Cloud Storage File Recoverability[⋆]

Christian A. Gorke[1], Christian Janson[2], Frederik Armknecht[1], and Carlos Cid[3]

[1] University of Mannheim, Germany
gorke@uni-mannheim.de, armknecht@uni-mannheim.de
[2] Technische Unversität Darmstadt, Germany
christian.janson@cryptoplexity.de
[3] Information Security Group, Royal Holloway, University of London, UK
Carlos.Cid@rhul.ac.uk

Abstract. Data loss is perceived as one of the major threats for cloud storage. Consequently, the security community developed several challenge-response protocols that allow a user to remotely verify whether an outsourced file is still intact. However, two important practical problems have not yet been considered. First, clients commonly outsource multiple files of different sizes, raising the question how to formalize such a scheme and in particular ensuring that *all* files can be simultaneously audited. Second, in case auditing of the files fails, existing schemes do not provide a client with any method to prove if the original files are still recoverable.

We address both problems and describe appropriate solutions. The first problem is tackled by providing a new type of "Proofs of Retrievability" scheme, enabling a client to check all files simultaneously in a compact way. The second problem is solved by defining a novel procedure called "Proofs of Recoverability", enabling a client to obtain an assurance whether a file is recoverable or irreparably damaged. Finally, we present a combination of both schemes allowing the client to check the recoverability of all her original files, thus ensuring cloud storage file recoverability.

Keywords: Proofs of Retrievability, Proofs of Recoverability, Cloud Storage, Cloud Security

1 Introduction

Cloud service providers (CSPs) have gained continuous importance over the last decade, e.g. Amazon AWS, Google Cloud Platform, or Windows Azure. They offer various services in numerous application domains such as storage, computation, and key management. Especially storage has matured into one of the main applications with growing interests. However, as the client loses control

[⋆] A preliminary version of this paper appears in the proceedings of the *5th International Workshop on Security in Cloud Computing (ASIACCS-SCC 2017)*, DOI: https://doi.org/10.1145/3055259.3055264. This is the full version.

over her data, at the same time new security concerns rise. For instance, one of the major risks perceived in the context of cloud storage is the fear of data loss [1].

Proofs of Storage (PoS) [9] allow a client to remotely verify that a server truthfully stores a file. Well-studied examples of a PoS are *Proofs of Retrievability* (PoR) [24, 32] and *Proofs of Data Possession* (PDP) [8]. In a nutshell, such schemes work as follows. Before uploading a file, a user pre-processes it and stores locally some meta information about the file. For verification, a challenge-response-protocol is executed. Here, a challenge covers some blocks of the file and security of the scheme ensures that a provider can only provide a correct response if these blocks are stored entirely. In other words, such schemes aim to ensure for *a single file* that if the provider yields *correct responses*, the outsourced file is *stored correctly.* We argue that these guarantees are often not sufficient in practice and that these gaps require novel solutions.

First, clients commonly outsource multiple files of various sizes. A straightforward approach would be to run PoR over each file. However, the effort scales over all procedures of PoR with the number of files. Alternatively, one may consider to randomly select blocks over the set of all blocks of all files. However, then small files risk to be overlooked regularly.

Second, PoR guarantee that if the response is correct then the *outsourced* file is retrievable. However, PoR provide only limited information about the *original* file in case of wrong responses. At a first glance, one may argue that in case incorrect responses are given (and hence some blocks are missing), the provider neglected his task and is ultimately accountable. However, in practice, the Service Level Agreement (SLA) never guarantees 100% reliable storage, and hence, the CSP could *always* claim that the missing blocks are part of the expected loss [3] (including natural data degradation), which we call *regular data loss.* Hence, there is an inherent gap to provide any assurance at all about the original file as soon as even small loss occurs. This leaves the client with the uncertainty whether it is worth downloading the remaining damaged file hoping to recover the original file since she would have to invest her own resources (storage, communication, and computation). Note that posing another set of large challenges to cover most of the blocks not only imposes a huge communication and computation effort for the CSP which he may not be willing to do so. Large challenges also dramatically increase the detection probability of finding a damaged block which typically results in an incorrect response and no information about the recoverability of the original files. Consequently, this leaves only the alternative to make many short challenges. This induces the questions on determining the optimal challenge size as well as how often those challenges need to be posed to obtain a sufficient level of confidence that an original file can be recovered from the damaged one.

In this work, we propose solutions to both problems. With respect to the multi-file case, we first describe a formal extension of PoR to the multi-file scenario and provide an instantiation. With respect to the second problem, we show how to solve it by using the previous solution as a stepping stone enabling us to formalize proofs of recoverability that provide the required assurance to argue about the retrievability of original files even if some parts of the outsourced version are damaged. Finally, we combine both schemes to give a complete solution for cloud storage file recoverability.

The paper is structured as follows. We begin in Section 2 with reviewing relevant related work. In Section 3, we provide the necessary background as well as discuss insecure straightforward solutions and challenges that arise from accommodating multiple files into the realm of PoR, and discuss problems that occur in the PoR functionality as soon as it detects file corruption. In Section 4, we describe an extension of the PoR framework that captures multiple files. In Section 5, we propose solutions towards the question on how to check retrievability in the context of a potential file corruption. We close the paper in Section 6 with a conclusion.

2 Related Work

Proofs of Retrievability [6, 11, 13, 15, 17, 22, 24, 29, 30, 32, 34, 38, 37] allow a client to store her data on a remote server and provably check that all her data is still fully intact and can be retrieved. The concept was initially defined by Juels and Kaliski [24]. Concurrently, Ateniese et al. [8] proposed a close variant of PoR called Proofs of Data Possession (PDP). The main difference between PoR and PDP is the notion of security they achieve. More precisely, a PoR provides stronger security guarantees than PDP, as a PoR assures that the server maintains full knowledge of the client's processed data whereas a PDP only assures that most of the data is retained. Both concepts have received much research attention.

On the one hand, there are works focusing on the case where the data is static. Here works have been developed that propose improvements [13] compared to [24], offer the use of homomorphic authenticators yielding compact proofs [32], and [17] introducing the notion of PoR codes. The latter one improves the communication costs employing hitting samplers and exponentiation, which was also used by [37] to further shrink the communication costs of the response but increasing the computation burden. In [6] the notion of an outsourced PoR scheme was introduced in which a user can task an external auditor to perform and verify PoR procedures.

On the other hand, some approaches deal with the construction of dynamic schemes supporting efficient updates. Cash et al. [15] achieve dynamic updates using oblivious RAM, whereas [34] improves the performance by relying on a Merkle hash tree. Stefanov et al. [35] consider updates where a trusted "portal" performs operations on the client's behalf. Furthermore, dynamic PDP solutions were proposed in [10] where the problem of dynamic writes/updates is considered, and [18] uses authenticated dictionaries based on rank information. Some works explore the direction to extend works into the multi-server setting [12, 14, 16] and [21] introduces a third party enabling the client to efficiently check the integrity of the data. In particular, Bowers et al. [12] use a related notion of recoverability compared to ours in the multi-server scenario. Here after detecting a file corruption, a *test-and-redistribute protocol* is initiated which recovers the file from uncorrupted samples of other servers and restores it. Guan et al. [22] explore the usage of indistinguishability obfuscation for building a PoR scheme that offers public verification while the encryption process is based on symmetric key primitives. Recently, Armknecht et al. [4] introduce a unified model for proving data replication and data retrievability. Vasilopoulos et al. [36] suggest a similar scheme proposing a message-locked PoR approach rendering the involved algorithms to be deterministic and therefore enabling file-based deduplication. In [5], Armknecht et al. extend the classical PoR scheme accommodating multiple clients proposing a storage efficient PoR solution by using data deduplication. Other contributions [31, 33, 38] deal with public verifiable PoR schemes.

3 Preliminaries

In this section, we briefly recall the PoR model as described in previous work [24, 32]. Furthermore, we discuss shortcomings of the notion since we want to accommodate multiple files and regular data loss in the realm of cloud storage.

3.1 Proofs of Retrievability

Proofs of Retrievability are a minimally interactive protocol between a client and cloud storage provider which cryptographically proves the retrievability of an outsourced file. In more detail, a PoR scheme consist of three basic procedures, namely Setup, Store, and PoRP. The first two procedures basically initialize the scheme as well as prepare the file to be outsourced, i.e., the file is processed with an erasure-correcting code (ECC). An ECC encoding is a process that adds redundant data to the *original* file in such a way that a receiver may recover the original file even when a number of erasures were introduced into the *processed* (outsourced) file[4], either during the transmission of the file, or while storing it. We denote the ECC coding rate by ρ with $0 < \rho \leq 1$. The final procedure PoRP is a minimally interactive protocol between a verifier and a prover determining whether the outsourced file is retrievable by outputting a decision bit $\delta \in \{\mathtt{accept}, \mathtt{reject}\}$.[5] The term minimally refers to a *single* execution of a challenge-response protocol providing a provable statement about the retrievability of the outsourced file. Such a single execution suffices here since the ECC functionality boosts the probability to detect a misbehaving server. In more detail, the *detection probability* of a cheating server is approximately $1-(1-\rho)^{\ell}$ where ℓ corresponds to the size of the challenge which describes the number of different blocks of the processed file that are simultaneously checked. In case a malicious server is detected, the procedure outputs $\delta = \mathtt{reject}$ with overwhelming probability indicating that the *processed* file is not fully intact anymore. In other words, the CSP is misbehaving.

[4] Note that we use the terms "outsourced file" and "processed file" interchangeably.

[5] For a formal definition of PoR please refer to [32].

3.2 Adversarial Model

We consider a stateful rational attacker in form of a malicious storage provider that may try to delete bits, blocks, or rearrange specific files in order to make storage space available, thus obtaining a financial benefit, e.g. by letting the same space multiple times. Since the adversary can precisely choose which bits or bytes to delete within the outsourced file, we call this an *adversarial erasure* strategy. Note that we assume the adversary only deletes data *prior* to a PoRP procedure. In other words, the adversary does not dynamically delete any data while the file is being checked.

3.3 Multiple Files

It is natural that a client aims to store multiple (different-sized) files $F^{(1)}, \dots, F^{(f)}$ at a storage provider and wishes to obtain a provable assurance that the provider is indeed in possession of the files as well as them being retrievable. However, current known PoR proposals do not support the multiple files case well. In more detail, assume one outsources multiple files $F^{(1)}, \dots, F^{(f)}$ to a provider and simply performs a separate PoR for each file individually. However, even if this approach theoretically works, it is inefficient due to the increased workload that scales in the number of files over all procedures. Another approach is to simply concatenate all files into one (large) file $\widehat{F} = F^{(1)} \| F^{(2)} \| \dots \| F^{(f)}$ and execute a PoR scheme for the composed file $\widehat{F}$. Unfortunately, the employed type of ECC encoding used while processing the file becomes the bottleneck rendering this approach to be infeasible. For example, a "concatenated-file ECC" encoding results in a *processed* file of the form $\mathcal{F} = F^{(1)} \| F^{(2)} \| \dots \| F^{(f)} \| P^{(1,\dots,f)} = \widehat{F} \| P^{(1,\dots,f)}$ where $P^{(1,\dots,f)}$ denotes the added redundancy generated over the concatenation of all files. In this particular case it may be possible to use existing PoR notions depending on the size of f, however, this approach suffers from other drawbacks making it an unattractive solution. In more detail, in case a client wishes to update a single file $F^{(i)}$, $i \in [f] := \{1, \dots, f\}$, then she is required to download the whole processed file $\mathcal{F}$ since the redundancy was generated over the concatenated file and thus makes (individual) file updates expensive. Note that the same holds in case the client wants to delete files and that $\mathcal{F}$ may include all outsourced files.

Another type of ECC encoding called "individual-file ECC" results in obtaining a processed file of the form $\mathcal{F} = F^{(1)} \| P^{(1)} \| F^{(2)} \| P^{(2)} \| \dots \| F^{(f)} \| P^{(f)}$ where each original file $F^{(i)}$, $i \in [f]$, is initially processed before all files are concatenated and thus all parity parts $P^{(i)}$ are independent from each other. Here, updating a file $F^{(i)}$ is easier since we can solely download the required file, however, this approach suffers from the "small file problem". In more detail, if some file $F^{(i)}$ of the processed file $\mathcal{F}$ is small (e.g. the file solely consists of a password) then there is a non-negligible probability that within a single PoRP execution this file does *not* get examined while the procedure outputs `accept`, since only a small number of blocks is being checked. However, this acceptance token may be false positive since the procedure has not checked all files and hence the statement cannot provide sufficient assurance about the retrievability of all files. Yet another drawback of this approach is that in case any file and respective parity block is deleted then this specific file is completely deleted. Both approaches suffer from the problem that in case PoRP fails (i.e., outputting `reject`), it is unclear in which file(s) the error has occurred and thus forces the client to download the whole processed file and losing her initial advantage of outsourcing the files in the first place.

To overcome the above problems we introduce a new PoR notion called *cloud storage proofs of retrievability* (CSPoR) in Section 4.

3.4 Recovering Corrupted Files

So far, a single execution of PoRP solely enables one to detect a cheating server or regular data loss with overwhelming probability. Thus, in case PoRP returns `reject` we know that the outsourced file is *not* retrievable (with overwhelming probability) and that at least one block is missing or damaged. Usually, the literature does not further investigate this case and it seems to be a common agreement that the client is supposed to blame the provider and also to initiate countermeasures in order to secure the remaining data by typically downloading all remaining file parts. In practice, however, the provider claims that he is not the one to blame since the SLA never guarantees 100% reliable storage and hence the missing block(s) are part of the (potential) expected regular data loss, yielding corrupted files. The countermeasure of downloading the file is not a very satisfying solution since the client is now in the position of losing the initial advantage of outsourcing her files and is required to invest her own precious resources to get the files. Furthermore, she does not know whether a large erasure (i.e., more data than the amount of parity data encoded into the file got deleted) or a small erasure (i.e., at most the amount of parity data encoded into the file got deleted) occurred. Hence, putting it all together we observe that a negative PoRP answer does not provide us with any information about the retrievability or irretrievability of the *original* file. Thus, it is the client's goal to determine whether the *original* file is recoverable *without* downloading it. To achieve this goal, we need to ensure that we obtain the knowledge that at least a certain minimal amount of file blocks in the processed file (at least as many blocks as the original file consists of) is valid. In order to sample this minimum amount of blocks, we require to perform multiple audits over the file. If this is successful, then, by the properties of the ECC decoding procedure, we are able to recover the original file. Otherwise the file is irrecoverable.

To the best of our knowledge, we are the first to investigate a solution towards ensuring recoverability of a file in the single-server setting after a PoR scheme returns a negative reply. In Section 5, we discuss in detail our approach and solution towards ensuring recoverability of the original file and thus close an important gap within the PoR functionality.[6] We also propose a solution which is applicable in the multi file case.

[6] Note that we provide a different solution towards ensuring recoverability than Bowers et al. [12] who aim to restore a corrupted file after file tampering has been detected.

3.5 On Erasure-Correcting Codes

Our CSPoR solution follows previous PoR proposals and also utilizes a *forward erasure-correction code* (ECC). On the one hand, ECCs are mainly used to increase the probability of detecting a malicious provider while, on the other hand, they enable one to reconstruct the original data from *any* part of the encoded data even if an error, i.e. erasure, has been introduced, as long as this part is sufficiently large. For example, using Reed-Solomon (RS) codes as in SW-PoR, a fraction of the ECC coding rate ρ of the outsourced file blocks $\tilde{n}$ is enough in order to decode them to the original file blocks n. Note that without ECC already a small bit degradation renders a file to be irretrievable even if 99% of the file is truthfully stored and thus may satisfy the SLA. Recall that in the single-file case of classical PoR schemes, one has generally two options to encode the ECC into the file. One possibility is to divide a file into chunks and apply the ECC to each chunk and another possibility is to encode the whole file as a single dataword. The first proposal is insecure since all chunks are independently encoded of each other and thus deleting a huge part of a single chunk with respective parity data makes this chunk absolutely irrecoverable. According to our security model, cf. Section 3.2, an adversary is able to apply adversarial erasure from bits up to nearly every size. In order to prevent adversarial erasure in practice one would use variants of a RS code. Unfortunately RS has an encoding and decoding time being quadratic in the number of symbols s, i.e., $\mathcal{O}(s^2)$. Recent research has shown that this can be reduced to $\mathcal{O}(s\log(s))$ [25]. Since each symbol is involved to generate parity data, a linear-time encoding and decoding is desired, i.e., $\mathcal{O}(s)$. However, the second proposal has an efficiency drawback because to the best of our knowledge, there is no linear-time ECC which is secure against adversarial erasure. Thus, both encodings are not suitable for our requirements.

In our context, we also aim to maximize data output and storage space and thus employ a *systematic* code (actual data is separated from the parity data), which is at the same time *maximum distance separable*, i.e. an optimal code. Coming back to RS codes, it is known that they meet those properties for any size of code and data words. However, to overcome the performance drawback of RS mentioned earlier and also accommodating the multi-file case, we follow [7] and scramble the generated parity data, i.e., we permute and encrypt the parity data block-wise for each file under a specific key for this file. To encrypt each block independently a tweakable block cipher may be used [26], and for constructing large permutations one may follow [23]. In more detail, the original file is split in chunks of a fixed size which can be processed very fast by the employed RS code, for example Cauchy-RS $(255, 223)$ over $\mathrm{GF}(256)$. Then, all code words are separated in data words and parity words in such a way that at the beginning of the file all data words are stored sorted followed by the according parity words in the same sorting. Next, the parity words are treated as one long word which gets divided in fractions of same size and then permuted. Again, the result is treated as one single word which gets split in words of a certain length to be encrypted. To summarize this, an original file F, consisting of n blocks, gets processed with the ECC of code rate ρ to a file $\mathcal{F} = F\|P$, consisting of $\tilde{n} = n/\rho$ blocks, where

P represents the scrambled parity data. Observe that the adversary may still delete a data word and its constrained parity data. However, with increasing file size the probability of erasure is negligible, i.e. the adversary can at most delete randomly. Obviously, if the file size is very small, the chances of the adversary to erase two constrained words is not negligible. [7] Then we advise to use RS in a mode where the whole file is a single data word.

In order to overcome the restrictions of the non-linear-time effort of RS, splitting the file as described will decrease the running time immensely. However, if that is not enough, one may employ *near optimal erasure codes* (NOEC) which are linear-time, an overview is given in [28]. A well-known example of this class of erasure codes are *low-density parity-check codes* (LDPC) [25]. Note that they do not protect against adversarial erasure. To solve this, the previous scrambling method may be utilized. That is, the adversary cannot do better than randomly delete data since the relation between the original data and parity data is unknown to him and thus the adversarial erasure corresponds to random erasure.

While for decoding, RS needs any n of $\tilde{n}$ blocks, NOEC requires any $n(1+\epsilon)$ blocks, $\epsilon > 0$. In other words, original data can be recovered by the decoding procedure if at most $(1-(1-\epsilon)\rho)$ of the processed data has been deleted.[8] We do not apply an ECC over multiple files at the same time because of the drawbacks mentioned in Section 3. As a final note, the stronger the robustness of the ECC, the less replicas are needed on the side of the CSP.

4 Cloud Storage Proofs of Retrievability

In this section we introduce our *new* PoR notion called *Cloud Storage Proofs of Retrievability* (CSPoR) scheme which is a natural generalization of "classical" PoR systems, overcoming the previously discussed problems. Furthermore, we briefly present the appropriate security model and provide details about the concrete instantiation.

4.1 Formal Definition

In this section we present a formal definition of CSPoR. Prior to this, let us briefly introduce the notion of a *cloud storage* which acts as the underlying abstract model of data storage in which digital data can be stored. We denote the cloud storage by $\mathfrak{S}$ and assume it can store multiple arbitrary files $F \in \{0,1\}^*$.

Definition 1. *A* cloud storage proofs of retrievability (CSPoR) scheme CSPoR *comprises the following procedures:*

[7] For example, if $n = 2$ and $\tilde{n} = 4$, then there is a chance of 33% of deleting constrained blocks when erasing two blocks randomly.

[8] One can think of $(1-(1-\epsilon)\rho)$ as an abbreviation for $(1-(1-\epsilon)\rho)\cdot 100$ percent, for RS $\epsilon = 0$.

$(pk, sk, \mathfrak{S}) \xleftarrow{\$} \mathsf{CSPoRSetup}(1^\lambda)$**:** *this randomized algorithm generates a public-private key pair* (pk, sk) *and takes as input the security parameter* λ. *It initializes a cloud storage* $\mathfrak{S}$*;*

$(\widehat{\mathcal{F}}, \widehat{\tau}, \mathfrak{S}) \xleftarrow{\$} \mathsf{CSPoRStore}(sk, \widehat{F})$**:** *this randomized data storing algorithm takes as input a secret key sk and the set of all files* $\widehat{F}$ *a client wishes to store at the provider's cloud storage. The set of files consists of* $K \in \mathbb{N}$ *files where* $\widehat{F} := \{F^{(k)} \mid F^{(k)} \in \{0,1\}^*, k \in [K]\}$. *Each file within the cloud storage gets processed yielding the set of all processed files* $\widehat{\mathcal{F}} := \{\mathcal{F}^{(k)} \mid k \in [K]\}$ *and a respective set of file tags is generated* $\widehat{\tau} := \{\tau^{(k)} \mid k \in [K]\}$ *where each tag contains additional information (e.g. meta data) about the processed file. Furthermore, the algorithm outputs the updated cloud storage* $\mathfrak{S}$*;*

$\delta \xleftarrow{\$} \left[\mathsf{CSPoRVerify}(pk, sk, \widehat{\tau}') \rightleftharpoons \mathsf{CSPoRProve}(pk, \widehat{\mathcal{F}}', \widehat{\tau}')\right]$**:** *this challenge-response protocol defines a protocol for proving cloud storage retrievability. The prover algorithm takes as input the public key pk, the file tag set* $\widehat{\tau}' := \{\tau^{(k)} \mid k \in [K']\}$ *and the set of the processed files* $\widehat{\mathcal{F}}' := \{\mathcal{F}^{(k)} \mid k \in [K']\}$, *where* $[K'] \subseteq [K]$. *The verification algorithm uses as input the key pair* (pk, sk) *and the file tag set* $\widehat{\tau}'$. *Algorithm* $\mathsf{CSPoRVerify}$ *finally outputs a binary value* δ *which equals* `accept` *if verification succeeds, indicating the files* $\widehat{\mathcal{F}}'$ *are being stored and retrievable from the cloud storage provider, and* `reject` *otherwise.*

Note that $\widehat{\mathcal{F}}$ may not be exactly equal to $\widehat{F}$ but it must be guaranteed that $\widehat{F}$ can be recovered from $\widehat{\mathcal{F}}$. We wish to remark that the involved file tag set $\widehat{\tau}'$ in the challenge-response protocol can correspond to either the full set of file tags $\widehat{\tau}$ or any arbitrary subset of file tags enabling a CSPoR scheme to flexibly check any set of files by specifying the appropriate tags. Informally, a CSPoR scheme is *correct* if all processed files $\widehat{\mathcal{F}}$ outputted by the store procedure $\mathsf{CSPoRStore}$ will be accepted by the verification algorithm when interacting with a valid prover. More formally this is captured as follows.

Definition 2. *A CSPoR protocol is* correct *if there exists a negligible function* negl *such that for every security parameter* λ, *every key pair* (pk, sk) *generated by* $\mathsf{CSPoRSetup}$, *for all sets of files* $\widehat{F}$ *(containing files* $F^{(k)} \in \{0,1\}^*$*), and for all* $(\widehat{\mathcal{F}}, \widehat{\tau}, \mathfrak{S})$ *generated by* $\mathsf{CSPoRStore}$, *it holds that*

$$\Pr\left[\left(\mathsf{CSPoRVerify}(pk, sk, \widehat{\tau}') \rightleftharpoons \mathsf{CSPoRProve}(pk, \widehat{\mathcal{F}}', \widehat{\tau}')\right) \not\twoheadrightarrow \texttt{accept}\right] = \mathrm{negl}(\lambda).$$

Furthermore, we denote the above challenge-response procedure of CSPoR by $\mathsf{CSPoRP}(pk, sk, \widehat{\tau}', \widehat{\mathcal{F}}')$. If the involved files and tags are clear from the context, we simply abbreviate it as CSPoRP which we also refer to as an *audit* throughout the paper.

Remark 1 (Cloud Storage and Storage Container). Recall that we denote by cloud storage an abstract model of data storage in which one stores digital data. However, moving towards realizing a cloud storage architecture, we can introduce another storage unit called a *storage container*. Such a storage unit allows for storing multiple files within one location (in the physical layer of the cloud environment), providing a client with a file system structure. Note that similar concepts are already in practical use called *Buckets* [2, 20] or *Blobs* [27]. Usually, a storage container is limited by a pre-defined storage space size. Hence, a client may create and handle multiple storage containers simultaneously. Ultimately, we call the set of all storage containers a *cloud storage*. In Appendix A.1 we present an extended definition of CSPoR which takes into account multiple cloud containers and the respective instantiation can be found in Appendix A.2.

4.2 Security Model

In this section we discuss the security model within our proposed model. Note that we do not explicitly consider *confidentiality* of a file F, but assume that a client may encrypt the files before the initiation of the CSPoR protocol. The adversary aims to convince a client with overwhelming probability that the outsourced files are still fully intact and retrievable. In the following we define the security notion of *extractability* for our CSPoR scheme following existing security notions for PoR models.

Extractability Intuitively, we wish to formalize and say that a CSPoR protocol is secure if any cheating prover that convinces the verification algorithm to accept is indeed storing all files in $\widehat{\mathcal{F}}$ with a sufficient level of probability. In other words, we wish to guarantee that, whenever a malicious prover is in a position of successfully passing a CSPoRP instance, it must *know* the entire file content of all files. We require an extractor algorithm $\mathcal{E}(pk, sk, \widehat{\tau}, \mathcal{P}')$ taking as input the generated key pair, the set of file tag $\widehat{\tau}$ as well as a description of the machine implementing the prover's role in the CSPoR protocol. The extractor's output is the set of files $\widehat{F}$. As noted above, the extractor is given (non black-box) access to $\mathcal{P}'$ and in particular can rewind it. Furthermore, we require that the algorithm is efficient, i.e. $\mathcal{E}$'s running time needs to be polynomial in the security parameter.

Consider the following *extractability* game $\mathbf{Exp}_{\mathcal{A},\epsilon}^{\text{EXTRACT}}\left[\mathcal{CSPOR}, 1^{\lambda}\right]$ between a malicious adversary $\mathcal{A}$, an extractor $\mathcal{E}$, and a challenger $\mathcal{C}$.

1. The challenger initializes the system by running CSPoRSetup to generate the public and private key pairs. The public keys are provided to $\mathcal{A}$. It also generates a cloud storage $\mathfrak{S}$ which is also given to the adversary $\mathcal{A}$.
2. The adversary $\mathcal{A}$ is now able to interact with the challenger that takes the role of an honest client. $\mathcal{A}$ is allowed to request executions to a CSPoRStore oracle by providing, for each query, a set of files $\widehat{F} := \{F^{(k)} \mid F^{(k)} \in \{0,1\}^*, k \in [K]\}$.
3. Likewise, $\mathcal{A}$ can request executions of the CSPoRP procedure for any set of files on which it previously made a CSPoRStore query by specifying the corresponding tags $\widehat{\tau}$. In the procedures, the challenger will play the role of the honest verifier $\mathcal{V}$ and the adversary the role of the corrupted prover, i.e. $\mathcal{V}(pk, sk, \widehat{\tau}) \rightleftharpoons \mathcal{A}$. In the end of the execution the adversary is provided with

the output of the verifier. Furthermore, the CSPoRStore oracle queries and the executions of CSPoRP can be interleaved arbitrarily.

4. Finally, the adversary outputs a set of challenge tags $\widehat{\tau}'$ returned from some CSPoRStore query and the description of a prover $\mathcal{P}'$.
5. Run the extractor algorithm $\widehat{F}' \leftarrow \mathcal{E}(pk, sk, \widehat{\tau}', \mathcal{P}')$ inputting the challenge tags $\widehat{\tau}'$ and description $\mathcal{P}'$ where $\mathcal{E}$ gets black-box rewinding access to $\mathcal{P}'$, and attempts to extract the file content of all files as $\widehat{F}'$.
6. If $\Pr\left[(\mathcal{V}(pk, sk, \widehat{\tau}) \rightleftharpoons \mathcal{P}') \rightarrow \texttt{accept}\right] \geq \epsilon$ and $\widehat{F}' \neq \widehat{F}$ then output 1, else 0.

Note that we say a malicious prover $\mathcal{P}'$ is *ϵ-admissible* if the probability that it convincingly answers verification challenges is at least ϵ, i.e., if

$$\Pr\left[(\mathcal{V}(pk, sk, \widehat{\tau}) \rightleftharpoons \mathcal{P}') \rightarrow \texttt{accept}\right] \geq \epsilon.$$

Here the probability is over the coins of the verifier and prover.

Definition 3. *We say that a $\mathcal{CSPOR}$ scheme is ϵ-*extractable *(or* secure*) if there exists an efficient extraction algorithm $\mathcal{E}$ such that, for all PPT adversaries $\mathcal{A}$ it holds that*

$$\Pr\left[\mathbf{Exp}_{\mathcal{A},\epsilon}^{\text{EXTRACT}}\left[\mathcal{CSPOR}, 1^{\lambda}\right] \rightarrow 1\right]$$

is negligible in the security parameter.

4.3 Instantiation Details

Our concrete instantiation is based on the private PoR scheme of Shacham and Waters (SW-PoR) [32] mainly due to its ability to handle an unbounded number of verification queries in a compact way.[9] In case a better communication complexity is required, one may build upon the schemes presented in [13, 37]. On a high level, our instantiation exploits the homomorphic properties of the SW-PoR proposal enabling us to aggregate a proof for *all* files into a small value. Our CSPoR instantiation overcomes the identified limitations, as discussed in Section 3.3, when employing existing schemes straightforwardly to prove retrievability for multiple different-sized files simultaneously. After outlining our main building blocks, we provide details about our instantiation.

Building Blocks Unless otherwise specified all operations are performed over the finite field $\mathbb{F} = \mathbb{Z}_p$ where p is a λ-bit prime with λ being the security parameter. As we instantiate a private CSPoR system, it suffices to use a symmetric encryption scheme and we set the public key $pk = \perp$. We utilize a MAC scheme and a pseudo-random function (PRF) $g\colon \{0,1\}^* \times \{0,1\}^{\phi_{\text{prf}}} \to \mathbb{F}$, where ϕ_{prf} is the key length of the PRF. and a MAC scheme. Furthermore, we make use of a cloud storage $\mathfrak{S}$, cf. Section 4.1, which contains all outsourced data.

[9] For example, the scheme of [24] or a MAC based PoR are either bound or inefficient in communication and computation.

Specification of the $\mathsf{CSPoRSetup}$ Procedure In the $\mathsf{CSPoRSetup}$ procedure, the client derives a random symmetric key $\kappa_{\text{enc}} \xleftarrow{\$} \mathcal{K}_{\text{enc}}$ and a random MAC key $\kappa_{\text{mac}} \xleftarrow{\$} \mathcal{K}_{\text{mac}}$, where $\mathcal{K}_{\text{enc}}$ and $\mathcal{K}_{\text{mac}}$ are the respective key spaces. The secret key is $sk = (\kappa_{\text{enc}}, \kappa_{\text{mac}})$ and requests create a cloud storage $\mathfrak{S}$.

Specification of the $\mathsf{CSPoRStore}$ Procedure The $\mathsf{CSPoRStore}$ procedure is initiated by the client holding a set of $K \in \mathbb{N}$ files where $\widehat{F} := \{F^{(k)} \mid F^{(k)} \in \{0,1\}^*, k \in [K]\}$ that she wishes to store in $\mathfrak{S}$. The following steps are carried out for each file $F^{(k)}$ of $\widehat{F}$:

1. First, we apply an information dispersal algorithm (i.e. an erasure code, e.g. a systematic MDC ECC like permuted and encrypted Reed-Solomon code [7]) with code rate ρ over the file $F^{(k)}$ which originally consists of $n^{(k)} \in \mathbb{N}$ blocks. The resulting processed file is denoted by $\mathcal{F}^{(k)}$;
2. Next, we divide the processed file $\mathcal{F}^{(k)}$ into $\tilde{n}^{(k)} \in \mathbb{N}$ blocks, each block being s symbols long. That is $\mathcal{F}^{(k)} = \{f_{ij}^{(k)}\}$, where $1 \leq i \leq \tilde{n}^{(k)}$, $1 \leq j \leq s$, and $f_{ij}^{(k)} \in \mathbb{F}$. Note that s is *constant* for all files while the number of blocks $\tilde{n}^{(k)}$ varies depending on the respective underlying original file size;
3. We sample uniformly at random a PRF key $\kappa_{\text{prf}}^{(k)} \xleftarrow{\$} \{0,1\}^{\phi_{\text{prf}}}$ and sample s random elements from the finite field $\mathbb{F}$ which are kept private by the client, that is $\alpha_1^{(k)}, \ldots, \alpha_s^{(k)} \xleftarrow{\$} \mathbb{F}$;
4. Then, we compute for each file block of $\mathcal{F}^{(k)}$ an authentication tag $\sigma_i^{(k)}$, $i \in [\tilde{n}]$, as follows
$$\sigma_i^{(k)} \longleftarrow g_{\kappa_{\text{prf}}^{(k)}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} f_{ij}^{(k)} \qquad \in \mathbb{F};$$
5. At last, a file tag $\tau^{(k)} := \tau_0^{(k)} \| \mathsf{MAC}_{\kappa_{\text{mac}}}(\tau_0^{(k)})$ is computed, where $\tau_0^{(k)} := \tilde{n}^{(k)} \| \mathsf{Encrypt}_{\kappa_{\text{enc}}}\left(\kappa_{\text{prf}}^{(k)} \| \alpha_1^{(k)} \| \ldots \| \alpha_s^{(k)}\right)$.

Finally, the client combines all authentication tags into the set $\widehat{\sigma}$, all file tags into the set $\widehat{\tau}$, as well as all processed files are aggregated as the set $\widehat{\mathcal{F}}$. The three sets are uploaded to the cloud storage $\mathfrak{S}$ of the provider while $\widehat{\sigma}$ and $\widehat{\mathcal{F}}$ are removed locally from the client ($\widehat{\tau}$ is optional). Note that in the fifth step the provider only learns the size of the outsourced file, since the remaining part of $\tau^{(k)}$ is encrypted with the client's secret key.

Specification of the CSPoRP Procedure The CSPoRP procedure obtains an assurance about the retrievability of the files. In the following we describe the technical details of an audit step providing a reply δ. Note that the client may wish to audit only a subset K' of all K outsourced files, hence we have $[K'] \subseteq [K]$.

1. The client first verifies the MAC on each $\tau^{(k)}$ within $\widehat{\tau}$. If the MAC is invalid the client aborts the protocol and outputs `reject`. Otherwise, she parses all $\tau^{(k)}$ from $\widehat{\tau}$ and uses κ_{enc} in order to recover $\tilde{n}^{(k)}$, $\kappa_{\text{prf}}^{(k)}$ and $\alpha_1^{(k)}, \ldots, \alpha_s^{(k)}$ for all $k \in [K']$;
2. Next the client selects a random subset $I^{(k)} \subseteq_{\$} [\tilde{n}^{(k)}]$ of size $\ell^{(k)}$ and chooses for each $i \in I^{(k)}$ a random element from the finite field $\nu_i^{(k)} \xleftarrow{\$} \mathbb{F}$ for all $k \in [K']$;
3. Then the client generates the challenge by aggregating the sampled values from Step (2) per file to a set $Q^{(k)} = \{(i, \nu_i^{(k)})_{i \in I^{(k)}}\}$ of size $\ell^{(k)}$, for all $k \in [K']$. All sets $Q^{(k)}$ are combined to $\widehat{Q} := \{Q^{(k)} \mid k \in [K']\}$ which is then sent to the provider.

The cloud service provider now parses all files from $\widehat{\mathcal{F}}$ as $\{f_{ij}^{(k)}\}$ and $\{\sigma_i^{(k)}\}$, and the corresponding challenges $Q^{(k)}$ from $\widehat{Q}$. Then, the provider computes for $1 \leq j \leq s$ and all $k \in [K']$

$$\mu_j^{(k)} \longleftarrow \sum_{\left(i, \nu_i^{(k)}\right) \in Q^{(k)}} \nu_i^{(k)} f_{ij}^{(k)}, \qquad \sigma^{(k)} \longleftarrow \sum_{\left(i, \nu_i^{(k)}\right) \in Q^{(k)}} \nu_i^{(k)} \sigma_i^{(k)}.$$

Next, the CSP accumulates all responses and authentication tags to output for each $1 \leq j \leq s$

$$\widetilde{\mu}_j := \sum_{k \in [K']} \mu_j^{(k)} \qquad \text{and} \qquad \widetilde{\sigma} := \sum_{k \in [K']} \sigma^{(k)}.$$

Finally, the client parses the provider's response and checks

$$\widetilde{\sigma} \stackrel{?}{=} \sum_{k \in [K']} \left(\sum_{\left(i, \nu_i^{(k)}\right) \in Q^{(k)}} \nu_i^{(k)} g_{\kappa_{\text{prf}}^{(k)}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} \widetilde{\mu}_j \right).$$

If this equality check is successful, the verifier outputs $\delta = $ `accept`, and otherwise she outputs $\delta = $ `reject`. Note that it is easy to check the correctness for the above instantiation. A formal treatment can be found in Appendix A.2.

Now we formulate our theorem capturing the security of our CSPoR scheme.

Theorem 1. *If the MAC scheme is unforgeable, the symmetric encryption scheme is semantically secure, and the PRF is secure, then no adversary (except with negligible probability) against the extractability game* EXTRACT *of our CSPoR scheme ever causes the verifier to accept a cloud storage proofs of retrievability protocol instance, except by responding with correctly computed responses* $\widetilde{\mu}_j$ $(1 \leq j \leq s)$ *and authentication tag* $\widetilde{\sigma}$.

Proof. The security of our proposed scheme follows immediately from the security of the private-key SW scheme [32] as we base our construction on their scheme and thus inherit the security property. The only main difference in the proof is that we have to ensure the correctness of the aggregated set of responses $\{\widetilde{\mu}_j\}$.

First, we need to argue that the verification algorithm will reject answers except if the answers $\{\widetilde{\mu}_j\}$ were computed correctly by the prover. This can be shown via a sequence of games for which we argue that the adversary's distinguishing advantage between two consecutive games is negligible. The analysis follows via the same game hops as in [32] while adapting the notion of having aggregated answers $\{\widetilde{\mu}_j\}$ from all outsourced files.

Secondly, we need to argue that the extraction procedure can efficiently reconstruct a ρ fraction of file blocks when interacting with a prover that provides correctly computed responses for a non-negligible fraction of the query space. The same arguments as in [32] apply to our scheme. Here we only need to slightly change the proof details when showing that a well-behaved ϵ-admissible cheating prover $\mathcal{P}'$ as the output of the extractability game (cf. Section 4.2) can be turned into an ϵ-polite adversary $\mathcal{B}$ (implemented as a probabilistic polynomial-time Turing machine). Note that we say an adversary is ϵ-polite if it responds with probability ϵ to given queries Q covering an ϵ fraction of the query- and randomness-tape space. $\mathcal{P}'$ can be used to construct the adversary $\mathcal{B}$. For a query Q, $\mathcal{P}'$ interacts with the verifier according to $\mathcal{V}(pk, sk, \widehat{\tau}) \rightleftharpoons \mathcal{P}'$. If the interaction is successful the responses $(\widetilde{\mu}_1, \ldots, \widetilde{\mu}_s)$ will be written to its output tape while a wrong interaction leads to writing $\perp$ on the tape. Note that after k interactions we can represent all responses as a $(k \times s)$ matrix. Each time the adversary $\mathcal{B}$ runs the prover $\mathcal{P}'$ it is able to effectively rewind the prover. Since $\mathcal{P}'$ is well-behaved a successful interaction computes valid $(\widetilde{\mu}_1, \ldots, \widetilde{\mu}_s)$ and given that $\mathcal{P}'$ is ϵ-admissible we know that an ϵ fraction of the answers are computed correctly. Having this we can further follow SW and can represent the extractor's knowledge by a row in the matrix for each audit step, i.e. as $(\widetilde{\mu}_1^{(t)}, \ldots, \widetilde{\mu}_s^{(t)})$ where $t \in [A]$, and thus have sampled enough information to permit successful extraction.

Lastly, we argue that following the ECC reconstruction property it suffices to positively check any n blocks of a processed file consisting of $\tilde{n}$ blocks to recover the original file $F^{(k)}$ with all but negligible probability. This is trivially fulfilled by employing Reed-Solomon codes of rate ρ, since any ρ fraction of encoded file blocks suffice in order to reconstruct the underlying file. Note, however, that this does not protect a client from revealing correlations between the plaintext blocks and redundant blocks through the access pattern. This can be avoided if a client encrypts and permutes the parity part of the file following Ateniese et al. [7]. $\square$

Remark 2 (Applicability of CSPoR to Current Cloud Architectures). The above introduced CSPoR system can be translated straightforwardly into present cloud architectures. This can be achieved by introducing procedures (e.g., CSPoRStore) that capture the communication steps between a client and a storage provider. Let us assume that a provider exposes a standard interface to its client offering a handful of commands in order to execute some basic operations such as storing or downloading a file, as well as other commands. To implement such an interface for our CSPoR system, we can use currently employed APIs from

Amazon [2], Google [19] or Microsoft [27]. Following those APIs, it suffices to use only two commands to implement the above procedures for a CSPoR system in current cloud architectures, namely `POST` and `GET`. Note that all formal details and discussions can be found in Appendix A.3.

5 Determining File Recoverability

As mentioned in the previous sections, PoR schemes detect with an overwhelming probability whether data loss has occurred within a single audit. Since data may be lost without violating the mutually agreed SLA, the CSPoR scheme will output $\delta = \texttt{reject}$, although the original files may still be retrievable. Thus, in summary, we can check multiple files with CSPoR simultaneously. If the scheme returns `accept` this indicates that all files are retrievable with overwhelming probability, while in contrast we only know that at least one block of some file is corrupted if CSPoR returns `reject`. Recall that the literature has not further considered a solution towards forming a provable statement about the retrievability of the original file in case the scheme returns a rejection token.[10] In the following, we provide a solution to close this gap.

Let us assume that CSPoR returns a rejection token. In the following let us redefine CSPoRP to take as input a single processed file $\mathcal{F}$, a single file tag τ, and the challenged block identifiers I, which we abbreviate by CSPoRP', i.e.

$$\mathsf{CSPoRP}'(I, \tau, \mathcal{F}) := [\mathsf{CSPoRVerify}(I, \tau) \rightleftharpoons \mathsf{CSPoRProve}(\mathcal{F}, \tau)].$$

Next we introduce a new algorithm called *Proofs of Recoverability* (PoRec), see Algorithm 1, which is initiated by the verifier C and involves the provider S. It takes as input the ECC code rate ρ, a file tag τ of the outsourced file $\mathcal{F}$ from C, and S inputs the outsourced file $\mathcal{F}$. At the end, the algorithm outputs `accept` if and only if the original file F is recoverable from $\mathcal{F}$, otherwise `reject`. Line 1 represents the extraction of $\tilde{n}$ from τ, Line 3 follows from Theorem 2, Line 8 denotes a random sampling of ℓ disjunctive elements of the set $[\tilde{n}] \setminus S$, and Lines 13 and 14 follow from Theorem 3. Note that a non-random sampling would give the attacker information about the verifier's query pattern and hence may enable him to predict her behavior to determine specific parts of the file which are usually seldomly checked and thus motivates the attacker to delete them.

We stress again that CSPoR is used to detect corruptions, and PoRec determines if an original file is fully recoverable from the respective damaged outsourced file. Combining both allows us to prove if *all* original files are fully recoverable, see Section 5.3.

5.1 Challenge Size

The situation we consider is that some data loss has occurred in the outsourced file $\mathcal{F}$ which results in a `reject` using CSPoR. Recall that in the procedure

[10] As discussed in Section 3.4, it seems that the literature assumes that in case a rejection token is returned that one downloads all remaining parts of the file independent of the actual degree of data loss.

Algorithm 1: PoRec (Proofs of Recoverability)

```
   Input: C: Filetag $\tau$, ECC code rate $\rho$; S: processed file $\mathcal{F}$
   Output: accept if original $F$ is recoverable, else reject
1  $\tilde{n} \hookleftarrow \tau$                                    // extract number of blocks of $\mathcal{F}$
2  $n \longleftarrow \tilde{n}\rho$                                          // number of blocks of $F$
3  $\ell \longleftarrow 1$                                                  // Theorem 2
4  $a \longleftarrow 0$                                               // number of accepts
5  $r \longleftarrow 0$                                               // number of rejects
6  $S \longleftarrow \emptyset$                              // set of previously challenged block ids
7  for $A \leftarrow 1$ to $\tilde{n}$ do
8      $I \overset{\$!\{\ell\}}{\longleftarrow} [\tilde{n}] \setminus S$
9      $S \longleftarrow S \cup I$
10     $\delta' \longleftarrow$ CSPoRP'$(I, \tau, \mathcal{F})$
11     if $\delta' =$ accept then $a \leftarrow a + 1$
12     else $r \longleftarrow r + 1$
13     if $a = n$ then return accept                                 // Theorem 3
14     if $r = \tilde{n} - n + 1$ then return reject                   // Theorem 3
15 return reject
```

CSPoRVerify the challenge size ℓ is usually chosen conservatively, i.e. $\ell = \lambda$. To obtain an assurance that F is fully recoverable from its respective damaged outsourced file, we need to prove that there exist at least any n valid blocks out of $\tilde{n}$ blocks in the outsourced file, and hence enables us to recover the original file by using the ECC decoding procedure. In Theorem 2 we show that $\ell = 1$ enables us to learn whether a certain block is valid.

Theorem 2 (Challenge Size). *Let $0 < \rho \leq 1$ be the ECC code rate, $|\mathcal{F}| = \tilde{n}$, $|F| = n = \tilde{n}\rho$, and let $1 \leq \ell \leq \tilde{n}$ be the challenge size of each audit $A \in \mathbb{N}$. Assume that at least one of the blocks of $\mathcal{F}$ is damaged. To ensure that at least any valid n blocks are contained in $\mathcal{F}$ using the* CSPoRP*' algorithm, it must hold $\ell = 1$.*

Proof. Let $\ell \in \mathbb{N}_0$. Obviously $\ell < 1$ results in no challenge at all and thus we can ignore this case. If $\ell > 1$, then CSPoRP$'$ likely returns reject since the detection probability of finding a damaged block is overwhelming. However, this does not provide any information on how many blocks are in fact damaged. At this point, we only know that at least one block is damaged but at most ℓ. Of course, there is a probability to hit the non-damaged blocks with $\ell > 1$, however it gets very small depending on the degree of erasure. In other words, CSPoRP$'$ may return reject for $\ell > 1$, even if the original data could indeed be recovered due to the ECC. Therefore, we explicitly require to know if any n valid blocks are contained in $\mathcal{F}$, and thus need to determine this number precisely. Hence, $\ell = 1$, which allows us to count the non-damaged blocks in a precise manner. □

In terms of CSPoRP$'$, this means that the challenge set $\widehat{Q}$ consists only of a single block identifier and coefficient.

5.2 Number of Audits

In order to count the number of valid blocks, we need to know how often CSPoRP' needs to be performed, i.e. the number of audits A. Theorem 3 gives lower and upper bounds for A.

Theorem 3 (Audit Bounds). *Let ρ, $\tilde{n}$, n be defined as in Theorem 2, let $\ell = 1$, and assume that at least one of the blocks of $\mathcal{F}$ is damaged. Then $\min(n, \tilde{n} - n + 1) \leq A \leq \tilde{n}$ disjunctive audits are required for the* PoRec *algorithm to output either* `accept` *or* `reject`*.*

Proof. The number of audits required is lower bounded by the minimum of two values. First, after any n disjunctive audits have yielded `accept`, the PoRec procedure returns `accept`. The other lower bound is fulfilled when the ECC decoding is not able to reconstruct F out of the remaining blocks of $\mathcal{F}$. That is, if any $\tilde{n} - n + 1$ audits resulted in `reject`, the PoRec algorithm aborts and outputs `reject`. The upper bound is reached if $n - 1$ audits resulted in `accept` but the last undamaged block may be on the last remaining unchecked position. Depending on the retrievability of the final block, the PoRec procedure returns `accept` or `reject`. □

Finally, the verifier performs A times CSPoRP' accumulating the number of valid and invalid responses. Since the randomly sampled block identifiers are disjunct, the verifier performs audits for as long as it takes until she is convinced that the number of valid (`accept`) or invalid (`reject`) responses is sufficiently large. The algorithm PoRec defines this formally and finally either outputs `accept` or `reject` meaning that the file F is recoverable from $\mathcal{F}$ or not, respectively.

Remark 3 (SW-PoR Scheme and Recoverability). Note that the SW-PoR scheme on which our CSPoR scheme CSPoR builts upon yields no recoverability guarantees for a large challenge size ℓ, e.g. $\ell = n$. This holds since the detection probability is overwhelming even if only one of the ℓ challenged blocks of $\mathcal{F}$ is damaged resulting in the SW-PoR scheme outputting a rejection token, and hence, we cannot determine whether the original file can be recovered. Also, if we perform the SW-PoR scheme A times with $\ell = 1$, this will not provide an assurance about the recoverability, since the challenges are chosen randomly and hence are not disjunct with high probability. Therefore, this results in likely challenging too few blocks or inefficiency.

5.3 Locating Damaged Files

As described in both preceding Sections 5.1 and 5.2, the output of PoRec determines whether a file F is recoverable from the remaining parts of $\mathcal{F}$. Now we apply this to the multi-file case with the goal to convince the verifier that any n out of $\tilde{n}$ blocks for each file are still valid which enables us to argue that all files are recoverable. In other words, we combine CSPoR and PoRec.

Employing a File Tree. The verifier usually performs the CSPoRP procedure of the CSPoR scheme over all stored files $\widehat{F}$ with $\ell = \lambda$ for each file (except for $\tilde{n} < \lambda$, then $\ell = \tilde{n}$). Each time CSPoRP outputs `accept`, the verifier knows that the probability of some fraction of the data being damaged is negligible. As a result, *all* original files $\widehat{F}$ can be recovered. However, if one CSPoRP results in a `reject`, the verifier stops with the regular execution of CSPoR. Since this does not provide any information about the retrievability of the original files, the verifier organizes her files in a b-ary tree and performs a multiple b-ary search on this tree. The root of the tree represents the result of CSPoRP over all processed files $f := \widehat{\mathcal{F}}' \subseteq \widehat{\mathcal{F}}$, which the verifier wishes to check. Then, the first level of the tree consists of b nodes where each contains disjunctive filesets. For each of these nodes, the verifier again performs a CSPoRP with $\ell = \lambda$. If a CSPoRP returns `accept`, the verifier discards the node from his tree since all associated (outsourced) files are retrievable. Otherwise the node is split again into b nodes and for each node a CSPoRP is executed. This process is repeated until the set of files which a single CSPoRP execution checks contains only one file. At the end, the verifier gets a list of all processed files $f_c := \widehat{\mathcal{F}}'_c$ which are corrupted. Finally, for each file $\mathcal{F}_c$ of f_c, the verifier executes $\mathsf{PoRec}(\tau, \rho, \mathcal{F}_c)$. Now the verifier knows which files can be recovered and which are ultimately lost. Note that all files $f \setminus f_c$ are also obviously recoverable since they did not contain any corrupted blocks with overwhelming probability.

An example is shown in Figure 1 for values $b = 3$, $|f| = 4$, and $|f_c| = 2$. Traversing the tree yields the corrupted and sound files. The input of a CSPoRP ($\bigcirc$) consists of the set of all files to which the CSPoRP node is a parent node. The output of the procedure is displayed next to the respective node. Regarding PoRec ($\square$), the input consists of the outsourced file labeled below the box, and the output is displayed at the bottom of each PoRec. The fourth CSPoRP returned `accept`, hence all files belonging to this specific node do not need any further inspection and are immediately marked as accepted, i.e., retrievable. This is shown by the dashed lines between the accepted CSPoR and its leaves, as well as the omitted PoRec executions, which are not required to be executed since there is no output and the files are directly marked as sound ($\checkmark$).

Observe that in the worst case, all f files need to be checked which requires $1 + \sum_{i=1}^{\log_b(f)} b^i$ CSPoRP executions. In the best case of only a single file being damaged, a maximum of $1 + b \log_b(f)$ CSPoRP executions are required. Further steps, regarding repairing the damaged files, changing or taking legal actions against the cloud storage provider, is out of the scope of this work. An adaption of $\mathsf{CSPoR\text{-}PoRec}$ to the multiple cloud container case is presented in Appendix B.

Combining CSPoR and PoRec. Combining CSPoR and PoRec to a single procedure yields our final algorithm $\mathsf{CSPoR\text{-}PoRec}(\widehat{F}, \rho, \lambda)$, cf. Algorithm 2. The goal of the algorithm is to determine which files of the set $\widehat{F}$ are recoverable and which ones are irrecoverable, if an error occurs. After initially outsourcing the file set $\widehat{F}$ to the cloud storage provider, the algorithm $\mathsf{CSPoR\text{-}PoRec}$ checks repeatedly the retrievability of the file set $\widehat{\mathcal{F}}$ (or any subset thereof) using CSPoRP. As

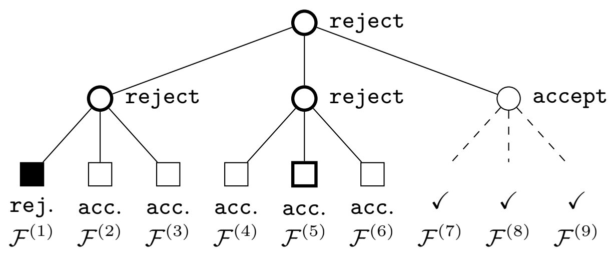


Fig. 1. Traversing the file tree spanned over nine different outsourced files. Four times CSPoRP is performed (◯) and six times PoRec (□). As a result $\mathcal{F}^{(1)}$ is damaged beyond repair (■), $\mathcal{F}^{(5)}$ is damaged but recoverable (▣), and all other files are sound and recoverable.

long as this check is successful, this leads to the conclusion that each file in this set is fully stored, hence the original files are retrievable and therefore obviously also recoverable. However, as soon as the procedure returns an error message, i.e. $\delta = \texttt{reject}$, we know that at least one and at most all files within the set $\widehat{\mathcal{F}}$ are not fully stored. Hence the algorithm creates a b-ary file tree to partition $\widehat{\mathcal{F}}$ into smaller sets checking whether those files are retrievable as described above. This is repeated until the algorithm finally returns the set of all corrupted files $\widehat{\mathcal{F}}_c'$, i.e. non-retrievable files. Therefore, the algorithm initializes two empty lists where we denote the list of recoverable files by A (accept) and the list of irrecoverable files by R (reject), see Line 10 in Algorithm 2. Then, the algorithm PoRec determines for each file of $\widehat{\mathcal{F}}_c'$ whether it is recoverable and adds the file tag either to the list A or R, respectively.

At the end the algorithm outputs the list of files which are corrupted but can still be recovered A (and repaired or updated) as well as the list of all files which are damaged beyond repair R. However, it is obvious that the list of recoverable files A also could contain all file tags of all retrievable files since they are clearly also recoverable. In the same way, one could also restrict the output to the non-recoverable files R only.

Remark 4 (Optimizations for the Verifier). First note that the verifier might want to check the retrievability of all files regularly. More precisely, she is able to run a *scheduled* CSPoRP routine, where each audit is planned for a certain time period and file set. This is represented in Algorithm 2 by Lines 3-5. Further, the verifier might optimize the way she performs CSPoRP and PoRec. For CSPoRP, depending on the ECC, the verifier might change the size of $\ell^{(k)}$, for some files $\mathcal{F}^{(k)}$, $k \in [K']$, in order to decrease the effort required for an audit. Regarding PoRec, the verifier might be already convinced if tn accept tokens are counted for a certain threshold $0 < t \leq 1$.

Algorithm 2: CSPoR-PoRec

```
Input: C: set of files $\widehat{F}$, ECC code rate $\rho$, security parameter $\lambda$
Output: List of recoverable ($A$) and non-recoverable ($R$) files
1  $(pk, sk, \mathfrak{S}) \longleftarrow \mathsf{CSPoRSetup}(1^\lambda)$                  // Definition 1
2  $(\widehat{\mathcal{F}}, \widehat{\tau}, \mathfrak{S}) \longleftarrow \mathsf{CSPoRStore}(sk, \widehat{F})$                  // Definition 1
3  repeat
4      $\delta \longleftarrow \mathsf{CSPoRP}(pk, sk, \widehat{\tau}', \mathfrak{S}, \widehat{\mathcal{F}}')$                  // Definition 1
5  until $\delta =$ reject
6  Create $b$-ary tree $T$ using $\widehat{\mathcal{F}}'$
7  repeat
8      Perform CSPoRP for all child nodes of rejected nodes of $T$
9  until tree traversed as needed, yielding $\widehat{\mathcal{F}}_c'$                  // Section 5.3
10 $A \longleftarrow \emptyset, R \longleftarrow \emptyset$
11 foreach $\mathcal{F}_c \in \widehat{\mathcal{F}}_c'$ do
12     $\delta \longleftarrow \mathsf{PoRec}(pk, sk, \tau, \rho, \mathcal{F}_c)$                  // Algorithm 1
13     if $\delta =$ accept then $A \longleftarrow A \cup \tau$
14     else $R \longleftarrow R \cup \tau$
15 return $(A, R)$
```

5.4 Efficiency Comparison

Let f denote the number of multiple different files being checked simultaneously. We can compare the storage and communication overhead of CSPoR to SW-PoR. Regarding storage, in SW-PoR-Setup the keys are file-dependent and thus require a storage amount of $2f\lambda$. CSPoR uses the same keys κ_{enc} and κ_{mac} for each file, requiring a file-independent storage amount of 2λ. The challenge phase in both SW-PoR and CSPoR demand the same communication overhead of $2\ell f\lambda$ from the client. However, the response in SW-PoR has a communication effort of $(s+1)f\lambda$ for the provider, while in CSPoR only a file-independent effort of $(s+1)\lambda$ is needed. Similarly, the verification phase has a computation effort of f in SW-PoR, while having a constant computation effort of a single execution in CSPoR.

The algorithm PoRec is an even smaller version of the audit phase of CSPoR, hence requiring a constant minimal computation and communication effort. However, each execution of PoRec is repeated up to $\tilde{n}$ times for each file. Observe that downloading x disjunctive blocks is much larger in terms of computation, communication, and storage overhead than checking the recoverability of these x blocks using PoRec. Regarding computation, the ECC decoding procedure requires more information than a single block resulting in a lot of overhead due to downloading additional data and decoding all of it. In terms of communication, the actual block would need to be transferred instead of a single bit per block as in PoRec. Lastly, the client would need to store the whole downloaded blocks, however, in PoRec only about $\log(A) + S$ bits need to be stored per file. This is why PoRec is more efficient than downloading the blocks directly.

6 Conclusion

In this paper we have introduced two extensions to the traditional PoR concept which we call cloud storage proofs of retrievability (CSPoR) and proofs of recoverability (PoRec) as well as provide a combined CSPoR-PoRec solution. This scheme is motivated by the natural desire to outsource multiple different-sized files to a cloud storage provider and also takes a model of an abstract storage unit into account to map current cloud storage practice such as regular data loss into the realm of PoR. We showed that there is an inherent gap in the functionality of PoR such that in case the scheme returns a rejection token one is not able to formalize a provable statement about the retrievability of the *original* file. Hence, we close this gap by systematically studying this problem and propose solutions towards formalizing a proof of recoverability based on PoR techniques. In order to gather enough knowledge to output a proof of recoverability, our technique relies on repeatedly auditing the damaged files with special parameters, that is formally executing PoRP with a small challenge size. Future work may consider a different adversarial model where the adversary may dynamically delete data while the verifier aims to obtain a proof of recoverability.

Acknowledgements. The authors thank the anonymous reviewers for their valuable comments. Frederik Armknecht and Christian Gorke were financed by the Baden-Württemberg Stiftung as a part of the PAL SAaaS project. Christian Janson was supported by the German Research Foundation (DFG) SPP 1736. This work was partially supported by COST Action IC1306.

References

1. C. S. Alliance. The Treacherous 12 – Cloud Computing Top Threats in 2016, 2016. `https://downloads.cloudsecurityalliance.org/assets/research/top-threats/Treacherous-12_Cloud-Computing_Top-Threats.pdf`.
2. Amazon. Amazon S3 API, 2016. `http://docs.aws.amazon.com/AmazonS3/latest/API/s3-api.pdf`.
3. Amazon. Amazon S3 Reduced Redundancy Storage (RRS), 2017. `https://aws.amazon.com/s3/reduced-redundancy/`.
4. F. Armknecht, L. Barman, J. Bohli, and G. O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 1051–1068. USENIX Association, 2016.
5. F. Armknecht, J. Bohli, D. Froelicher, and G. O. Karame. SPORT: sharing proofs of retrievability across tenants. *IACR Cryptology ePrint Archive*, 2016:724, 2016.
6. F. Armknecht, J. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter. Outsourced proofs of retrievability. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 831–843. ACM, 2014.
7. G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. N. J. Peterson, and D. Song. Remote data checking using provable data possession. *ACM Trans. Inf. Syst. Secur.*, 14(1):12, 2011.

8. G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 598–609. ACM, 2007.
9. G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In M. Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2009.
10. G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In A. Levi, P. Liu, and R. Molva, editors, *4th International ICST Conference on Security and Privacy in Communication Networks, SECURECOMM 2008, Istanbul, Turkey, September 22-25, 2008*, page 9. ACM, 2008.
11. M. Azraoui, K. Elkhiyaoui, R. Molva, and M. Önen. Stealthguard: Proofs of retrievability with hidden watchdogs. In M. Kutylowski and J. Vaidya, editors, *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I*, volume 8712 of *Lecture Notes in Computer Science*, pages 239–256. Springer, 2014.
12. K. D. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 187–198. ACM, 2009.
13. K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In R. Sion and D. Song, editors, *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009*, pages 43–54. ACM, 2009.
14. 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 Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 501–514. ACM, 2011.
15. D. Cash, A. Küpçü, and D. Wichs. Dynamic Proofs of Retrievability via Oblivious RAM. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 279–295. Springer, 2013.
16. R. Curtmola, O. Khan, R. C. Burns, and G. Ateniese. MR-PDP: Multiple-Replica Provable Data Possession. In *ICDCS*, pages 411–420. IEEE Computer Society, 2008.
17. Y. Dodis, S. P. Vadhan, and D. Wichs. Proofs of Retrievability via Hardness Amplification. In O. Reingold, editor, *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2009.
18. C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 213–222. ACM, 2009.
19. Google. Google Storage API Reference, 2015. `https://cloud.google.com/storage/docs/json_api/v1/`.
20. Google. Google Cloud Platform: Concepts and Techniques, 2016. `http://cloud.google.com/storage/docs/concepts-techniques/`.
21. C. Gritti, W. Susilo, and T. Plantard. Efficient dynamic provable data possession with public verifiability and data privacy. In E. Foo and D. Stebila, editors, *Information Security and Privacy - 20th Australasian Conference, ACISP 2015,*

Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings, volume 9144 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2015.

22. C. Guan, K. Ren, F. Zhang, F. Kerschbaum, and J. Yu. Symmetric-key based proofs of retrievability supporting public verification. In G. Pernul, P. Y. A. Ryan, and E. R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, volume 9326 of *Lecture Notes in Computer Science*, pages 203–223. Springer, 2015.
23. V. T. Hoang, B. Morris, and P. Rogaway. An enciphering scheme based on a card shuffle. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2012.
24. A. Juels and B. S. K. Jr. PORs: Proofs Of Retrievability for Large Files. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 584–597. ACM, 2007.
25. S. Lin, T. Y. Al-Naffouri, Y. S. Han, and W. Chung. Novel polynomial basis with fast fourier transform and its application to reed-solomon erasure codes. *IEEE Trans. Information Theory*, 62(11):6284–6299, 2016.
26. M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. *J. Cryptology*, 24(3):588–613, 2011.
27. Microsoft. Microsoft Azure: How to use Blob storage from .NET, 2015. `https://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-blobs/`.
28. M. Mitzenmacher. Digital fountains: A survey and look forward. In *Information Theory Workshop, 2004. IEEE*, pages 271–276. IEEE, 2004.
29. M. B. Paterson, D. R. Stinson, and J. Upadhyay. A coding theory foundation for the analysis of general unconditionally secure proof-of-retrievability schemes for cloud storage. Cryptology ePrint Archive, Report 2012/611, 2012. `http://eprint.iacr.org/`.
30. M. B. Paterson, D. R. Stinson, and J. Upadhyay. Multi-prover proof-of-retrievability. Cryptology ePrint Archive, Report 2016/265, 2016. `http://eprint.iacr.org/`.
31. 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*, 7(6):11–20, 2013.
32. H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, volume 5350 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2008.
33. S.-T. Shen and W.-G. Tzeng. Delegable provable data possession for remote data in the clouds. In S. Qing, W. Susilo, G. Wang, and D. Liu, editors, *ICICS*, volume 7043 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2011.
34. E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM Conference on Computer and Communications Security*, pages 325–336. ACM, 2013.
35. E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In R. H. Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 229–238. ACM, 2012.

36. D. Vasilopoulos, M. Önen, K. Elkhiyaoui, and R. Molva. Message-locked proofs of retrievability with secure deduplication. In E. R. Weippl, S. Katzenbeisser, M. Payer, S. Mangard, E. Androulaki, and M. K. Reiter, editors, *Proceedings of the 2016 ACM on Cloud Computing Security Workshop, CCSW 2016, Vienna, Austria, October 28, 2016*, pages 73–83. ACM, 2016.
37. J. Xu and E. Chang. Towards efficient proofs of retrievability. In H. Y. Youm and Y. Won, editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 79–80. ACM, 2012.
38. J. Yuan and S. Yu. Proofs of retrievability with public verifiability and constant communication cost in cloud. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, Cloud Computing '13, pages 19–26, New York, NY, USA, 2013. ACM.

A Additional Details for Cloud Storage Proofs of Retrievability

We have introduced the notion of cloud storage proofs of retrievability in Section 4. Recall that we used the notion of a cloud storage as the underlying abstract model of data storage and noted that towards realizing a cloud storage architecture one needs to further introduce a *storage container*. In this section, we first provide a formal treatment of storage containers and then give an updated definition of CSPoR accommodating this new notion. Note that throughout the appendix we will stay on this abstract level and provide all details in respect to the storage containers.[11]

Let a *storage container* be denoted by $\mathfrak{S}$ storing multiple arbitrary files $F \in \{0,1\}^*$. Throughout this chapter, we assume that a client may possess several storage containers $\mathfrak{S}^{(c)}$ hosted at a cloud storage provider. We denote the total number of different storage containers by $\Gamma \in \mathbb{N}$, hence $1 \leq c \leq \Gamma$. This describes the client's potential need to handle different types of data in different storage containers, e.g. a client wishes to store important documents separately from her picture library. This also captures the common cloud storage practice where each storage container's storage space (size) is upper bounded by $\mathfrak{S}^{(c)}_{\max}$ which corresponds to the maximum number of storable files within a container, and thus we need the possibility to create multiple storage containers. The set of all storage containers is denoted by $\widehat{\mathfrak{S}}$ and is called a *cloud storage*.

A.1 Formal Definition of CSPoR

In the following we present an extended definition of the CSPoR scheme CSPoR in respect to Definition 1 using the notion of storage containers.

Definition 4. *A* cloud storage proofs of retrievability (CSPoR) scheme CSPoR *comprises the following procedures:*

[11] In case one wishes to neglect the storage containers one may simply interpret $\mathfrak{S}^{(c)}$ as a simple cloud storage and reduces the notions to $\mathfrak{S}$.

$(pk, sk, \widehat{\mathfrak{S}}, \widehat{\mathfrak{S}_{\mathrm{id}}}, \widehat{\gamma}) \xleftarrow{\$} \mathsf{CSPoRSetup}(1^\lambda)$**:** *this randomised algorithm generates a public-private key pair* (pk, sk) *and takes as input the security parameter* λ*. Additionally, it creates a cloud storage (i.e. a set of* Γ *storage containers)* $\widehat{\mathfrak{S}} := \{\mathfrak{S}^{(c)} \mid c \in [\Gamma]\}$ *and their set of respective associated unique identifiers* $\widehat{\mathfrak{S}_{\mathrm{id}}} := \{\mathfrak{S}^{(c)}_{\mathrm{id}} \mid c \in [\Gamma]\}$*. Furthermore, some meta data* $\widehat{\gamma} := \{\gamma^{(c)} \mid c \in [\Gamma]\}$ *is created for each storage container;*

$(\widehat{\mathcal{F}}, \widehat{\tau}, \widehat{\mathfrak{S}}, \widehat{\gamma}) \xleftarrow{\$} \mathsf{CSPoRStore}(sk, \widehat{F}, \widehat{\mathfrak{S}_{\mathrm{id}}})$**:** *this randomised data storing algorithm takes as input a secret key sk,* $\widehat{\mathfrak{S}_{\mathrm{id}}}$*, and the set of all files* $\widehat{F} := \{\widehat{F}_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \mid c \in [\Gamma]\}$ *a client wishes to store at the provider's cloud storage. Each* $\widehat{F}_{\mathfrak{S}^{(c)}_{\mathrm{id}}}$ *consists of* $K_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \in \mathbb{N}$ *files that will be stored within a particular* $\mathfrak{S}^{(c)}$ *where* $\widehat{F}_{\mathfrak{S}^{(c)}_{\mathrm{id}}} := \{F^{(k)} \mid F^{(k)} \in \{0,1\}^*, k \in [K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]\}$ *for* $c \in [\Gamma]$*. Each file within each storage container gets processed yielding the set of all processed files for this storage container* $\widehat{\mathcal{F}}_{\mathfrak{S}^{(c)}_{\mathrm{id}}} := \{\mathcal{F}^{(k)} \mid k \in [K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]\}$ *and a respective set of file tags is generated* $\widehat{\tau}_{\mathfrak{S}^{(c)}_{\mathrm{id}}} := \{\tau^{(k)} \mid k \in [K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]\}$ *where each tag contains additional information (e.g. meta data) about the processed file. Finally the algorithm outputs the set of all such processed files* $\widehat{\mathcal{F}} := \{\widehat{\mathcal{F}}_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \mid c \in [\Gamma]\}$*, tags* $\widehat{\tau} := \{\widehat{\tau}_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \mid c \in [\Gamma]\}$ *and the updated cloud storage* $\widehat{\mathfrak{S}}$*. The meta data* $\widehat{\gamma}$ *is also updated;*

$\delta \xleftarrow{\$} \big[\mathsf{CSPoRVerify}(pk, sk, \widehat{\tau}', \widehat{\mathfrak{S}_{\mathrm{id}}}) \rightleftharpoons \mathsf{CSPoRProve}(pk, \widehat{\mathcal{F}}', \widehat{\tau}', \widehat{\mathfrak{S}})\big]$**:** *this challenge-response protocol defines a protocol for proving cloud storage retrievability. The prover algorithm takes as input the public key pk, the file tag set* $\widehat{\tau}' := \{\widehat{\tau}'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \mid \widehat{\tau}'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} = \{\tau^{(k)} \mid k \in K'_{\mathfrak{S}^{(c)}_{\mathrm{id}}}\}, c \in \Gamma'\}$ *and the set of the processed files* $\widehat{\mathcal{F}}' := \{\widehat{\mathcal{F}}'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \mid \widehat{\mathcal{F}}'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} := \{\mathcal{F}^{(k)} \mid k \in K'_{\mathfrak{S}^{(c)}_{\mathrm{id}}}\}, c \in \Gamma'\}$*, where* $[K'_{\mathfrak{S}^{(c)}_{\mathrm{id}}}] \subseteq [K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]$ *and* $[\Gamma'] \subseteq [\Gamma]$*. The verification algorithm uses as input the key pair* (pk, sk)*, the file tag set* $\widehat{\tau}'$*, and identifiers* $\widehat{\mathfrak{S}_{\mathrm{id}}}$*. Algorithm* CSPoRVerify *finally outputs a binary value* δ *which equals* `accept` *if the verification succeeds, indicating the files in* $\widehat{\mathcal{F}}'$ *are being stored and retrievable from the cloud storage provider, and* `reject` *otherwise.*

At the beginning of the CSPoRSetup procedure, the involved parties agree on the storage containers in the set $\widehat{\mathfrak{S}}$. Similarly, they agree on the files in the set $\widehat{\mathcal{F}}$ at the beginning of the CSPoRStore procedure. Note this does not require the files being given in clear within the agreement. An agreement could also consist of hashes of these files. Moreover note that the cloud storage $\widehat{\mathfrak{S}}$ may already contain data from previously performed CSPoRStore procedures.

As described in Section 4.1, observe that $\widehat{\mathcal{F}}$ may not be exactly equal to $\widehat{F}$ but it must be guaranteed that $\widehat{F}$ can be recovered from $\widehat{\mathcal{F}}$. Additionally, we wish to remark that the involved file tag set $\widehat{\tau}'$ in the challenge-response protocol can correspond to either the full set of file tags $\widehat{\tau}$ or any arbitrary subset of file tags enabling a CSPoR scheme to flexibly check any set of files by specifying the appropriate tags.

Definition 5. *We denote the challenge-response procedure of the CSPoR scheme* CSPoR *given as*

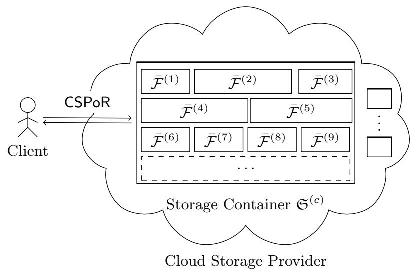
$$\Big[\mathsf{CSPoRVerify}(pk, sk, \widehat{\tau}', \widehat{\mathfrak{S}_{\mathrm{id}}}) \rightleftharpoons \mathsf{CSPoRProve}(pk, \widehat{\mathcal{F}}', \widehat{\tau}', \widehat{\mathfrak{S}})\Big]$$

by $\mathsf{CSPoRP}(pk, sk, \widehat{\tau}', \widehat{\mathfrak{S}_{\mathrm{id}}}, \widehat{\mathcal{F}}', \widehat{\mathfrak{S}})$*, and if the context is clear briefly* CSPoRP*. A single challenge-response step of a* CSPoRP *is called an* audit*.*

Informally, a CSPoR protocol is *correct* if all processed files $\widehat{\mathcal{F}}$ outputted by the store procedure $\mathsf{CSPoRStore}$ will be accepted by the verification algorithm when interacting with a valid prover. More formally this is captured as follows.

Definition 6. *A CSPoR protocol is* correct *if there exists a negligible function* negl *such that for every security parameter* λ*, every key pair* (pk, sk) *and set of storage containers* $\widehat{\mathfrak{S}}$ *with respective identifiers* $\widehat{\mathfrak{S}_{\mathrm{id}}}$ *and meta data* $\widehat{\gamma}$ *generated by* $\mathsf{CSPoRSetup}$*, for all sets of files* $\widehat{F}$ *(containing files* $F^{(k)} \in \{0,1\}^*$*), and for all* $(\widehat{\mathcal{F}}, \widehat{\tau}, \widehat{\mathfrak{S}}, \widehat{\gamma})$ *generated by* $\mathsf{CSPoRStore}$*, it holds that*

$$\begin{aligned}\Pr\big[\big(&\mathsf{CSPoRVerify}(pk, sk, \widehat{\tau}', \widehat{\mathfrak{S}_{\mathrm{id}}}) \\ &\rightleftharpoons \mathsf{CSPoRProve}(pk, \widehat{\mathcal{F}}', \widehat{\tau}', \widehat{\mathfrak{S}})\big) \nrightarrow \texttt{accept}\big] = \mathrm{negl}(\lambda).\end{aligned}$$



Client
CSPoR
$\bar{\mathcal{F}}^{(1)}$
$\bar{\mathcal{F}}^{(2)}$
$\bar{\mathcal{F}}^{(3)}$
$\bar{\mathcal{F}}^{(4)}$
$\bar{\mathcal{F}}^{(5)}$
$\bar{\mathcal{F}}^{(6)}$
$\bar{\mathcal{F}}^{(7)}$
$\bar{\mathcal{F}}^{(8)}$
$\bar{\mathcal{F}}^{(9)}$
...
Storage Container $\mathfrak{S}^{(c)}$
Cloud Storage Provider

Fig. 2. Model of a CSPoR scheme between a client and cloud storage provider with a detailed representation of a storage container containing multiple different-sized files.

In Figure 2, we illustrate the execution of a CSPoR scheme between a client and a cloud storage provider as well as represent a model of a storage container $\mathfrak{S}^{(c)}$. The client is able to access the storage container held by the CSP via a secure channel. The storage container $\mathfrak{S}^{(c)}$ can contain an arbitrary set of files $\widehat{\mathcal{F}}$ which were uploaded by the client during a $\mathsf{CSPoRStore}$ procedure, and the CSP may hold an arbitrary amount Γ of storage containers (displayed right of $\mathfrak{S}^{(c)}$). Here, $\bar{\mathcal{F}}^{(k)} := (\mathcal{F}^{(k)} \| \sigma^{(k)} \| \tau^{(k)})$ for $k \in [K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]$ denotes a processed file with respective authentication tags and file tag stored in the storage container $\mathfrak{S}^{(c)}$ located at the cloud storage provider. Note that in practice the data of the processed files and associated tags may be stored separately.

A.2 Instantiation

In this section we present an instantiation which is to a large extent similar to the one in Section 4.3, however takes the notion of a storage container into account.

Building Blocks Unless otherwise specified all operations are performed over the finite field $\mathbb{F} = \mathbb{Z}_p$ where p is a λ-bit prime with λ being the security parameter. As we instantiate a private-key CSPoR system it suffices to make use of a symmetric encryption scheme and we set the public key $pk = \perp$. We make use of a pseudo-random function $g\colon \{0,1\}^* \times \{0,1\}^{\phi_{\text{prf}}} \to \mathbb{F}$, where ϕ_{prf} is the key length of the PRF[12], and a MAC scheme. Furthermore, we make use of storage containers $\mathfrak{S}^{(c)}$, $c \in [\mathit{\Gamma}]$, of the cloud storage $\widehat{\mathfrak{S}}$ which contains all outsourced data.

Specification of the $\mathsf{CSPoRSetup}$ Procedure In the $\mathsf{CSPoRSetup}$ procedure the client derives a random symmetric key $\kappa_{\text{enc}} \overset{\$}{\leftarrow} \mathcal{K}_{\text{enc}}$ and a random MAC key $\kappa_{\text{mac}} \overset{\$}{\leftarrow} \mathcal{K}_{\text{mac}}$. The secret key $sk = (\kappa_{\text{enc}}, \kappa_{\text{mac}})$ is kept secret and requests create a cloud storage $\mathfrak{S}$, i.e. a set of storage containers located at the server S.

Specification of the $\mathsf{CSPoRStore}$ Procedure The $\mathsf{CSPoRStore}$ procedure is initiated by the client holding files $\widehat{F} = \{\widehat{F}_{\mathfrak{S}^{(c)}_{\text{id}}} \mid c \in [\mathit{\Gamma}]\}$ that she wishes to store in $\widehat{\mathfrak{S}}$. Then, for each file $F^{(k)}$ of $\widehat{F}$, $k \in [K_{\mathfrak{S}^{(c)}_{\text{id}}}]_{c\in[\mathit{\Gamma}]}$, the steps given for the specification of $\mathsf{CSPoRStore}$ in Section 4.3 are carried out. At the end, the client combines all authentication tags into the set $\widehat{\sigma}$, all file tags into the set $\widehat{\tau}$, as well as all processed files are aggregated as the set $\widehat{\mathcal{F}}$. The three sets will be uploaded to the respective storage container in $\widehat{\mathfrak{S}}$ located at the cloud storage provider while $\widehat{\sigma}$ and $\widehat{\mathcal{F}}$ will be removed locally from the client ($\widehat{\tau}$ is optional). Note that in the fifth step the provider only learns the size of the outsourced file, since the remaining part of $\tau^{(k)}$ is encrypted with the client's private key.

Specification of the CSPoRP Procedure The CSPoRP procedure obtains an assurance about the retrievability of the files. In the following we describe the technical details of an audit step providing a reply δ. Note that the client may wish to audit only a subset of all outsourced files, hence we have $[K'_{\mathfrak{S}^{(c)}_{\text{id}}}] \subseteq [K_{\mathfrak{S}^{(c)}_{\text{id}}}]$, $c \in [\mathit{\Gamma}']$, $[\mathit{\Gamma}'] \subseteq [\mathit{\Gamma}]$.

1. The client first verifies the MAC on each $\tau^{(k)}$ within $\widehat{\tau}$. If the MAC is invalid the client aborts the protocol and outputs `reject`. Otherwise, she parses all $\tau^{(k)}$ from $\widehat{\tau}$ and uses κ_{enc} in order to recover $\tilde{n}^{(k)}$, $\kappa^{(k)}_{\text{prf}}$ and $\alpha^{(k)}_1, \ldots, \alpha^{(k)}_s$ for all $k \in [K'_{\mathfrak{S}^{(c)}_{\text{id}}}]_{c\in[\mathit{\Gamma}']}$;

[12] Note that we use the shorthand $g_{\kappa^{(k)}_{\text{prf}}}(i) := g(\kappa^{(k)}_{\text{prf}}, i)$.

2. Next the client selects a random subset $I^{(k)} \subseteq_{\$} [\tilde{n}^{(k)}]$ of size $\ell^{(k)}$ and chooses for each $i \in I^{(k)}$ a random element from the finite field $\nu_i^{(k)} \xleftarrow{\$} \mathbb{F}$ for all $k \in [K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]_{c\in[\Gamma']}$;
3. Then the client generates the challenge by aggregating the sampled values from the step before per file to a set $Q^{(k)} = \{(i, \nu_i^{(k)})_{i\in I^{(k)}}\}$ of size $\ell^{(k)}$, for all $k \in [K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]_{c\in[\Gamma']}$. All sets $Q^{(k)}$ per storage container are aggregated as $\mathcal{Q}_{\mathfrak{S}_{\mathrm{id}}^{(c)}} := \{Q^{(k)} \mid k \in [K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]\}$. Finally, the combined set over all containers $\widehat{Q} := \{\mathcal{Q}_{\mathfrak{S}_{\mathrm{id}}^{(c)}} \mid c \in [\Gamma']\}$ is then sent to the provider.

The cloud service provider now parses all files from $\widehat{\mathcal{F}}$ as $\{f_{ij}^{(k)}\}$ and $\{\sigma_i^{(k)}\}$, and the corresponding challenges $Q^{(k)}$ from $\widehat{Q}$. Then, the provider computes for $1 \leq j \leq s$ and all $k \in [K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]_{c\in[\Gamma']}$

$$\mu_j^{(k)} \longleftarrow \sum_{\left(i,\nu_i^{(k)}\right)\in Q^{(k)}} \nu_i^{(k)} f_{ij}^{(k)}, \qquad \sigma^{(k)} \longleftarrow \sum_{\left(i,\nu_i^{(k)}\right)\in Q^{(k)}} \nu_i^{(k)} \sigma_i^{(k)}.$$

This execution is repeated for all files in all storage containers contained in $\widehat{Q}$. Next, S accumulates all responses and authentication tags to output for $1 \leq j \leq s$

$$\widetilde{\mu}_j := \sum_{c\in[\Gamma']} \sum_{k\in\left[K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}\right]} \mu_j^{(k)} \qquad \text{and} \qquad \widetilde{\sigma} := \sum_{c\in[\Gamma']} \sum_{k\in\left[K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}\right]} \sigma^{(k)}.$$

Finally, the client parses the provider's accumulated response and checks

$$\widetilde{\sigma} \stackrel{?}{=} \sum_{c\in[\Gamma']} \sum_{k\in\left[K'_{\mathfrak{S}_{\mathrm{id}}^{(c)}}\right]} \left(\sum_{\left(i,\nu_i^{(k)}\right)\in Q^{(k)}} \nu_i^{(k)} g_{\kappa_{\mathrm{prf}}^{(k)}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} \widetilde{\mu}_j \right).$$

If this equality check is successful, the verifier outputs $\delta = \texttt{accept}$, and otherwise she outputs $\delta = \texttt{reject}$.

Correctness of the Instantiation

Now we present that our above scheme is correct. Let the PRF key be $\kappa_{\mathrm{prf}}^{(k)}$ and $\alpha_1^{(k)}, \ldots, \alpha_s^{(k)} \xleftarrow{\$} \mathbb{F}$ be the secret coefficients for all $k \in [K_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]_{c\in[\Gamma]}$. Let the file symbols be denoted by $\{f_{ij}^{(k)}\}$, and the block authenticators are expressed as $g_{\kappa_{\mathrm{prf}}^{(k)}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} f_{ij}^{(k)}$. For a prover that responds honestly to queries from $\widehat{Q}$ such that $\widetilde{\mu}_j = \sum_{c\in[\Gamma]} \sum_{k\in[K_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]} \mu_j^{(k)}$ and $\widetilde{\sigma} = \sum_{c\in[\Gamma]} \sum_{k\in[K_{\mathfrak{S}_{\mathrm{id}}^{(c)}}]} \sigma^{(k)}$ then we have

$$\begin{aligned}
\widetilde{\sigma} &= \sum_{c\in[\mathit{\Gamma}]} \sum_{k\in[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]} \sigma^{(k)} = \sum_{c\in[\mathit{\Gamma}]} \sum_{k\in[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]} \left(\sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} \sigma_i^{(k)} \right) \\
&= \sum_{c\in[\mathit{\Gamma}]} \sum_{k\in[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]} \left(\sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} \left(g_{\kappa^{(k)}_{\mathrm{prf}}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} f_{ij}^{(k)} \right) \right) \\
&= \sum_{c\in[\mathit{\Gamma}]} \sum_{k\in[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]} \left(\sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} g_{\kappa^{(k)}_{\mathrm{prf}}}(i) + \sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} \sum_{j=1}^{s} \alpha_j^{(k)} f_{ij}^{(k)} \right) \\
&= \sum_{c\in[\mathit{\Gamma}]} \sum_{k\in[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]} \left(\sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} g_{\kappa^{(k)}_{\mathrm{prf}}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} \sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} f_{ij}^{(k)} \right) \\
&= \sum_{c\in[\mathit{\Gamma}]} \sum_{k\in[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}}]} \left(\sum_{(i,\nu_i^{(k)})\in Q^{(k)}} \nu_i^{(k)} g_{\kappa^{(k)}_{\mathrm{prf}}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} \mu_j^{(k)} \right),
\end{aligned}$$

which shows that verification is satisfied.

A.3 Communication Model

We have already mentioned in Section 4 that the above CSPoR system can be translated straightforwardly into present cloud architectures. This can be achieved by introducing procedures that capture the communication steps between a client and a cloud storage provider. The expression

$$\mathit{\Pi}\colon\ [\mathsf{C}\colon \mathit{in}_{\mathsf{C}};\ \mathsf{S}\colon \mathit{in}_{\mathsf{S}}] \longrightarrow [\mathsf{C}\colon \mathit{out}_{\mathsf{C}};\ \mathsf{S}\colon \mathit{out}_{\mathsf{S}}]$$

denotes the event that a client C and a provider S run an interactive protocol $\mathit{\Pi}$ where $\mathit{in}_{\mathcal{X}}$ and $\mathit{out}_{\mathcal{X}}$ denote the input and output of entity $\mathcal{X}$ (either C or S), respectively.

In the following we describe the execution of the required procedures. Following the order of our algorithms in Definition 4, we first need to run the procedure Create between C and S in order to create a storage container $\mathfrak{S}^{(c)}$ located at the server in which the client stores her files. Note that a storage container is upper-bounded by $\mathfrak{S}^{(c)}_{\max}$, i.e. C and S need to engage in another Create procedure to create a new storage container as soon as the maximal storage capacity is reached or a client wishes to store different types of data in different storage containers. In more detail, the procedure

$$\mathsf{Create}\colon\ [\mathsf{C}\colon pk;\ \mathsf{S}\colon pk] \longrightarrow \left[\mathsf{C}\colon \mathfrak{S}^{(c)}_{\mathrm{id}}, \gamma_{\mathsf{C}};\ \mathsf{S}\colon \mathfrak{S}^{(c)}, \gamma_{\mathsf{S}}\right]$$

takes no other inputs than the public keys of both parties and outputs the identifier $\mathfrak{S}_{\mathrm{id}}^{(c)}$ to identify the storage container $\mathfrak{S}^{(c)}$ and a tag γ_{C} which contains meta data related to $\mathfrak{S}^{(c)}$ for the client. The CSP initialises the storage container $\mathfrak{S}^{(c)}$ on its infrastructure and obtains a tag γ_{S} as its output.

After the successful generation of a storage container a client wishes to store her files by executing a Store procedure as follows

$$\mathsf{Store}\colon \quad \left[\mathsf{C}\colon \widehat{F}, \mathfrak{S}_{\mathrm{id}}^{(j)};\ \mathsf{S}\colon \mathfrak{S}^{(c)}\right] \longrightarrow \left[\mathsf{C}\colon \widehat{\kappa}, \widehat{\tau};\ \mathsf{S}\colon \widehat{\mathcal{F}}, \mathfrak{S}^{(c)}, \widehat{\tau}\right].$$

A client needs to provide as input her set of files $\widehat{F}$ and the respective storage container identifier $\mathfrak{S}_{\mathrm{id}}^{(c)}$ to store the files in $\mathfrak{S}^{(c)}$. The procedure outputs the set of processed files $\widehat{\mathcal{F}}$ and the updated storage container $\mathfrak{S}^{(c)}$ for the server. Furthermore, the client receives a set $\widehat{\kappa}$ which contains keys for file dependent functions (e.g. MACs or PRFs) and a set of verification tags $\widehat{\tau}$ which are computed on the client side. Those tags are also provided to S and used to check consistency of the file sizes.

Finally, a client C and cloud storage provider S engage in a CSPoRP procedure.

$$\mathsf{CSPoRP}\colon \quad \left[\mathsf{C}\colon \widehat{\tau}, \widehat{\mathfrak{S}_{\mathrm{id}}};\ \mathsf{S}\colon \widehat{\tau}, \widehat{\mathfrak{S}}\right] \longrightarrow [\mathsf{C}\colon \delta;\ \mathsf{S}\colon \bot].$$

In this procedure, a client provides her file tags $\widehat{\tau}$ and her respective set of storage container identifier $\widehat{\mathfrak{S}_{\mathrm{id}}}$. Note that in general our CSPoR scheme enables a client to check whether all outsourced files in $\widehat{\mathfrak{S}}$ are intact and retrievable. However, it is also possible for a client to check only a subset of her outsourced files by simply choosing a subset of tags $\widehat{\tau}'$ from $\widehat{\tau}$. The server inputs $\widehat{\mathfrak{S}}$ and the file tags $\widehat{\tau}$. The protocol run is accepted by the verifier if $\delta = \texttt{accept}$, or rejected otherwise. More precisely, the CSPoRP procedure uses additional locally computed values in order to be executed. Following our instantiation in Appendix A.2, a client prepares her challenge set $\widehat{Q}$ according to the file tags $\widehat{\tau}$ and sends the challenge set to S, i.e.,

$$\mathsf{SendChallenge}\colon \quad \left[\mathsf{C}\colon \widehat{Q};\ \mathsf{S}\colon \bot\right] \longrightarrow \left[\mathsf{C}\colon \bot;\ \mathsf{S}\colon \widehat{Q}\right].$$

The server uses the challenge set and computes the *authentication tags* σ and *responses* μ as its replies which are returned to C, i.e.,

$$\mathsf{Response}\colon \quad \left[\mathsf{C}\colon \bot;\ \mathsf{S}\colon \widehat{\mathfrak{S}}, \widehat{\tau}, I\right] \longrightarrow [\mathsf{C}\colon \sigma, \mu;\ \mathsf{S}\colon \bot].$$

Finally, a client uses the authentication tags and response values to verify the CSPoRP procedure as in Appendix A.2. The client outputs a binary decision value δ indicating whether she accepts or rejects the CSPoRP procedure.

The introduced procedures are easily translated into current cloud architectures as mentioned in Remark 2. The `POST` and `GET` commands can achieve different functionalities by simply specifying different parameters as detailed in their respective APIs.

B CSPoR-PoRec for Multiple Cloud Storage Container

The algorithm for CSPoR-PoRec given in Algorithm 2 can be easily extended to the multiple cloud storage container case. For each audit (cf. Definition 5) performed in Lines 3-5, the set of all files to be checked $\widehat{\mathcal{F}}'$ is defined as

$$\widehat{\mathcal{F}}' := \left\{ \widehat{\mathcal{F}}'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \mid c \in [\Gamma'] \right\}$$

where

$$\widehat{\mathcal{F}}'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} := \left\{ \mathcal{F}^{(k)} \mid k \in \left[K'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \right] \right\}$$

for

$$\left[K'_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \right] \subseteq \left[K_{\mathfrak{S}^{(c)}_{\mathrm{id}}} \right] \qquad \text{and} \qquad [\Gamma'] \subseteq [\Gamma] \,.$$

The remaining parts of the algorithm CSPoR-PoRec work as previously specified in Section 5.3.