure encryption algorithm Enc published by KS to encrypt the symbols in file $\ddot{D}$. Let $\hat{D}$ denote the encrypted file.
4. *Tag generation*: $\hat{D}$ is further divided into n equally-sized segments each comprising m symbol. We denote $\hat{d}_{ij}$ the j^{th} symbol of the i^{th} segment where $1 \leq i \leq n$ and $1 \leq j \leq m$. ML.Encode then generates a PRF key $K_{\mathsf{prf}} = H_{\mathsf{prf}}(K_D)$ and generates m random numbers $\alpha_j = H_\alpha(K_D, j)$ where $1 \leq j \leq m$. Then, for each split $\hat{D}_i$, ML.Encode computes the following linearly-homomorphic tag σ_i:

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

Algorithm ML.Encode then picks a unique identifier fid, and terminates its execution by outsourcing to the cloud storage provider C the authenticated data object:

$$\mathcal{D} := \big\{\mathsf{fid};\ \ \{\hat{d}_i j\}_{\substack{1 \leq j \leq m \\ 1 \leq i \leq n}};\ \ \{\sigma_i\}_{1 \leq i \leq n}\big\}.$$

- ML.Chall $(K_D, \mathsf{fid}) \to (\mathsf{chal})$: This algorithm invoked by the user U picks l random elements $\nu_c \in \mathbb{Z}_q$ and l random symbol indices i_c, and sends to the cloud storage

provider C the challenge

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

- $\mathsf{ML.Prove}$ $(\mathsf{fid}, \mathsf{chal}) \rightarrow (\mathsf{proof})$: Upon receiving the challenge chal, C invokes this algorithm which computes the proof $\mathsf{proof} := (\mu_j, \tau)$ as follows:

$$\mu_j := \sum_{(i_c, \nu_c) \in \mathsf{chal}} \nu_c \, d_{ij} \ , \quad \tau := \sum_{(i_c, \nu_c) \in \mathsf{chal}} \nu_c \, \sigma_{i_c}.$$

- $\mathsf{ML.Verify}$ $(K_D, \mathsf{proof}, \mathsf{chal}) \rightarrow (\mathsf{dec})$: This algorithm invoked by the user U, verifies that the following equation holds:

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

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

Notice that if there is another user U' wishing to outsource the same file D, cloud storage provider C will easily detect the duplicate copy if she executes the proposed ML.Private-Compact-PoR and consequently remove it to save storage space. Thanks to the deterministic nature of the underlying algorithms, U' will still be able to check the retrievability of $\mathcal{D}$.

6.5.2 ML.StealthGuard: A message-locked PoR scheme based on watchdogs

This section describes a *message-locked* PoR that extends an existing watchdog-based solution named StealthGuard (c.f Section 2.4.2). In StealthGuard, data retrievability is achieved thanks to the oblivious insertion of pseudo-randomly generated symbols named *watchdogs*. Furthermore, StealthGuard leverages a privacy-preserving word search scheme in order to query the watchdogs without leaking any information about their value or their position within the data.

Overview of StealthGuard. A user U wishing to outsource her file D proceeds as follows:

- U calls the algorithm KeyGen to derive a secret key K that is used to process D before

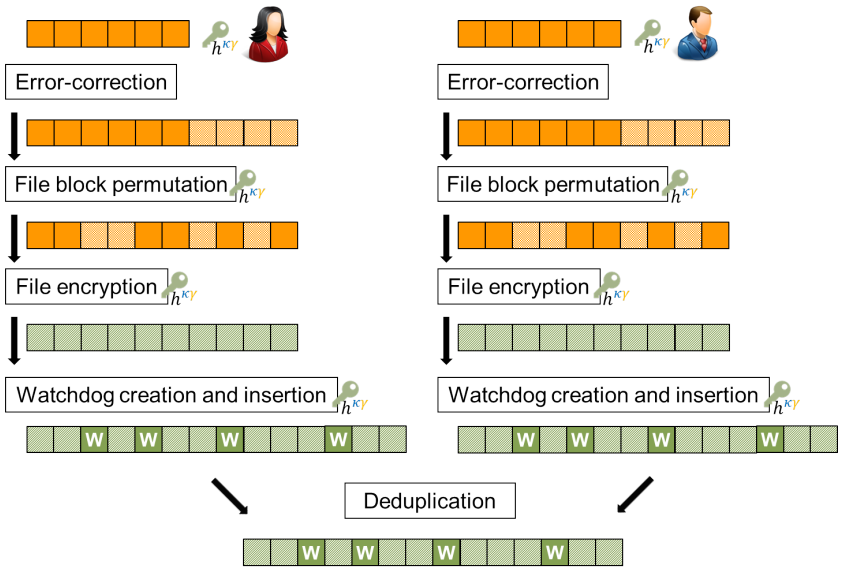

Figure 6.4: Steps of the ML.Encode algorithm of ML.StealthGuard.

outsourcing that data object $\mathcal{D}$ to cloud storage provider C as well as to verify its retrievability later.

- U then calls algorithm Encode, which is the algorithm that produces data object $\mathcal{D}$. Hence, algorithm Encode applies an erasure code to D, and then using secret key K determines the value and location of the watchdogs within the encoded file and permutes and encrypts the file.

Once $\mathcal{D}$ is uploaded, user U can indefinitely query randomly chosen watchdogs thanks to the underlying privacy preserving search mechanism without leaking any information about the actual watchdog and its position.

ML.StealthGuard. Similarly to Private Compact PoR, the original StealthGuerd becomes compatible with deduplication whenever it uses a *message-locked* key together with some other parameters that are common for all users in the system. Therefore ML.StealthGuard builds upon the newly proposed ML.KeyGen and assumes that all users fetch the public parameters from a key server KS.

Assume that user U intends to outsource a file D. Accordingly, U prepares data object $\mathcal{D}$ for upload as follows:

- ML.KeyGen $(1^\lambda, D) \rightarrow (K_D, \mathsf{param}_{\mathsf{system}})$: User U calls this algorithm in order to generate a secret key K_D and a set of public parameters $\mathsf{param}_{\mathsf{system}}$ that will be

Public Parameter	Description
λ	security parameter of StealthGuard
b	size of a symbol in bits
m	number of symbols in a segment D_i
v	number of watchdogs in one segment
ρ	code rate of erasure code, $\rho = \frac{m}{m+d}$
$[m, k, d]$–Erasure code	Erasure code correcting up to $\frac{d}{2}$ errors per segment
$\mathsf{Enc} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$	encryption algorithm
$\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^l$	watchdog generator
$\mathsf{PRP}_D : \{0,1\}^\lambda \times [\![1, n \cdot D]\!] \to [\![1, n \cdot D]\!]$	pseudo-random file-level permutation
$\mathsf{PRP}_{D_i} : \{0,1\}^\lambda \times [\![1, D + v]\!] \to [\![1, D + v]\!]$	pseudo-random segment-level permutation
$H_{\mathsf{permD}} : \{0,1\}^* \to \{0,1\}^\lambda$	file-level permutation key generator
$H_{\mathsf{enc}} : \{0,1\}^* \to \{0,1\}^\lambda$	encryption key generator
$H_{\mathsf{wdog}} : \{0,1\}^* \to \{0,1\}^\lambda$	watchdog key generator
$H_{\mathsf{permD_i}} : \{0,1\}^* \times [\![1, n]\!] \to \{0,1\}^\tau$	segment-level permutation key generator

Table 6.2: ML.StealthGuard's Public Parameters

used to prepare D for upload and to verify its retrievability later. These public parameters are listed in Table 6.2.

- $\mathsf{ML.Encode}\ (K_D, D, \mathsf{param_{system}}) \to (\mathsf{fid}, \mathcal{D})$: Given secret key K_D and the public parameters $\mathsf{param_{system}}$ advertised by key sever KS, User U calls algorithm ML.Encode that performs the following operations:

  1. *Erasure coding*: As a first step, ML.Encode divides file D into n segments $\{D_1, D_2, ..., D_n\}$ where each segment D_i with $1 \leq i \leq n$, regroups m blocks of size l. ML.Encode further applies the erasure code published by KS to each file segment D_i. This yields a new encoded file $\dot{D}$.
  2. *File symbol permutation*: At this step, ML.Encode first generates a permutation key K_{permF} such that $K_{\mathsf{permD}} = H_{\mathsf{permF}}(K_D)$. This key and the published pseudo-random permutation PRP_D are used to shuffle all the blocks in file $\dot{D}$. Let $\ddot{D}$ denote the resulting file.
  3. *File encryption*: Given secret key K_D, ML.Encode derives an encryption key $K_{\mathsf{enc}} = H_{\mathsf{enc}}(K_D)$, and with this encryption key ML.Encode further encrypts the symbols in file $\ddot{D}$ using the semantically secure encryption algorithm Enc published by KS. We denote by $\tilde{D}$ file $\ddot{D}$ after encryption.
  4. *Watchdog insertion*: ML.Encode computes a watchdog generation key $K_{\mathsf{wdog}} = H_{\mathsf{wdog}}(K_D)$ and uses this key and the published pseudo-random function PRF to generate $n \times v$ watchdogs $w_{ij} = \mathsf{PRF}(K_{\mathsf{wdog}}, i, j)$ for $1 \leq i \leq n$ and $1 \leq j \leq v$. Since the watchdogs are pseudo-randomly generated and the symbols in

the splits are encrypted using a semantically secure encryption, cloud storage provider C cannot differentiate between watchdogs and data symbols. Once all watchdogs are generated, ML.Encode appends v watchdogs $\{w_{i1}, ..., w_{iv}\}$ to each $\tilde{D}_i$in $\tilde{D}$. Each split is permuted using their corresponding newly generated permutation key $K_{\mathsf{permS,i}} = H_{\mathsf{permD_i}}(K_D, i)$ and the published pseudo-random permutation PRP_{D_i}. Without loss of generality, we denote $\mathcal{D}$ file $\tilde{D}$ after the insertion of watchdogs.

Algorithm ML.Encode then picks a unique identifier fid, and terminates its execution and further outsources to the cloud storage provider C the verifiable data object $\mathcal{D}$.

- ML.Chall $(K_D, \mathsf{fid}) \rightarrow (\mathsf{chal})$: User U calls this algorithm which chooses a γ segments and a watchdog within each of these segments. Thereafter algorithm ML.Chall uses the underlying PRISM.Query algorithm to issue a privacy preserving search queries for each of the selected watchdogs. Algorithm ML.Chall terminates its execution by sending to the cloud storage provider C the challenge chal comprising the γ search queries.

- ML.Prove $(\mathsf{fid}, \mathsf{chal}) \rightarrow (\mathsf{proof})$: Upon receiving the challenge chal the cloud storage provider C invokes this algorithm which uses the underlying PRISM.Process algorithm to construct the response for each of the γ search queries. Thanks PRISM.Process, cloud storage provider C cannot not learn either the content of the search query or the corresponding response. Algorithm ML.Prove terminates its execution by sending to the user U the proof proof comprising the responses to γ search queries.

- ML.Verify $(K_D, \mathsf{fid}, \mathsf{chal}, \mathsf{proof}) \rightarrow (\mathsf{dec})$: User U calls this algorithm which uses the underlying PRISM.Analysis algorithm to processes all responses included to the proof proof. Algorithm ML.Verify outputs dec := accept if the all queried watchdogs are present or dec := reject otherwise.

Furthermore, another user U′ wishing to outsource an already uploaded file D, will compute the same *message-locked* key thanks to the newly proposed ML.KeyGen protocol; K_D together with the public parameters will be subsequently used by ML.Encode to output the same verifiable file $\mathcal{D}$. Thanks to their deterministic nature, ML.KeyGen and ML.Encode enable C to perform cross-user deduplication while still providing cryptographic assurances on the retrievability of the outsourced files.

6.6 Security analysis

Security guarantees of PoR. Proofs of retrievability ensure that if the cloud storage provider C succeeds in providing a valid PoR (i.e. proof that passes the verification at the user) for some outsourced data object $\mathcal{D}$, then one can have the assurance that $\mathcal{D}$ is stored in its entirety and can be correctly retrieved from C. This security guarantee derives from a combination of erasure codes that allow file recovery from accidental errors, and integrity mechanisms that detect deliberate file corruption. More specifically, the security of PoR mechanisms relies on two measures:

- hiding the dependencies between data and redundancy symbols through semantically-secure encryption and secure permutations;
- authenticating the outsourced files either by inserting (unforgeable) watchdogs (c.f. [9, 15]) or by computing (unforgeable) tags (cf. [10, 19]).

Security of message-locked PoR and file unpredictability. Note that the security of our *message-locked* PoR is assured so long as the key K_D derived in the key generation phase is not compromised. By having access to the secret key used during upload, cloud storage provider C not only can compromise the confidentiality of the file, but can also corrupt the uploaded file without being detected. Notably, C can mount the following types of attacks:

- Use the secret key K_D to find the dependencies between data and redundancy symbols by decrypting the file and inverting the permutations used to shuffle the symbols. In this fashion, C can corrupt the file in such a way that the file becomes irretrievable while ensuring that the probability of detecting the tampering is negligible.
- In the case of tag-based PoR schemes, given the secret key K_D cloud storage provider C can modify the data symbols and compute the corresponding tags correctly.
- In the case of watchdog-based PoR scheme, the secret key K_D enables C to discover which symbols are data symbols and which are watchdogs, and accordingly, it can keep only the watchdogs and get rid of the data symbols.

Taking these attacks into account, we conclude that similarly to previous work on secure deduplication, the security of our scheme is closely tied to how well C guesses the content of the data object file: namely, the unpredictability of the uploaded file. Nevertheless thanks to our $\mathsf{ML.KeyGen}$ protocol, cloud storage provider C cannot run offline dictionary attacks, as it must go online and connect to the key server KS to generate the secret key K_D and test the correctness of its guesses.

Rate limiting. The confidentiality and the integrity guarantees of our scheme depend on the *unpredictability of the file* and *the number of message-locked key generation queries* a user is allowed to issue. Intuitively, the more predictable the file is, the less the number of key generation queries an adversary (e.g. cloud storage provider C) has to make to divulge its content; inversely, the more key generation queries an adversary makes, the more predictable the file becomes (by ruling out the files that do not match). It follows that in order to contain the threat of online attacks, it is important to limit the number of such queries a given user makes. This in reality will be implemented through authentication mechanisms: both cloud storage provider C and key server KS will authenticate and identify (and therewith keep track of) users submitting upload queries. Still as rightly pointed out by [68], C or KS for instance can masquerade as any number of users voiding thus the benefits of any rate limiting countermeasure. This illustrates that in order for rate limiting to work, we need to put in place mechanisms to verify the identity of users engaging in the key derivation protocol. One approach to achieve this is to link the user's identity to the user's IP address; however, this approach can be insufficient if users mount spoofing attacks. A more secure alternative, albeit more costly is to have an *identity manager* –as in [57]– that verifies the identity of the users and provides these with the credentials necessary to operate the functionalities of the storage service.

6.7 Performance analysis

Performance of ML.PoR. Thanks to the newly proposed *message-locked* PoR scheme, cloud storage provider C succeeds in saving storage space using deduplication techniques and users are provided with some guarantees with respect to the integrity of their data. The generation of a *message-locked* key and the use of deterministic operations for the pre-processing (encoding) of the outsourced files enable C to discover redundant data and further perform deduplication while still being able to produce cryptographic proofs for data retrievability. The only additional cost added by the newly designed ML-PoR schemes, compared with their original versions is the one originating from the *server-aided message-locked key generation* protocol which is mandatory to ensure the security of ML.PoR.

Tables 6.3 and 6.4 illustrates this overhead for the instantiation of two *massage-locked* PoR schemes, namely, ML.Private-Compact-PoR and ML.StealthGuard. We analyze the cost of each algorithm –Encode, Prove, and Verify– of these two schemes with respect to a file D of size 4 GB and on the basis of a PoR assurance equivalent to a security parameter $\lambda := 45$ as defined in [15]. Computation costs are represented with exp for

Scheme	**Parameters**	U	KS	C	**Storage (20 copies)**
Private Compact PoR [10]	Symbol size: 80 bits Symbols/segment: 160 Tag size: 80 bits	1 EC 1 Enc 5.4×10^6 PRF 1.1×10^9 mult 4.3×10^8 PRP	$\perp$	$\perp$	$\mathcal{D}$: 80 GB Tags: 420 MB
ML.Private-Compact-PoR	Symbol size: 80 bits Symbols/segment: 160 Tag size: 80 bits	1 EC 1 Enc 5.4×10^6 PRF 1.1×10^9 mult 4.3×10^8 PRP 4 exp	1 exp	1 exp	$\mathcal{D}$: 4 GB Tags: 21 MB
StealthGuard [15]	Symbol size: 256 bits Symbols/segment: 4096	1 EC 1 Enc 2.6×10^5 PRF 1.3×10^8 PRP	$\perp$	$\perp$	$\mathcal{D}$: 80 GB Watchdogs: 160 MB
ML.StealthGuard	Symbol size: 256 bits Symbols/segment: 4096	1 EC 1 Enc 2.6×10^6 PRF 1.3×10^8 PRP 4 exp	1 exp	1 exp	$\mathcal{D}$: 4 GB Watchdogs: 8 MB

Table 6.3: Computation ans storage overhead imposed by ML.PoR to the underlying PoR schemes during the outsource processing (algorithms KeyGen and Encode) of a 4 GB file D.

Scheme	Prove	Verify	**Bandwidth**	
			Out	**In**
Private Compact PoR [10]	7245 mult	45 PRF 365 mult	1.9 KB	1.6 KB
ML.Private-Compact-PoR	7245 mult	45 PRF 365 mult	1.9 KB	1.6 KB
StealthGuard [15]	6.3×10^8 mult	1.4×10^5 mult 1719 PRF	23.4 MB	26.2 MB
ML.StealthGuard	6.3×10^8 mult	1.4×10^5 mult 1719 PRF	23.4 MB	26.2 MB

Table 6.4: Computation and communication overhead imposed by ML.PoR to algorithms Prove and Verify of the underlying PoR schemes for a 4 GB file D.

exponentiation, mult for multiplication, PRF for pseudo-random function and PRP for pseudo-random permutation. Table 6.3 shows the overhead imposed on the outsourcing process (algorithms KeyGen and Encode) from the *message-locked key generation* protocol is 4 exponentiations at user U and one at cloud storage provider C and key server KS respectively. Since algorithms KeyGen and Encode are executed only once for every file D user U outsources to C, this overhead can be considered as minimal compared to the computational cost of the remaining algorithms. Moreover, the last column of Table 6.3

shows a scenario where 20 users outsource to cloud storage provider C the same 4 GB file. The advantages of our *message-locked* PoR solution are clear as the only storage overhead in this case is the one of the integrity values of the underlying PoR schemes whereas in the non message-locked case the storage overhead is linear to the number of users. Table 6.4 illustrates that ML.PoR does not induce any additional computational or communication cost to algorithms Chall, Prove, and Verify of the underlying PoR scheme. Concerning performance of our *server-aided message-locked key generation* protocol, the proposed ML.KeyGen protocol, compared to existing solutions [59, 64], relaxes the trust towards KS at the cost of one more communication round.

ML.PoR with client-side deduplication. The currently proposed *message-locked* PoR schemes can achieve even more savings by implementing a client-side deduplication strategy whereby a user U uploads a file object $\mathcal{D}$ only if not already stored. Consequently, both users and the cloud storage provider C benefit from bandwidth savings. To enable client-side deduplication, a user U only needs to execute ML.KeyGen and ML.Encode and to check whether the resulting verifiable data object $\mathcal{D}$ is already stored at C (e.g. by hashing this file and comparing it with a hash table stored at C). However, as noted in [61], client-side deduplication is exposed to some attacks launched by potentially malicious users: an adversary that discovers the identifier of a file can claim possession of it. To circumvent such attacks, solutions in the literature [63, 64] propose to combine client-side deduplication mechanisms with Proofs of Ownership (PoW) [65] which help cloud storage provider C to verify that a user U *owns* a file without the need to upload it.

A solution that implements client-side deduplication using PoW, would require user U to first execute the most costly algorithm of *message-locked* PoR – namely, ML.Encode – and thereafter to prove to cloud storage provider C the ownership of the verifiable data object $\mathcal{D}$. Given that the computational cost of current PoW schemes is linear to the size of the file (see Table 2 in [75]), users have to consider the trade-off between the bandwidth savings and the computational overhead of PoW before deciding to use client-side deduplication.

When using client-side deduplication, one might argue that if user U does not upload the data object $\mathcal{D}$, then she should not be burdened with the execution of the algorithm ML.Encode. Instead she could compute the proof of ownership using the original file D. Unfortunately, this approach would break the security of PoR since to be able to verify a PoW proof cloud storage provider C needs to process the original file D. Hence, C would immediately derive the secret *message-locked* key K_D. Thus, the Proof of Ownership should be computed using the verifiable data object $\mathcal{D}$.

6.8 Summary

In this chapter, we proposed the *message-locked* PoR approach which makes sure that all algorithms in a PoR scheme are deterministic and therefore enables file-based deduplication. The two described instantiations of existing PoR solutions (ML.Private-Compact-PoR and ML.StealthGuard) mainly implement a new algorithm ML.Encode, that basically differs from the original algorithm Encode in one aspect: instead of pseudo-randomly generated, the required keying material is derived from the file itself. Thus, for a given file, ML.Encode will always output the same verifiable data object $\mathcal{D}$ irrespective of the identity of the user executing it. As such *message-locked* keys are exposed to dictionary attacks that could be launched by potentially malicious cloud storage providers, the proposed ML.PoR solutions initially call for a server-aided key generation technique, namely, ML.KeyGen which helps in protecting the secrecy of such keys. Thanks to the newly proposed ML.KeyGen solution that involves both the key server and the cloud provider, none of these parties can discover the *message-locked* key alone.

Chapter 7

Concluding Remarks and Future Research

7.1 Summary

Cloud storage is certainly one of the most attractive services offered by cloud computing. Indeed, cloud storage systems offer users the opportunity to outsource their data without the need to invest in expensive and complex storage infrastructure. However, in this new paradigm users lose control over their data and lend it to cloud storage providers. Hence, users lose the ability to safeguard their data using traditional IT and security mechanisms. As a result, technical solutions aiming at establishing users confidence on the services provided by a cloud storage system would be highly beneficial both to users and cloud storage providers.

In this thesis, we focused on the aspect of verifiability in the context of cloud storage systems. We started our analysis with Proofs of Storage: a family of cryptographic protocols that enable a cloud storage provider to prove to a user that the integrity of her data has not been compromised. The user can ask the cloud storage provider to provide such proofs for an outsourced data object without the need to download the entire object in order to verify that the latter is stored correctly – only a small fraction of the data object has to be accessed in order to generate and verify the proof. Our study revealed the limitations of current Proof of Storage protocols with respect to two key characteristics of cloud storage systems, namely, *reliable data storage with automatic maintenance*, and *data deduplication*.

Verification of reliable data storage with automatic maintenance. Reliable data storage relies on *redundancy* and *data repair* mechanisms to detect and restore corrupted

data. Hence, in addition to the integrity of the data outsourced by users, a Proof of Storage scheme must also verify that a cloud storage provider stores sufficient amount of redundancy information to be able to guarantee the maintenance of the data in the face of corruption. However, redundancy is a function of the original data itself. As a result, 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.

In Chapter 3, we introduced the notion of *Proofs of Data Reliability*, a comprehensive verification scheme that aims to resolve the conflict between reliable data storage verification and automatic maintenance.

In Chapter 4, we proposed POROS, our first attempt 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 rest* 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. Finally, we finally proposed an extended version of POROS where users can run multiple instances of the challenge-response protocol in order to increase their trust in the cloud storage system.

Based on the expertise we earned during the design of POROS, we decided to work on a new Proof of Data Reliability scheme that copes with the shortcomings of POROS. More precisely:

- 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.

In Chapter 5, we proposed PORTOS, new scheme that 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. 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.

Conflict between PoS schemes and data deduplication. The integration of Proofs of Storage with deduplication presents several challenges due to the diverging objectives of the two: PoR aims at imprinting the data with retrievability guarantees that are unique for each user whereas deduplication tries, whenever feasible, to factor several data segments submitted by different users into a unique copy kept in storage. The integrity values – tags or watchdogs – resulting from the PoR Encode algorithm are generated using a secret key that is only known to the owner of the file, and thus unique. Therefore, the encoding of a given file by two different users results in two different outputs which cannot be deduplicated.

Inspired by previous attempts in solving the problem of duplicating encrypted data, in Chapter 6 we devised a solutions that addresses the conflict between Proofs of Retrievability and deduplication. More specifically, we proposed the *message-locked* PoR approach which makes sure that all operations of algorithm Encode are deterministic and therefore enables file-based deduplication. The key idea behind *message-locked* PoR is that the required he required keying material is derived from the file itself. Thus, for a given file, different users will always output the same verifiable data object irrespective of their identity. In addition we proposed a novel *message-locked* key generation protocol which is more resilient against off-line dictionary attacks compared to existing solutions.

Proofs of Data Reliability with deduplication. It is worth to mention that the solutions we propose in this thesis – namely, POROS and PORTOS on the one hand, and

ML.PoR on the other hand – are fully compatible. Indeed, both POROS and PORTOS are symmetric schemes. Hence, by replacing the key generation mechanism that is included in algorithm Store with ML.KeyGen, we obtain a *message-locked* Proof of Data Reliability scheme that allows the cloud storage provider to perform *cross-user file-level* deduplication.

7.2 Future Work

Here, we present possible research directions that stem from the results presented in this thesis.

Proofs of Data Reliability with efficient repair. Both of our Proof of Data Reliability schemes, namely POROS and PORTOS, rely on $[n, k]$–MDS codes to generate the required redundancy in order to meet the reliability guarantee t (c.f. Section 3.2). While, MDS codes achieve the optimal performance in terms of the trade-off between failure recovery and storage overhead, in the event of data loss they require the reconstruction of the original file before computing the lost symbols. Hence, other erasure code types that reduce the cost of repair for lost symbols, would greatly benefit honest cloud storage providers. Local Reconstruction Codes (LRC) [76, 77] are a family of erasure codes that fulfill the above requirement. More specifically, LRC codes split the symbols of a data segment in groups and compute two types of redundancy symbols: *global* redundancy symbols that are computed from all symbols of a data segment, and *local* redundancy symbols that are computed for each group. On the one hand, local redundancy symbols reduce the minimum number of symbols needed to regenerate a corrupted symbol and therefore reduce the network overhead and computation cost of the repair. On the other hand, global redundancy allow for a slightly worse trade-off between failure recovery and storage overhead compared to MDS codes.

Assessment of cloud storage provider capabilities. POROS, PORTOS as well as other Proof of Data Reliability schemes that allow for automatic or "semi-automatic" maintenance by the cloud storage provider [36, 37, 44, 51] 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. Hence, there is a need to estimate C's computational capacity in a reliable and accurate manner.

Moreover, all of the above Proof of Data Reliability schemes make the assumption that computation is more expensive than storage, which may not always be true.

Therefore, there is a need for a security model that encompass all operational resources – computation, network, and storage – of a cloud storage system.

Verification/secure deduplication of dynamic data. In this thesis, we focused our efforts on designing verification schemes compatible with the functional requirements of *automatic maintenance*, *reliable data storage*, and *data deduplication* (c.f. Section 1.1). However, there exists an additional functional requirement, namely, support for *dynamic data*, that greatly increases the complexity of integrating security primitives to a cloud storage system. Literature features a number of PoS protocols [18, 19, 22, 23] and Proof of Data Reliability schemes [40–42] that allow for dynamic writes/updates of the verifiable data object. Nonetheless, all of these solutions are applicable only to the object storage setting. Another option for future research is integrate PoS and Proof of Data Reliability protocols with more volatile settings such as block storage or relational data bases. In such a context, the high volume of write/update/delete operations which are performed on data segments of smaller size presents new challenges to the verification of data integrity and data availability. Similar challenges arise in the case of secure deduplication in this context.

Ideally, we would like to have a cloud computing platform that incorporate security primitives a variety of security concerns. Such primitives include solutions for *privacy preserving data processing*, *verifiable data processing*, and *verifiable storage*. Unfortunately, in most cases the security primitives have conflicting requirements and therefore it is not trivial to deploy them side-by-side on the same system. It our belief that this thesis contributes to the realization of such a cloud computing platform.

Bibliography

[1] K. Elkhiyaoui, M. Önen, D. Vasilopoulos, D. V. García, B. G. N. Crespo, R. M. V. Alvarez, H. Ritzdorf, P. Louridas, A. de Caro, A. Kurmus, A. Sorniotti, R. Lescuyer, G. Karame, W. Li, A. Fischer, B. Fuhry, and M. Kohler, "TREDISEC Project, D2.2 Requirements Analysis and Consolidation," http://tredisec.eu/content/d22-requirements-analysis-and-consolidation, December 2015, accessed: 2019-05-21.

[2] "Using Amazon Web Services for Disaster Recovery," October 2014.

[3] "HDFS Architecture Guide," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, accessed: 2017-08-11.

[4] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15, 2015, pp. 213–226.

[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011.

[6] "HDFS RAID," https://wiki.apache.org/hadoop/HDFS-RAID, accessed: 2017-08-11.

[7] R. Ross, M. McEvilley, and J. Oren, "Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems," https://csrc.nist.gov/publications/detail/sp/800-160/vol-1/final, November 2016, accessed: 2019-05-21.

[8] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, 2007.

[9] A. Juels and B. S. Kaliski, Jr., "Pors: Proofs of retrievability for large files," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, 2007.

[10] Shacham, H. and Waters, B., "Compact proofs of retrievability," in *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '08, 2008.

[11] G. Hogben and A. Pannetrat, "Mutant apples: A critical examination of cloud sla availability definitions," in *IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, Bristol, United Kingdom, December 2-5, 2013, Volume 1*, 2013.

[12] C. Xing and S. Ling, *Coding Theory: A First Course*. Cambridge University Press, 2003.

[13] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. CRC Press, 2014.

[14] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '82, 1982.

[15] M. Azraoui, K. Elkhiyaoui, R. Molva, and M. Önen, "Stealthguard: Proofs of retrievability with hidden watchdogs," in *Proceedings of the 19th European Symposium on Research in Computer Security*, ser. ESORICS '14, 2014.

[16] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, Jun. 2011.

[17] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, ser. SecureComm '08, 2008.

[18] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, 2009.

[19] B. Chen and R. Curtmola, "Robust dynamic remote data checking for public clouds," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[20] J. Xu and E.-C. Chang, "Towards efficient proofs of retrievability," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '12, 2012.

[21] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification theory of cryptography," vol. 2009, 2009.

[22] D. Cash, A. Küpçü, and D. Wichs, "Dynamic proofs of retrievability via oblivious ram," *J. Cryptol.*, vol. 30, no. 1, Jan. 2017.

[23] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013.

[24] R. C. Merkle, "A digital signature based on a conventional encryption function," in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, ser. CRYPTO '87, 1988.

[25] Y. Ren, J. Xu, J. Wang, and J.-U. Kim, "Designated-verifier provable data possession in public cloud storage," *International Journal of Security and Its Applications*, vol. 7, 11 2013.

[26] S.-T. Shen and W.-G. Tzeng, "Delegable provable data possession for remote data in the clouds," in *Proceedings of the 13th International Conference on Information and Communications Security*, ser. ICICS'11, 2011.

[27] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in *Proceedings of the 14th European Conference on Research in Computer Security*, ser. ESORICS'09, 2009.

[28] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 5, May 2011.

[29] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *Proceedings of the 29th Conference on Information Communications*, ser. INFOCOM'10, 2010.

[30] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, "Outsourced proofs of retrievability," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014.

[31] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, ser. CCSW '09, 2009.

[32] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *J. ACM*, vol. 33, no. 4, Aug. 1986.

[33] E.-O. Blass, R. Di Pietro, R. Molva, and M. Önen, "Prism: Privacy-preserving search in mapreduce," in *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, ser. PETS'12, 2012.

[34] J. Trostle and A. Parrish, "Efficient computationally private information retrieval from anonymity or trapdoor groups," in *Proceedings of the 13th International Conference on Information Security*, ser. ISC'10, 2011.

[35] B. Chen and R. Curtmola, "Towards self-repairing replication-based storage systems using untrusted clouds," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13, 2013.

[36] ——, "Remote data integrity checking with server-side repair," *Journal of Computer Security*, vol. 25, 2017.

[37] F. Armknecht, L. Barman, J.-M. Bohli, and G. O. Karame, "Mirror: Enabling proofs of data replication and retrievability in the cloud," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16, 2016.

[38] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "Mr-pdp: Multiple-replica provable data possession," in *Proceedings of the 28th International Conference on Distributed Computing Systems*, ser. ICDCS '08, 2008.

[39] A. F. Barsoum and M. A. Hasan, "Enabling dynamic data and indirect mutual trust for cloud computing storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, Dec. 2013.

[40] ——, "Integrity verification of multiple data copies over untrusted cloud servers," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID '12, 2012.

[41] ——, "Provable multicopy dynamic data possession in cloud computing systems," *IEEE Transactions on Information Forensics and Security*, vol. 10, 2015.

[42] M. Etemad and A. Küpçü, "Transparent, distributed, and replicated dynamic provable data possession," in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ser. ACNS'13, 2013.

[43] M. van Dijk, A. Juels, A. Oprea, R. L. Rivest, E. Stefanov, and N. Triandopoulos, "Hourglass schemes: How to prove that cloud files are encrypted," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[44] I. Leontiadis and R. Curtmola, "Secure storage with replication and transparent deduplication," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18, 2018.

[45] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, 2009.

[46] B. Chen, A. K. Ammula, and R. Curtmola, "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*, ser. CODASPY '15, 2015.

[47] B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote data checking for network coding-based distributed storage systems," in *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '10, 2010.

[48] A. Le and A. Markopoulou, "Nc-audit: Auditing for network coding storage," in *Proceedings of International Symposium on Network Coding*, ser. NetCod '12, 2012.

[49] ——, "Locating byzantine attackers in intra-session network coding using spacemac," in *Proceedings of International Symposium on Network Coding*, ser. NetCod '10, 2010.

[50] T. P. Thao and K. Omote, "Elar: Extremely lightweight auditing and repairing for cloud security," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC '16, 2016.

[51] K. D. Bowers, M. van Dijk, A. Juels, A. Oprea, and R. L. Rivest, "How to tell if your cloud files are vulnerable to drive crashes," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011.

[52] C. Suh and K. Ramchandran, "Exact-repair mds code construction using interference alignment," *IEEE Trans. Inf. Theor.*, vol. 57, no. 3, Mar. 2011.

[53] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: An optimal scheme for tolerating double disk failures in raid architectures," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ser. ISCA '94, 1994.

[54] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," Cambridge, MA, USA, Tech. Rep., 1996.

[55] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proceedings of the 22 Nd International Conference on Distributed Computing Systems*, ser. ICDCS '02, 2002.

[56] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *Trans. Storage*, vol. 7, no. 4, Feb. 2012.

[57] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A Secure Data Deduplication Scheme for Cloud Storage," in *18th International Conference on Financial Cryptography and Data Security* , ser. FC '14, 2014.

[58] P. Puzio, R. Molva, M. Önen, and S. Loureiro, "PerfectDedup: Secure data deduplication," in *10th International Workshop on Data Privacy Management*, ser. DPM '15, 2015.

[59] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, 2013.

[60] P. Puzio, R. Molva, M. Önen, and S. Loureiro, "ClouDedup: Secure Deduplication with Encrypted Data for Cloud Storage," in *Proceeings of the , 5th IEEE International Conference on Cloud Computing Technology and Science* , ser. CLOUDCOM '13, 2013.

[61] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," vol. 8, 01 2010.

[62] F. Armknecht, C. Boyd, G. T. Davies, K. Gjøsteen, and M. Toorani, "Side channels in deduplication: Trade-offs between leakage and efficiency," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS '17, 2017.

[63] R. di Pietro and A. Sorniotti, "Boosting efficiency and security in proof of ownership for deduplication," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* , ser. ASIACCS '12.

[64] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent Data Deduplication in the Cloud," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security* , ser. CCS '15, 2015.

[65] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proceedings of the 18nd ACM Conference on Computer and Communications Security* , ser. CCS '11, 2011.

[66] J. Camenish, G. Neven, and A. Shelat, "Simulatable adaptive oblivious transfer," in *Proceedings of EUROCRYPT*, 2007.

[67] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," *Cryptology*, 2002.

[68] J. Liu, N. Asokan, and B. Pinkas, "Secure Deduplication of Encrypted Data Without Additional Independent Servers," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security* , ser. CCS '15, 2015.

[69] R. Chen, Y. Mu, G. Yang, and F. Guo, "Bl-mle: Block-level message-locked encryption for secure large file deduplication," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, 2015.

[70] Q. Zheng and S. Xu, "Secure and efficient proof of storage with deduplication," in *Proceedings of the 2nd ACM conference on Data and Application Security and Privacy* , ser. CODASPY '12, 2012.

[71] Y. Shin, J. Hur, and K. Kim, "Security weakness in the proof of storage with deduplication," Cryptology ePrint Archive, Report 2012/554, 2012, http://eprint.iacr.org/2012/554.

[72] F. Armknecht, J.-M. Bohli, D. Froelicher, and G. Karame, "Sharing proofs of retrievability across tenants," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS '17, 2017.

[73] N. Koblitz and A. Menezes, "Pairing-based cryptography at high security levels," in *Proceedings of the 10th International Conference on Cryptography and Coding*, ser. IMA'05, 2005.

[74] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," *Discrete Appl. Math.*, vol. 156, no. 16, Sep. 2008.

[75] L. Gonzales-Manzano and A. Orfila, "An efficient confidentiality-preserving Proof of Ownership for deduplication," *Journal on Network and Computer Applications*, vol. 50, 2015.

[76] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, 2012.

[77] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar, "Codes with local regeneration and erasure correction," *IEEE Trans. Information Theory*, vol. 60, no. 8, 2014.

Appendix A

Résumé Français

Réconcilier les fonctionnalités de stockage cloud avec les besoins de sécurité:
Preuves de stockage avec la fiabilité des données et la déduplication sécurisée

A.1 Preuves de stockage

Grâce aux divers avantages tels que des infrastructures de stockage hautement efficaces et fiables et des gains en termes de coûts pour les entrerprises, l'adoption de la technologie *cloud* progresse considérablement. Amazon S3 et Google Drive, parmi les principaux fournisseurs de *cloud*, offrent de nos jours un espace de stockage illimité presque gratuitement. Alors que le nombre d'utilisateurs et le volume de données stockées ne cessent d'augmenter, les préoccupations se multiplient: d'une part, en externalisant le stockage de leurs données, les clients confient le contrôle total de leurs données à des fournisseurs de *cloud* et ne disposent d'aucun moyen de vérifier l'intégrité de leurs services; d'autre part, les fournisseurs de services cloud font face à une augmentation de la capacit'e de stockage exponentielle qui devient extrêmement difficile à contrôler.

Un système de stockage cloud doit avoir les propriétés suivantes:

- *Intégrité des données.* L'intégrité des données est un concept fondamental de sécurité qui consiste à s'assurer de leur crédibilité et fiabilité tout au long de leur cycle de vie. En d'autres termes, l'intégrité garantit que les données ne sont ni supprimées ni modifiées de quelque manière que ce soit par des parties non autorisées.

- *Disponibilité des données.* La disponibilité est une propriété de sécurité fondamentale qui garantit que les données sont accessibles à toutes les parties autorisées à tout moment. En d'autres termes, la disponibilité assure à l'utilisateur qu'il peut télécharger ses données en cas de besoin.

- *Stockage de données fiable.* Un stockage de données fiable peut être considéré comme le mécanisme sous-jacent qui garantit l'*intégrité des données* et la *disponibilité des données*. Les systèmes de stockage doivent être conçus de manière à ce que les données des utilisateurs puissent être récupérées en cas de panne matérielle ou de logiciel défectueux. Cette propriété est obtenue en ajoutant de la redondance au système de stockage afin de tolérer les pannes. Deux groupes de solutions de fiabilité existent dans l'état de l'art:

  - La *réplication des données*, qui consiste à stocker des copies de la totalité des données sur plusieurs différents serveurs de stockage.
  - Les *codes correcteurs* (erasure coding) qui consiste à encoder les données avant de stocker certains fragments chez différents serveurs de stockage. En cas de défaillance d'un ou de plusieurs périphériques de stockage, le système de stockage utilise ces correcteurs et la partie des données sur les périphériques de stockage sains restants pour reconstruire les données.

  Le fournisseur de cloud est responsable de l'intégration de ces mécanismes dans son infrastructure.

- *Déduplication des données* La déduplication est le processus par lequel un système de stockage inspecte les données qu'il stocke, identifie de grandes parties de données répétées, telles que des fichiers entiers ou des parties de fichiers, et n'en garde qu'une seule copie.

- *Maintenance automatique.* La maintenance automatique est une propriété essentielle des systèmes de stockage cloud. Cela implique que toutes les opérations de gestion de données telles que l'allocation de stockage, la réduction de données, la génération de redondance et la réparation de données sont effectuées par le système de stockage cloud de manière transparente pour l'utilisateur.

La perte de données est une des plus grandes menaces pour le *cloud*. Le terme de *perte de données* se réfère à non seulement la suppression non autorisée de données, mais aussi à la modification irréversible de toute ou partie des données. En d'autres termes, la perte de données compromet l'intégrité et la disponibilité de ces données.

Les propriétaires des données confiées au *cloud* devraient pouvoir vérifier que le fournisseur de *cloud* les stocke correctement, c'est-à-dire, vérifier que les données sont intactes et disponibles tout au long de la période de stockage. Cette problématique est abordée dans le domaine de la recherche en **preuves de stockage**. Ces preuves permettent au propriétaire de données de les confier au *cloud* tout en ayant la capacité de vérifier que le *cloud* les stocke correctement. Les preuves de stockage (PoS) sont des preuves cryptographiques qui sont générées et vérifiées dans le contexte d'un protocole entre le propriétaire de la donnée (l'utilisateur) et le *cloud*.

A.2 Problématique

Cette thèse tente de répondre aux deux problématiques suivantes :

A.2.1 Vérification du stockage fiable des données.

Les fournisseurs de stockage cloud n'assument actuellement aucune responsabilité pour la perte de données. Par conséquent, les utilisateurs sont réticents à adopter des services de stockage cloud, car ils délèguent un contrôle total de leurs données au fournisseur cloud sans aucun moyen pour vérifier l'exactitude des mécanismes de stockage de données fiables déployés par le fournisseur de cloud. Par conséquent, des solutions techniques visant à établir la confiance des utilisateurs en ce qui concerne l'intégrité et la disponibilité de leurs données seraient très bénéfiques à la fois pour les utilisateurs et les fournisseurs de services de stockage cloud. En ce qui concerne l'intégrité des données reçues, les solutions de preuve de stockage sont de nos jours compatibles avec des mécanismes de fiabilité de données.

Cependant, les solution de PoS (preuves de stockage) ont une valeur limitée en ce qui concerne l'audit de mécanismes de stockage de données fiables: en effet, lorsqu'une preuve PoS est vérifée étant valide, ceci n'indique pas si un fournisseur cloud a mis en place des mécanismes de fiabilité de données. Même dans le cas des schémas de preuve de récupérabilité (PoR) qui fournissent une définition plus solide, une vérification infructueuse indique que les données ne sont plus récupérables. Détecter qu'un fichier externalisé est corrompu n'aidera pas beaucoup un client/utilisateur car ce dernier n'est plus récupérable. Malgré le fait que les schémas PoR reposent sur des codes correcteurs, les informations de redondance pertinentes ne sont pas destinées aux opérations de réparation de données types: Les codes correcteurs aident à la réalisation des propriétés de sécurité PoR en permettant la récupération des données d'origine à partir d'erreurs accidentelles pouvant passer inaperçues du protocole PoR. Toutefois, ni le fournisseur de stockage cloud C ni

l'utilisateur U ne peut utiliser cette redondance afin de réparer les données corrompues externalisées vers C car, selon le modèle PoR, C ne peut pas distinguer les données d'origine des informations de redondance et, les moyens de détecter toute corruption de données avant qu'elle ne soit irréparable.

En outre, dans le contexte conflictuel du stockage externalisé, il semble exister un conflit inhérent entre les exigences des clients en matière de vérification de mécanismes de stockage de données fiables et une caractéristique essentielle des systèmes de stockage modernes, à savoir la *maintenance automatique*. D'une part, la maintenance automatique basée sur des codes de réplication ou les codes correcteurs nécessite le stockage d'informations redondantes avec les données et, d'autre part, pour garantir le stockage de la redondance, le fournisseur de stockage cloud C ne doit pas avoir accès au contenu des données, car les informations de redondance sont fonction des données d'origine elles-mêmes. Par conséquent, la cause fondamentale du conflit entre la vérification d'un stockage de données fiable et la maintenance automatique provient du fait que la redondance est une fonction des données d'origine lui-même. Cette propriété, qui est à la base de la maintenance automatique, peut être exploitée par un fournisseur de stockage malveillant afin d'économiser de l'espace de stockage tout en répondant de manière positive à la vérification des critères de stockage fiable des données. Un fournisseur de stockage malveillant peut en effet prouver sa possession de la redondance en calculant simplement et sur demande cette dernière à l'aide de sa capacité de maintenance automatique sans jamais stocker d'informations de redondance. Même si un tel fournisseur de stockage serait en mesure de répondre avec succès aux requêtes de vérification de la fiabilité des données, à la manière d'un système de point de vente, la fiabilité réelle des données ne serait pas nécessairement assurée, car le fournisseur de stockage ne pourrait pas récupérer les segments de données perdus ou corrompus sans l'information redondante. La maintenance automatique, qui est une caractéristique très efficace de la fiabilité des données, peut ainsi devenir le principal moyen de tromper la vérification de la fiabilité des données dans un contexte conflictuel.

A.2.2 Conflit entre PoR et déduplication.

Il semble que la combinaison simple d'une solution PoR avec la déduplication soit vouée à l'échec en raison d'un conflit inhérent entre les schémas de PoR actuels et les schémas de déduplication. La cause fondamentale du conflit est que les schémas PoR et la déduplication appellent des objectifs divergents: PoR a pour but d'ajouter aux données des informations de garantie de récupérabilité qui sont uniques pour chaque utilisateur, tandis que la déduplication tente, dans la mesure du possible, de ne garder qu'une seule copie de segments de données similaires soumis par différents utilisateurs. En effet, à la fois dans

les schémas PoR basés sur des watchdogs qui sont générés de manière pseudo-aléatoire et inséré à des positions aléatoires des données et les schémas PoR basés sur des balises (tag) qui ajoutent une balise non modifiable à chaque segment de données, la combinaison simple avec les techniques de déduplication échouerait. En effet, l'algorithme Encode de ces schémas inclut un cryptage sémantiquement sécurisé par chaque utilisateur ce qui empêche la détection des données dupliquées.

A.3 Contributions

Cette thèse propose les contributions suivantes en rapport avec les deux problèmes énoncés plus haut.

Vérification de la fiabilité des données stockées Nous introduisons la notion de *Preuves de fiabilité des données*, un schéma de vérification complet visant à résoudre le conflit entre la vérification fiable du stockage des données et la maintenance automatique. En particulier, nous fournissons la définition d'un protocole de preuve de stockage cryptographique et d'un nouveau modèle de sécurité contre un adversaire *rationnel*. Nous passons également en revue les solutions antérieures visant à concevoir une solution pratique de preuve de fiabilité des données. Enfin, nous proposons deux schémas de preuve de fiabilité des données qui permettent de vérifier le mécanisme de stockage de données fiable et permettent en même temps au fournisseur de stockage cloud d'effectuer de manière autonome des opérations de maintenance automatiques.

Conflit entre PoR et déduplication. Inspirés des solutions précédentes pour résoudre le problème de la déduplication de données cryptées, nous proposons une solution simple en combinant PoR et déduplication. De plus, nous proposons un nouveau protocole de génération de clés cryptographique pour les chiffrement dépendantes des données qui résiste mieux aux attaques de dictionnaire hors ligne par rapport aux solutions existantes.

A.4 Preuves de fiabilité des données

Par définition, trois entités sont impliqués dans un schéma de preuve de fiabilité des données. Celles -ci sont:

l'utilisateur U qui souhaite stocker son fichier D chez un fournisseur de stockage cloud C.

le fournisseur de stockage cloud C qui s'engage à stocker le fichier D dans son intégralité avec une redondance suffisante générée par des mécanismes de stockage de données fiables. Le fournisseur de stockage cloud C est considéré comme *malveillant*.

le vérificateur V qui interagit avec le fournisseur de stockage cloud C dans le contexte d'un protocole challenge-response et qui valide si C stocke le fichier D dans son intégralité.

Nous considérons un scénario dans lequel un utilisateur U enovie un fichier D vers un fournisseur de stockage cloud C et, dorénavant, un vérificateur V interroge périodiquement C pour obtenir des preuves d'intégrité et de stockage fiable des données de D. En réalité, l'utilisateur U génère un objet de données vérifiable $\mathcal{D}$ à partir du fichier D, qui contient des informations supplémentaires qui aideront davantage à la vérification de son stockage fiable. Si le schéma de fiabilité des données n'est pas compatible avec la maintenance automatique, $\mathcal{D}$ incorpore également la redondance requise pour le stockage fiable du fichier D. À ce stade, l'utilisateur U télécharge l'objet de données $\mathcal{D}$ vers le fournisseur de stockage cloud C et supprime le fichier D et l'objet de données $\mathcal{D}$ en ne conservant dans le stockage local que les éléments de clé utilisés $\mathcal{D}$. À son tour, le fournisseur de stockage cloud stocke $\mathcal{D}$ dans un ensemble de n nœuds de stockage avec *garantie de fiabilité* t: certaines garanties de service de stockage contre les défaillances de nœud de stockage t.

Nous définissons un schéma *preuve de fiabilité des données* comme un protocole exécuté entre le fournisseur de stockage cloud C avec ses noeuds de stockage $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$, d'une part, et un vérificateur V, de l'autre. Le but d'un tel protocole est de permettre au vérificateur V de vérifier (i) la *intégrité* de $\mathcal{D}$ et (ii) si la *garantie de fiabilité* t est satisfaite. Le vérificateur V lance un défi de preuve de fiabilité des données, qui fait référence à un sous-ensemble de données et de symboles de redondance, et l'envoie au fournisseur de stockage cloud C. Désormais, C génère une preuve et V vérifie si cette preuve est bien formée et l'accepte ou la rejette en conséquence. Afin de ne pas annuler les avantages en termes de stockage et de performances du cloud, vous devez effectuer toute cette vérification sans que V télécharge tout le contenu associé à $\mathcal{D}$ à partir de $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$.

Definition 9. (Schéma de preuve de fiabilité des données). *Un schéma de preuve de fiabilité des données est défini par sept algorithmes temporels polynomiaux:*

- $\mathsf{Setup}\ (1^\lambda, t) \rightarrow (\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}, \mathsf{param_{system}})$*: Cet algorithme prend en entrée le paramètre de sécurité* λ *et le paramètre de fiabilité* t *et renvoie l'ensemble des nœuds de stockage* $\{\mathcal{S}^{(j)}\}_{1\leq j\leq n}$*, les paramètres système* $\mathsf{param_{system}}$ *et la spécification de la redondance. mécanisme: le nombre de répliques ou le schéma de code d'effacement qui sera utilisé pour générer la redondance.*

- Store $(1^\lambda, D) \rightarrow (K_\mathsf{u}, K_\mathsf{v}, \mathcal{D}, \mathsf{param}_\mathcal{D})$*: Cet algorithme randomisé appelé par l'utilisateur* U *prend en entrée le paramètre de sécurité* λ *et le fichier à externaliser* D *et génère la clé d'utilisateur* K_u*, la clé de vérificateur* K_v *et l'objet de données vérifiable* $\mathcal{D}$*, qui inclut également un identificateur unique* fid *et éventuellement un ensemble de paramètres d'objet de données* $\mathsf{param}_\mathcal{D}$*.*

- GenR $(\mathcal{D}, \mathsf{param}_\mathsf{system}, \mathsf{param}_\mathcal{D}) \rightarrow (\tilde{\mathcal{D}})$*: Cet algorithme prend en entrée l'objet de données vérifiable* $\mathcal{D}$*, les paramètres système* $\mathsf{param}_\mathsf{system}$ *et, éventuellement, les paramètres de l'objet de données* $\mathsf{param}_\mathcal{D}$*, et génère l'objet de données* $\tilde{\mathcal{D}}$*. L'algorithme* GenR *peut être appelé par l'utilisateur* U *lorsque l'utilisateur crée lui-même la redondance, ou par* C*, lorsque le calcul de la redondance est entièrement sous-traité au fournisseur de stockage cloud. Selon le mécanisme de redondance,* $\tilde{\mathcal{D}}$ *peut comprendre plusieurs copies de* $\mathcal{D}$ *ou une version codée de celui-ci. De plus, algorithm* GenR *génère les valeurs d'intégrité nécessaires qui faciliteront la vérification de l'intégrité de la redondance de* $\tilde{\mathcal{D}}$*.*

- Chall $(K_\mathsf{v}, \mathsf{param}_\mathsf{system}) \rightarrow (\mathsf{chal})$*: Cet algorithme stateful et probabiliste appelé par le vérificateur* V *prend en entrée la clé du vérificateur* K_v *et les paramètres système* $\mathsf{param}_\mathsf{system}$ *et génère un défi* chal*.*

- Prove $(\mathsf{chal}, \tilde{\mathcal{D}}) \rightarrow (\mathsf{proof})$*: Cet algorithme appelé par* C *prend en entrée le défi* chal *et l'objet de données* $\tilde{\mathcal{D}}$*, et renvoie la réponse* C *de la preuve de fiabilité des données.*

- Verify $(K_\mathsf{v}, \mathsf{chal}, \mathsf{proof}, \mathsf{param}_\mathcal{D}) \rightarrow (\mathsf{dec})$*: Cet algorithme déterministe appelé par* V *prend en entrée la preuve de* C *correspondant à un défi* chal*, à la clé de vérificateur* K_v *et, éventuellement, aux paramètres d'objet de données* $\mathsf{param}_\mathcal{D}$*, et génère une décision* $\mathsf{dec} \in \{\mathsf{accept}, \mathsf{reject}\}$ *indiquant une vérification réussie ou échouée de la preuve, respectivement.*

- Repair $({}^*\tilde{\mathcal{D}}, \mathsf{param}_\mathsf{system}, \mathsf{param}_\mathcal{D}) \rightarrow (\tilde{\mathcal{D}})$*: Cet algorithme prend en entrée un objet de données corrompu* ${}^*\tilde{\mathcal{D}}$ *avec ses paramètres* $\mathsf{param}_\mathcal{D}$ *et les paramètres système* $\mathsf{param}_\mathsf{system}$*, puis reconstruit* $\tilde{\mathcal{D}}$*. ou génère un symbole d'échec* $\perp$*. L'algorithme* Repair *peut être invoqué par* U *ou* C *selon le schéma Preuve de fiabilité des données.*

A.4.1 POROS: Preuve de la fiabilité des données pour le stockage externalisé.

Nous décrivons maintenant notre approche pour permettre à un système de stockage cloud de fournir des garanties de fiabilité des données sans perturber ses opérations de maintenance automatique, malgré le conflit entre ces dernières et les systèmes de preuve de

stockage. La cause fondamentale de ce conflit est le simple fait que les informations de redondance sont fonction des données d'origine elles-mêmes: Les mécanismes de maintenance automatique exigent que le système de stockage cloud ait un accès sans obstruction à la redondance afin de réparer les données corrompues; malheureusement, cela offre à un fournisseur de stockage cloud malveillant C la possibilité de réaliser des économies de stockage en ne stockant pas la redondance alors qu'elle répond effectivement aux critères de preuve de fiabilité des données. Par conséquent, une nouvelle approche, qui traite la redondance séparément des données d'origine, visant à garantir aux utilisateurs que celle-ci est réellement stockée, semble être le bon moyen de résoudre ce conflit. Une telle assurance rend inoffensive la divulgation de la relation entre les données d'origine et la redondance au fournisseur de stockage cloud C, permettant ainsi des opérations de maintenance automatiques efficaces sans interaction avec l'utilisateur. De plus, l'utilisation de mécanismes de stockage de données fiables par C facilite également la sous-traitance du calcul des informations de redondance: C est la partie qui invoque l'algorithme GenR. En conséquence, la phase de sous-traitance de notre système Preuve de fiabilité des données devient considérablement plus légère pour les utilisateurs, car ils ne sont plus obligés d'effectuer cette opération très lourde en calculs. Toutefois, un nouveau problème de sécurité se pose, car les utilisateurs ont maintenant besoin de moyens pour vérifier le calcul correct des informations de redondance par ce C non approuvé.

En ce qui concerne la vérification de l'intégrité des données d'origine, notre schéma utilise les *balises linéairement homomorphes* utilisées dans le schéma de Private Compact PoR proposé par Shacham et Waters afin de construire un schéma de preuve de possession de données (PDP) garantissant: la détection éventuelle de toute tentative par un fournisseur de stockage cloud malveillant C de falsifier les données externalisées. Avant de télécharger leurs données sur le système de stockage cloud, les utilisateurs calculent un ensemble de balises linéairement homomorphes qui sont ensuite utilisées par C pour prouver le stockage des données d'origine.

Après avoir reçu l'objet de données $\mathcal{D}$ comprenant les données d'origine de l'utilisateur et ses balises associées, le fournisseur de stockage cloud C appelle l'algorithme GenR qui génère la redondance de $\mathcal{D}$ en fonction du mécanisme de stockage fiable des données. Afin de faciliter le traitement séparé des données d'origine et des informations de redondance, nous optons pour un *code linéaire systématique* permettant une délimitation claire entre les deux: ainsi, les symboles de redondance sont une combinaison linéaire des symboles de $\mathcal{D}$ et ces derniers ne sont pas modifiés par l'application du code d'effacement, d'où la vérification de l'intégrité de $\mathcal{D}$ - par le biais du schéma PDP - n'est pas affecté. En ce qui concerne la vérification de l'intégrité des symboles de redondance, notre système utilise également

le protocole PDP utilisé pour l'objet de données $\mathcal{D}$ et offre donc aux utilisateurs les deux garanties. Plus précisément, comme pour les symboles de l'objet de données $\mathcal{D}$, algorithm GenR applique le même code linéaire systématique aux balises associées, générant ainsi un nouvel ensemble de balises constituant des combinaisons linéaires des balises de $\mathcal{D}$. En supposant que les balises de notre schéma PDP soient *homomorphes* par rapport au code linéaire systématique utilisé par le mécanisme de stockage de données fiable, les balises résultant de ce calcul s'avèrent être des balises PDP associées à des symboles de redondance. En d'autres termes, la combinaison linéaire des étiquettes associées aux données d'origine peut être utilisée pour vérifier à la fois le calcul correct et l'intégrité des symboles de redondance, qui sont eux-mêmes la même combinaison linéaire de symboles de données d'origine.

Grâce à l'homomorphisme du système de balises sous-jacent au cœur de notre protocole PDP et au code linéaire systématique, le fournisseur de stockage cloud C n'a pas besoin de clés cryptographiques appartenant aux utilisateurs pour calculer les balises pour les informations de redondance. En outre, les utilisateurs peuvent effectuer une vérification PDP sur les informations de redondance à l'aide de ces nouvelles balises. Toute inconduite de C concernant l'intégrité des symboles de redondance ou leur génération appropriée sera finalement détectée par la vérification PDP, car le C malveillant ne peut pas falsifier les balises calculées.

Le schéma décrit jusqu'ici souffre d'une limitation en ce qu'un fournisseur de stockage cloud malveillant C peut tirer parti de sa capacité à calculer de manière indépendante les informations de redondance et les balises correspondantes, ouvrant la voie pour que C trompe la vérification de la preuve de fiabilité des données. en calculant en temps réel les réponses aux contrôles PDP sur les symboles de redondance. La dernière caractéristique de notre plan est donc une contre-mesure à ce type d'attaques. Cette contre-mesure repose sur les fonctionnalités de synchronisation des disques durs en rotation, composant commun des infrastructures de stockage dans le cloud. En raison de leurs caractéristiques techniques, ces lecteurs atteignent un débit beaucoup plus élevé lors de l'exécution de l'accès disque séquentiel par rapport à l'accès disque aléatoire. Ce dernier entraîne l'exécution de plusieurs opérations de recherche coûteuses afin que la tête de disque atteigne les différents emplacements du lecteur. Afin de tirer parti de cette variation de performances entre les deux opérations d'accès au disque, nous exigeons que les informations de redondance soient stockées dans un format personnalisé avec la propriété d'augmenter les opérations d'accès au disque aléatoires d'un fournisseur de stockage en cloud défectueux C. Par conséquent, nous pouvons introduire un seuil de temps $\mathsf{T}_{\mathsf{thr}}$, de sorte que chaque fois que C reçoit un défi de preuve de fiabilité des données, il est obligé de générer et de fournir la preuve avant

que le seuil de temps $\mathsf{T}_{\mathsf{thr}}$ ne soit dépassé; sinon la preuve est rejetée.

A.4.2 PORTOS: Preuve de la fiabilité des données pour un stockage externalisé distribué dans le monde réel.

PORTOS est un schéma de preuve de fiabilité des données conçu pour les systèmes de stockage cloud distribués. Comme pour POROS, PORTOS utilise un code correcteur linéaire systématique pour ajouter de la redondance aux données sous-traitées. Néanmoins, contrairement à POROS, PORTOS stocke les données codées sur plusieurs nœuds de stockage: chaque symbole de mot de code, les données et la redondance, est stocké sur un nœud de stockage distinct. Par conséquent, le système peut tolérer la défaillance de nœuds de stockage jusqu'à t et reconstruire avec succès les données d'origine à l'aide du contenu des nœuds restants.

En ce qui concerne la vérification de l'intégrité des données sous-traitées et leur redondance, PORTOS utilise les mêmes balises PDP (preuve de possession de données) que POROS. Plus spécifiquement, ce schéma PDP s'appuie sur des balises de linéairement homomorphes pour vérifier l'intégrité des symboles de données et tire parti des propriétés homomorphes de ces balises, vérifier l'intégrité des symboles de redondance. De plus, grâce à la combinaison des balises PDP avec le code d'effacement linéaire systématique, PORTOS garantit à l'utilisateur qu'il peut récupérer ses données dans leur intégralité.

Dans PORTOS, le fournisseur de stockage cloud dispose des moyens nécessaires pour générer la redondance requise, détecter les défaillances (matérielles ou logicielles) et réparer les données corrompues de manière autonome, sans aucune interaction avec l'utilisateur. Dans POROS, ce paramètre permet toutefois à un fournisseur de stockage cloud illicite de supprimer une partie des données codées et de calculer les symboles manquants à la demande. Pour se défendre contre une telle attaque, PORTOS s'appuie sur des casses-tête cryptographiques (cryptographic puzzle) pour augmenter les ressources (stockage et calcul) qu'un fournisseur de stockage cloud infidèle doit mettre à sa disposition afin de fournir une preuve valable de la fiabilité des données. Néanmoins, ce mécanisme n'entraîne aucun coût de stockage ou de calcul supplémentaire pour un fournisseur de stockage en cloud honnête qui génère la même preuve. De cette manière, un adversaire rationnel est fortement incité à se conformer au protocole de preuve de fiabilité des données. En conséquence, PORTOS est conforme au modèle actuel de systèmes de stockage distribués basés sur un code d'effacement. De plus, PORTOS ne fait aucune hypothèse concernant la technologie sous-jacente du système de stockage cloud, par opposition à POROS.

A.5 Stockage vérifiable avec déduplication sécurisée

Nous présentons une approche générique qui résout le conflit entre PoR et la déduplication, ouvrant ainsi la voie à une intégration simple de PoR et de la déduplication dans le même système de stockage en nuage. La raison fondamentale du conflit réside dans la différence de traitement des doublons entre segments de données soumis par différents clients: le fait que la déduplication récapitule tous les doublons en une seule copie, tandis que la PoR exige que chaque segment en double soit conservé séparément. afin de préserver l'effet de PoR spécifique à l'utilisateur sur chaque duplicata. Par conséquent, une nouvelle approche visant à obtenir des effets de PoR identiques sur les doublons soumis par différents clients semble être la bonne solution pour une composition simple de PoR avec déduplication. Dans la même direction, la nouvelle approche devrait garantir que l'effet de la PoR sur chaque segment de données dépend de la valeur du segment de données, quelle que soit la différence d'identité des clients soumettant les segments de données. En d'autres termes, ces opérations doivent être: fonction de la valeur du segment de données indépendamment de l'identité du client. Nous définissons donc les schémas PoR compatibles avec la déduplication comme *preuves de récupérabilité verrouillées par message*.

Une approche similaire a été proposée pour résoudre le problème de la déduplication par rapport aux données cryptées: les solutions de déduplication sécurisée utilisent le moyen par lequel la clé de cryptage est dérivée des données. Dans la mesure où un schéma PoR symétrique utilise des clés secrètes lors du processus de codage et de vérification de l'extractibilité des données, nous proposons également que ML-PoR extrait ces clés du fichier. Afin de protéger le secret de ces clés *verrouillées par message* qui peuvent facilement être découvertes si les fichiers sont prévisibles, le PoR proposé *verrouillé par message* utilise une technique de génération de clé dédiée telle que le serveur récemment proposé. techniques de génération de clé assistée qui empêchent les serveurs cloud de découvrir des clés *verrouillées par message* par le biais d'attaques par dictionnaire. De telles techniques génèrent une clé cryptographique qui dépend non seulement du fichier, mais également d'une clé secrète générée par un serveur de clés. Outre la conception d'un PoR *verrouillé par message*, nous proposons une nouvelle technique *génération de clé dépendante du message et avec un serveur de clés* qui, comparée aux solutions existantes, assouplit le modèle de confiance et garantit que ni le serveur cloud ni le serveur de clés ne peut deviner les *clés ou messages*.