Cloud Storage File Recoverability

Christian A. Gorke, Frederik Armknecht
University of Mannheim, Germany
gorke@uni-mannheim.de
armnkecht@uni-mannheim.de

Christian Janson
Technische Universität Darmstadt, Germany
christian.janson@crisp-da.de

Carlos Cid
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.

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 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, 28] and *Proofs of Data Possession* (PDP) [8]. In a nutshell, such schemes work as follows. Before uploading a file, a user preprocesses 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

SCC'17, April 2, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3055259.3055264

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.

2. PRELIMINARIES

In this section, we briefly recall the PoR model as described in previous work [24, 28]. Furthermore, we discuss shortcomings of the notion since we want to accommodate multiple files and regular data loss in the realm of cloud storage.

2.1 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[1], 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 $\delta \in \{\texttt{accept}, \texttt{reject}\}$.[2] 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 $\delta = \texttt{reject}$ with overwhelming probability indicating that the *processed* file is not fully intact anymore. In other words, the CSP is misbehaving. Note that we discuss in the full version [21] different types of ECC that can be employed in the realm of PoR.

2.2 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.

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

[2]For a formal definition of PoR please refer to [28].

2.3 Multiple Files

It is natural that a client aims to store multiple (different-sized) files $F^{(1)}, \ldots, F^{(f)}$ 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 $F^{(1)}, \ldots, F^{(f)}$ 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 $\widehat{F} = F^{(1)} \| F^{(2)} \| \ldots \| F^{(f)}$ 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 $\mathcal{F} = F^{(1)} \| F^{(2)} \| \ldots \| F^{(f)} \| P^{(1,\ldots,f)} = \widehat{F} \| P^{(1,\ldots,f)}$ where $P^{(1,\ldots,f)}$ 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 $F^{(i)}$, $i \in [f] := \{1, \ldots, f\}$, 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 $\mathcal{F} = F^{(1)} \| P^{(1)} \| F^{(2)} \| P^{(2)} \| \ldots \| F^{(f)} \| P^{(f)}$ 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 3.

2.4 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 4, we discuss in detail our approach and solution towards ensuring recoverability of the original file and thus closing an important gap within the PoR functionality. We also propose a solution which is applicable in the multi file case.

3. 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.

3.1 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:*

$(pk, sk, \mathfrak{S}) \xleftarrow{\$} \mathsf{CSPoRSetup}(1^\lambda)$**:** *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 $\mathfrak{S}$;*

$\delta \xleftarrow{\$} \big[\mathsf{CSPoRVerify}(pk, sk, \widehat{\tau}') \rightleftharpoons \mathsf{CSPoRProve}(pk, \widehat{\mathcal{F}}', \widehat{\tau}')\big]$**:** *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 $\widehat{\tau}' := \{\tau^{(k)} \mid k \in [K']\}$ 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* 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 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 CSPoRStore will be accepted by the verification algorithm when interacting with a valid prover. We denote the above challenge-response procedure of CSPoR by $\mathsf{CSPoRP}(pk, sk, \widehat{\tau}', \widehat{\mathcal{F}}')$, if the context is clear briefly CSPoRP, which we will refer to as an *audit*.

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 *[25]. 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.

3.2 Security Model

The underlying security model for a CSPoR scheme captures the usual *extractability* notion, i.e., the adversary aims to convince the verifier with overwhelming probability that the files are still fully intact and retrievable. In our model, we crucially extend the notion to accommodate multiple files. Due to space restrictions, the full security model and proof can be found in the full version [21].

3.3 Instantiation Details

Our concrete instantiation is based on the private PoR scheme of Shacham and Waters (SW-PoR) [28] mainly due to its ability to handle an unbounded number of verification queries in a compact way. In case a better communication complexity is required, one may build upon the scheme presented in [13]. 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 2.3, 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. Furthermore, we make use of a cloud storage $\mathfrak{S}$, cf. Section 3.1, which contains all outsourced data.

Specification of the CSPoRSetup *Procedure*

In the 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 CSPoRStore *Procedure*

The 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 $1 \le i \le \tilde{n}^{(k)}$, $1 \le j \le s$, 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 block of $\mathcal{F}^{(k)}$ an authentication tag $\sigma_i^{(k)} \leftarrow g_{\kappa_{\text{prf}}^{(k)}}(i) + \sum_{j=1}^{s} \alpha_j^{(k)} f_{ij}^{(k)} \in \mathbb{F}$, $i \in [\tilde{n}]$;
5. At last, compute file tag $\tau^{(k)} := \tau_0^{(k)} \| \mathsf{MAC}_{\kappa_{\text{mac}}}(\tau_0^{(k)})$, 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 $1 \le j \le s$ and all $k \in [K']$ the values $\mu_j^{(k)} \leftarrow \sum_{\left(i, \nu_i^{(k)}\right) \in Q^{(k)}} \nu_i^{(k)} f_{ij}^{(k)}$ and $\sigma^{(k)} \leftarrow \sum_{\left(i, \nu_i^{(k)}\right) \in Q^{(k)}} \nu_i^{(k)} \sigma_i^{(k)}$. Next, the CSP accumulates all responses and authentication tags to output $\widetilde{\mu}_j := \sum_{k \in [K']} \mu_j^{(k)}$ and $\widetilde{\sigma} := \sum_{k \in [K']} \sigma^{(k)}$ for each $1 \le j \le s$ 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 and the formal treatment is deferred to the full version [21].

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 [25]. 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 are deferred to the full version [21].*

4. 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.[3] 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

Algorithm 1: PoRec (Proofs of Recoverability)

Input: C: Filetag τ, ECC code rate ρ; 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 1
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 $I \xleftarrow{\$!\{\ell\}} [\tilde{n}] \setminus S$
9 $S \longleftarrow S \cup I$
10 $\delta' \longleftarrow \mathsf{CSPoRP}'(I, \tau, \mathcal{F})$
11 **if** $\delta' =$ `accept` **then** a $\leftarrow$ a $+1$
12 **else** r $\longleftarrow$ r $+1$
13 **if** a $= n$ **then return** `accept` // Theorem 2
14 **if** r $= \tilde{n} - n + 1$ **then return** `reject` // Theorem 2
15 **return** `reject`

the extraction of $\tilde{n}$ from τ, Line 3 follows from Theorem 1, 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 2. 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 4.3.

[3] As discussed in Section 2.4, 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.

4.1 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 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 1 we show that $\ell = 1$ enables us to learn whether a certain block is valid.

THEOREM 1 (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.

4.2 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 2 gives lower and upper bounds for A.

THEOREM 2 (AUDIT BOUNDS). *Let ρ, $\tilde{n}$, n be defined as in Theorem 1, 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.*

4.3 Locating Damaged Files

As described in both preceding Sections 4.1 and 4.2, 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 (◯) 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 (□), 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 (✓).

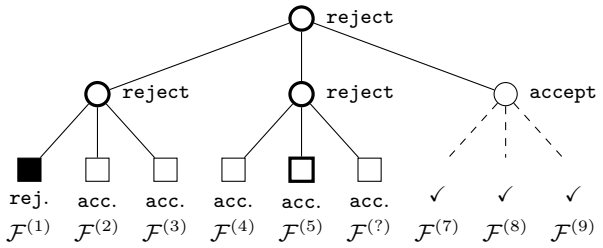

Figure 1: Traversing the file tree spanned over nine different outsourced files. Four times CSPoRP is performed (◯) and six times PoRec (□). As a result $\mathcal{F}^{(1)}$ is damaged beyond repair (■), $\mathcal{F}^{(5)}$ is damaged but recoverable (□), and all other files are sound and recoverable.

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.

Combining CSPoR *and* PoRec

Combining CSPoR and PoRec to a single procedure yields our final algorithm $\mathsf{CSPoR\text{-}PoRec}(\widehat{F}, \rho, \lambda)$, see Algorithm 2. CSPoR-PoRec checks from time to time the retrievability of the fileset $\widehat{\mathcal{F}}$ (or a subset thereof) employing CSPoR. This is done until an error occurs. Then, the corrupted files get located and PoRec is performed outputting references to all recoverable files as well as all irrecoverable files as A and R, respectively.

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*

Algorithm 2: CSPoR-PoRec

Input: C: set of files $\widehat{F}$, ECC code rate ρ, security parameter λ
Output: List of recoverable (A) and non-recoverable (R) files

$(pk, sk, \mathfrak{S}) \longleftarrow \mathsf{CSPoRSetup}(1^\lambda)$ // Definition 1
$(\widehat{\mathcal{F}}, \widehat{\tau}, \mathfrak{S}) \longleftarrow \mathsf{CSPoRStore}(sk, \widehat{F})$ // Definition 1
repeat
  $\delta \longleftarrow \mathsf{CSPoRP}(pk, sk, \widehat{\tau}', \mathfrak{S}, \widehat{\mathcal{F}}')$ // Definition 1
until $\delta =$ `reject`
Create b-ary tree T using $\widehat{\mathcal{F}}'$
repeat
  Perform CSPoRP for all child nodes of rejected nodes of T
until *tree traversed as needed, yielding* $\widehat{\mathcal{F}}'_c$ // Section 4.3
$A \longleftarrow \emptyset, R \longleftarrow \emptyset$
foreach $\mathcal{F}_c \in \widehat{\mathcal{F}}'_c$ **do**
  $\delta \longleftarrow \mathsf{PoRec}(pk, sk, \tau, \rho, \mathcal{F}_c)$ // Algorithm 1
  if $\delta =$ `accept` **then** $A \longleftarrow A \cup \tau$
  else $R \longleftarrow R \cup \tau$
return (A, R)

might be already convinced if tn accept tokens are counted for a certain threshold $0 < t \leq 1$.

4.4 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.

5. RELATED WORK

Proofs of Retrievability [6, 11, 13, 15, 17, 23, 24, 26, 27, 28, 29, 32] 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 [28], and [17] introducing the notion of PoR codes. 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 [29] improves the performance by relying on a Merkle hash tree. Stefanov et al. [30] 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 [22] 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. [23] 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. [31] 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.

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 dy-

namically delete data while the verifier aims to obtain a proof of recoverability.

Acknowledgements

Christian A. Gorke and Frederik Armknecht 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.

7. 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. A. Gorke, C. Janson, F. Armknecht, and C. Cid. Cloud storage file recoverability. Cryptology ePrint Archive, Report 2017/167, 2017. http://eprint.iacr.org/2017/167.

[22] 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.

[23] 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.

[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] 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/.

[26] 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.

[27] 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/.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.