Message-Locked Proofs of Retrievability with Secure Deduplication

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

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

Kaoutar Elkhiyaoui
EURECOM
Sophia Antipolis, France
elkhiyao@eurecom.fr

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

ABSTRACT

This paper addresses the problem of data retrievability in cloud computing systems performing deduplication to optimize their space savings: While there exist a number of proof of retrievability (PoR) solutions that guarantee storage correctness with cryptographic means, these solutions unfortunately come at odds with the deduplication technology. To reconcile proofs of retrievability with file-based cross-user deduplication, we propose the *message-locked* PoR approach whereby the PoR effect on duplicate data is identical and depends on the value of the data segment, only. As a proof of concept, we describe two instantiations of existing PoRs and show that the main extension is performed during the setup phase whereby both the keying material and the encoded version of the to-be-outsourced file is computed based on the file itself. We additionally propose a new *server-aided message-locked key generation* technique that compared with related work offers better security guarantees.

Keywords

secure cloud storage;proofs of retrievability; data deduplication; message-locked encryption; server aided encryption

1. INTRODUCTION

With the various advantages such as highly efficient and reliable storage facilities and cost savings, the adoption of the cloud technology is on a significant rise. Among major cloud providers, Amazon S3 and Google drive nowadays offer unlimited storage space almost for free [1, 2]. As the number of customers and the volume of stored data continues to increase, so do the concerns: On the one hand, by outsourcing the storage of their data, customers lend the full control of their data to cloud providers and have no means to verify the integrity of their services; on the other hand, cloud service providers are facing an exponential storage consumption which becomes extremely difficult to control.

To address the problem of data integrity and hence increase customer trust in the cloud, several proof of retrievability (PoR) schemes [18, 5, 23, 7] have been proposed whereby the cloud storage server proves to its customers that their data are correctly stored. Such techniques implement a secure encoding algorithm that incorporates some integrity values within the file before the upload. These values are further used during the PoR verification phase. While existing PoR schemes mainly look for means to optimize the performance of PoR at the customer side, they usually assume that cloud providers have potentially infinite resources to store data and compute proofs of retrievability. Such an assumption unfortunately becomes too strong with the explosion of digital content. Cloud providers, nowadays, look for different data reduction techniques including data deduplication [14] to optimize their storage capacity: By eliminating duplicate copies of data, cloud servers achieve high storage space savings [20]. Unfortunately, current PoR solutions are incompatible with data deduplication as the integrity values resulting from the encoding algorithm are generated using a secret key that is only known to the owner of the file, and thus unique. Therefore, the encoding of a given file by two different customers results in two different outputs.

In this paper, we aim at consolidating PoR with file-based deduplication and propose *message-locked proofs of retrievability* (ML-PoR) which assure that the underlying encoding algorithm is deterministic and hence produces the same output for duplicate files. As its name suggests, in a *message-locked* PoR, the PoR effect is derived from the file, solely. Hence, the encoding of duplicate files by different customers results in the same outputs which can thus be deduplicated. A similar approach has been proposed to cope with the problem of deduplication over encrypted data: Secure deduplication solutions make use of *message-locked encryption* [14, 9] whereby the encryption key is derived from the data. Since a symmetric PoR scheme makes use of secret keys during the encoding process and the verification of data retrievability, we propose that, likewise, a ML-PoR extracts these keys from the file. In order to protect the secrecy of these *message-locked* keys which can easily be discovered if files are predictable, the proposed *message-locked* PoR makes use

CCSW'16, October 28 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4572-9/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2996429.2996433

of a dedicated key generation technique such as the recently proposed server-aided key generation techniques [8, 4] which prevent cloud servers from discovering *message-locked* keys through simple dictionary attacks. Such techniques output a cryptographic key that not only depends on the actual file but also on a secret key generated by a key server. In addition to the design of a *message-locked* PoR, we propose a new *server-aided message-locked key generation* technique which compared to existing solutions, relaxes the trust model and ensures that neither the cloud server nor the key server can guess *message-locked* keys.

The rest of this paper is organized as follows. Section 2 reviews the definition of a proof of retrievability scheme and further introduces the inherent conflict between PoR and deduplication. The idea of the underlying solution is presented in Section 3. Section 4 describes the entire solution including the newly proposed server-aided message-locked key generation technique. Section 5 specifies two instantiations building upon two different PoR schemes. The security and performance of the solution are analyzed in Sections 6 and 7, respectively.

2. PROBLEM STATEMENT

In this section, we remind the formal definition of a proof of retrievability (PoR) scheme and briefly sketch existing solutions. We further describe the main problem we tackle in this paper, namely, the incompatibility of PoRs with data deduplication.

2.1 Proofs of Retrievability

With the almost unlimited storage capacity offered by cloud service providers, customers tend to outsource large amounts of data. Since the storage operation is remotely performed by potentially malicious cloud storage servers, customers lose the control over their data and therefore look for some guarantees on the correct storage operation. The recently proposed proofs of retrievability (PoR) [5, 23, 18, 7] are cryptographic techniques that enable customers to verify the correct storage of their large data without the need for retrieving it entirely.

Formally, a PoR scheme involves a data owner $\mathcal{DO}$ who wishes to upload a file F to a cloud storage server $\mathcal{CS}$ and to further query $\mathcal{CS}$ for some proofs on the integrity of F using five polynomial-time algorithms:

- $\mathsf{KeyGen}(1^\tau) \rightarrow K$: This probabilistic key generation algorithm is executed by $\mathcal{DO}$ as a first step. It takes as input a security parameter τ, and outputs some keying material K for $\mathcal{DO}$.

- $\mathsf{Encode}(K, F) \rightarrow (\mathsf{fid}, \hat{F})$: $\mathcal{DO}$ calls this algorithm before the actual outsourcing of her file. It takes as inputs the key K and the file F composed of n splits, namely, $F = \{S_1, S_2, ..., S_n\}$ and returns a unique identifier fid for F and an encoded version $\hat{F} = \{\hat{S}_1, \hat{S}_2, ..., \hat{S}_n\}$ which includes some additional information that will further help for the verification of F's retrievability.

- $\mathsf{Challenge}(K, \mathsf{fid}) \rightarrow \mathsf{chal}$: Once the encoded file $\hat{F}$ with identifier fid is outsourced to $\mathcal{CS}$, $\mathcal{DO}$ invokes this algorithm with secret key K and file identifier fid to compute a PoR challenge chal and sends the result to $\mathcal{CS}$.

- $\mathsf{ProofGen}(\mathsf{fid}, \mathsf{chal}) \rightarrow \mathcal{P}$: This algorithm is executed by $\mathcal{CS}$ and returns a proof of retrievability $\mathcal{P}$ given file identifier fid and the PoR challenge chal.

- $\mathsf{ProofVerif}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P}) \rightarrow b \in \{0, 1\}$: Upon reception of $\mathcal{P}$, data owner $\mathcal{DO}$ calls this algorithm which takes as input the key K, the file identifier fid, the challenge chal and the proof $\mathcal{P}$ and outputs $b = 1$ if $\mathcal{P}$ is a valid proof and $b = 0$ otherwise.

Existing PoR schemes are probabilistic solutions: They ensure the integrity of the entire data with a certain high probability. For each PoR query, $\mathcal{DO}$ samples a subset of file blocks and verifies their integrity. PoR solutions can be classified into two categories which mainly differ with respect to their setup phases where F is initially encoded before its actual outsourcing to $\mathcal{CS}$. Their respective encoding functions either integrate pseudo-randomly generated blocks within the data at random positions [18, 7], or compute an authentication tag for each block [5, 23]. In the first category of solutions, named as *watchdog-based* PoRs, $\mathcal{DO}$ embeds pseudo-randomly generated "watchdogs" [7] or "sentinels" [18] and further encrypts the data to make them indistinguishable from original data blocks. During the verification phase, $\mathcal{DO}$ retrieves a subset of these watchdogs. Other solutions which during the setup phase compute an authentication tag for each data block, retrieve a subset of data blocks together with the tags and verify their integrity. Most of recent *tag-based* solutions [23, 6] employ homomorphic tags which during the verification phase, reduces the communication overhead: $\mathcal{DO}$ will query the $\mathcal{CS}$ to receive a linear combination of a selected subset of blocks together with the same linear combination of the corresponding tags.

2.2 Conflict between PoR and deduplication

In the last decade a number of solutions have been suggested to address PoR and deduplication as separate problems. Yet any attempt to come up with a monolithic cloud storage system based on a simple composition of PoR with deduplication seems to be doomed to fail due to some inherent conflict between current PoR and deduplication schemes.

The root cause of the conflict is that PoR and deduplication call for diverging objectives: PoR aims at imprinting the data with retrievability guarantees that are unique for each user whereas deduplication tries, whenever feasible, to factor several data segments submitted by different users into a unique copy kept in storage. Indeed, in the case of watchdog-based PoR schemes such as [18, 7] that insert pseudo-randomly generated sentinels or watchdogs in the outsourced data, simple composition with deduplication would fail because the pre-processing akin to this family of PoR schemes includes semantically secure encryption by each user that prevents the detection of duplicate data segments (see Fig. 1). In the case of tag-based PoR solutions such as [5, 23], the users append a tag to each data segment before storing them in the cloud. Since the tags are computed under a private key generated by each user, the tags stored with the duplicate copies of identical segments submitted by different users will still be different; and in case of deduplication, even if storing a single copy for all duplicate segments would be consistent with respect to basic data management, the PoR scheme would still require that the tags generated by each user for the deduplicated data segment be kept separately in storage (see Fig. 2).

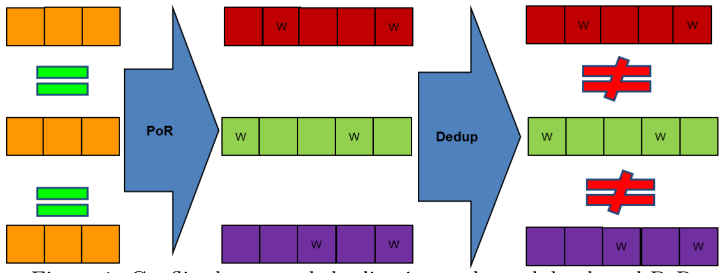

Figure 1: Conflict between deduplication and watchdog-based PoR

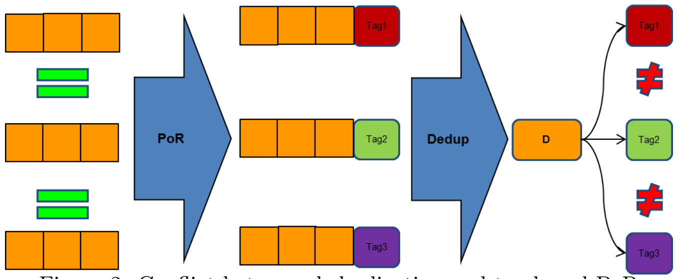

Figure 2: Conflict between deduplication and tag-based PoR

The additional storage resulting from these tags increase very rapidly with the number of users sharing the same data. In the private PoR scheme described in [23] with a split of 160 blocks of size 80 bits, the storage overhead resulting from PoR tags of 80 bits each, is 0.625% per user. For example, a 4GB file uploaded by 100 users would require 62.5% additional storage, that is 2.5GB whereas if tags are to be deduplicated this overhead would remain constant and equal to 25.6MB only.

3. MESSAGE-LOCKED POR

3.1 Idea

In this section we sketch the idea of a generic approach that solves the conflict between PoR and deduplication paving the way for a simple integration of PoR and deduplication within the same cloud storage system. The root cause of the conflict is the difference in the way duplicate data segments submitted by different customers are handled by PoR and deduplication, namely, the fact that deduplication summarizes all duplicates into a single copy whereas PoR requires that every duplicate segment be kept separately in order to preserve the user-specific effect of PoR on each duplicate. Consequently, a new approach aiming at achieving identical PoR effects on duplicates submitted by different customers seems to be the right solution for a simple composition of PoR with deduplication. Further along the same direction, the new approach should assure that PoR's effect on each data segment depends on the value of the data segment regardless of the difference in the identity of the customers submitting the data segments, in other words, these PoR operations should be a function of the data segment's value independently of the customer's identity. We thus define such PoR schemes that would be compatible with deduplication as *message-locked proofs of retrievability.*

A straightforward technique to implement *message-locked* PoR consists of defining the PoR operation on the data segment as a one-way function of the data segment. Without loss of generality existing PoR schemes can be represented by $P(d, K)$ as a function P of the data segment d and a key K that is determined by each data owner performing the PoR processing on d. In the case of tag-based PoR schemes K is the secret key used to produce the tags, whereas in PoR schemes relying on watchdogs K is the secret key used to generate the watchdogs and encrypt the data. Given a one-way hash function f, existing PoR schemes can be transformed to become *message-locked* by simply substituting K with $f(d)$ in the existing PoR function $P(d, K)$. Based on this technique any existing PoR scheme can be transformed into a *message-locked* PoR and as such be smoothly integrated with a deduplication function. Yet a new concern arises about the substitution of the secret key K with the result of a one-way hash function of the data segment. A similar question has been tackled when dealing with the problem of deduplication of encrypted data and several solutions ranging from convergent encryption [14] to schemes exploiting the popularity of data segments [25, 22] through server-based protocols [8, 21] have been suggested. Like *message-locked* PoR, these solutions also rely on a key derived from the data segment using some one-way function often called a *message-locked* key. The *message-locked* PoR transformation suggested in this paper can be based on any secure *message-locked* key generation technique. Therefore, for the sake of completeness, we review next, previous work on message-locked key generation protocols that were tailored for secure deduplication.

3.2 Message-Locked Key Generation for Secure Deduplication

The problem of secure *message-locked* key generation was already analyzed in the context of secure deduplication. Indeed, cross-user deduplication also raises an inherent incompatibility with data confidentiality: The semantically secure encryption of a same data segment performed by different data owners ensues completely different results for the same original copies.

To show that deduplication and encryption can coexist, authors in [14] introduce the concept of convergent encryption whereby all existing copies of a file is encrypted with the same encryption key: the encryption key simply is the hash of the file. While this initial approach where the encryption key is derived from the file itself seems ideal to achieve storage efficiency and data confidentiality at the same time, it unfortunately suffers from several weaknesses including dictionary attacks during which a cloud storage server $\mathcal{CS}$ can try to guess the file. A curious $\mathcal{CS}$ can compute the hash of potential candidates of a given file, derive an encryption key, and check whether the encryption of the file with this key is actually stored at its premises.

To thwart this type of attacks, [8] introduces a key server $\mathcal{KS}$. The idea is that every time a user wants to upload a file it will engage in an instance of an oblivious pseudo-random function (OPRF) protocol [11] with $\mathcal{KS}$ to generate a secret key with which the file to be uploaded is going to be encrypted. In order to ensure that users owning the same file agree on the same key, the user's input to the OPRF will depend on the file. As a result, the proposed scheme puts an end to offline attacks and forces the cloud storage server $\mathcal{CS}$ (or a user) to contact $\mathcal{KS}$ whenever it makes a guess for an uploaded file. This allows $\mathcal{KS}$ to implement rate-limiting measures to restrict the number of queries $\mathcal{CS}$ can issue, and as a by product limit the number of dictionary attacks $\mathcal{CS}$ can conduct in a given period of time.

Building on the same idea, the authors of [4] propose ClearBox which is a transparent storage service that provides privacy-preserving deduplication while making sure that users are charged only for the storage they actually use. The main contributions of ClearBox are twofold: Replace OPRF with a BLS blind signature [10] to reduce the communication and the computation cost of the key generation protocol; and combine Merkle trees and time-dependent randomness (obtained by leveraging some bitcoin functionalities) to ensure that the amount of money users pay is proportional to the rate of deduplication of their uploaded files.

While the work of [8, 4] succeeds in circumventing offline dictionary attacks by $\mathcal{CS}$, they are both prone to offline dictionary attacks by $\mathcal{KS}$. To address this issue, Liu et al. [19] devised a solution that allows users to agree on a secret key without connecting to a key server. The proposed solution relies on additively homomorphic encryption, password-authenticated key exchange (PAKE) and low-entropy hash to empower users uploading the same file to derive the same encryption key. Furthermore by using low-entropy hash, the scheme in [19] is able to have a rate-limiting strategy per file, which offers better security guarantees than [8, 4] that deploy rather a rate-limiting strategy per user.

As stated in the previous section, any of these solutions can be used as a building block to achieve message-locked PoR, nevertheless, we propose a different approach that does not rely on a single key server, but without the complexity of peer-to-peer systems. The idea is to have the data owner $\mathcal{DO}$ interact with both cloud server $\mathcal{CS}$ and key server $\mathcal{KS}$ to generate the message-locked secret key that will later be used as input to the message-locked PoR. The new solution will be described in Section 4.3.

4. MESSAGE LOCKED POR - SOLUTION

4.1 Overview

We consider a cloud storage model that comprises a number of affiliated data owners who are interested in securely outsourcing their files to a cloud storage server $\mathcal{CS}$. Moreover, these data owners wish to take advantage of the benefits of cross-user file-level deduplication performed by $\mathcal{CS}$ (e.g., reduced storage costs) whilst still being assured of the integrity of their outsourced data. The latter goal is achieved through proofs of retrievability which, as discussed earlier, offer a data owner $\mathcal{DO}$ cryptographic guarantees on the correct storage of her outsourced data at the cost of a pre-processing (setup) phase during which algorithms Keygen and Encode are executed to prepare the $\mathcal{DO}$'s files for upload. Generally, the Encode algorithm consists of combining error-correcting codes (ECC) with cryptographic primitives such as encryption and MACs to build a verifiable version of the file to be uploaded.

This entails that in order to allow $\mathcal{CS}$ to deduplicate the outsourced files, data owners uploading the same file should provide algorithm Encode with the same input. Notably, the data owners should agree on the secret keys of the cryptographic functions. To that effect, we propose a new ML-KeyGen algorithm that allows a data owner $\mathcal{DO}$ with some file F to generate a key K_F by communicating with a key server $\mathcal{KS}$ and cloud storage server $\mathcal{CS}$ such that K_F is generated using a one-way function of file F, and the secret keys of $\mathcal{KS}$ and $\mathcal{CS}$ (cf. Section 4.3). Thanks to ML-KeyGen, we can transform any PoR scheme into a message-locked PoR by applying minor changes to the Encode algorithm. This provides data owners with secure means to verify the integrity of their outsourced files while at the same time saving storage (via deduplication) at the cloud storage server.

As to what type of deduplication to use, ML-PoR goes with server-side deduplication. We recall that in server-side deduplication, the data owners upload their files to $\mathcal{CS}$, which in turn performs the deduplication. Client-side deduplication on the other hand lets data owners upload their files only if copies of these files are not already stored. Although in terms of bandwidth, client-side deduplication is generally more efficient, it requires that each data owner uploading an already existing file engages in a Proof of Ownership (PoW) protocol [16] with $\mathcal{CS}$. While to simplify the presentation we opt for server-side deduplication, we note that our solution marries well with any existing PoW scheme, as will be discussed in Section 7.

In subsequent sections, we describe ML-KeyGen, our protocol for message-locked generation and how to change the Encode algorithm of any PoR scheme to get a new algorithm ML-Encode that is message-locked (i.e the same input file results in the same encoding).

4.2 Threat model

Since the solution uses a server aided key generation solution for deduplication, similarly to all previously proposed server-aided encryption solutions, it assumes that the cloud server $\mathcal{CS}$ does not collude with the key server $\mathcal{KS}$. Furthermore, cloud users are assumed not to collude either with the cloud server or with the key server. Therefore, the only information the cloud server can have access to is users' encoded (encrypted) data and the messages exchanged during the message-locked key generation protocol.

4.3 ML-Keygen: Server-aided message-locked key generation

In this section, we describe a new server-aided message-

Data Owner		Key Server	Cloud Storage Server
$h \leftarrow H^*(F)$			
$\alpha, \beta \xleftarrow{R} \mathbb{Z}_p^*$			
$\hat{h} \leftarrow h \cdot g_1^{\alpha}$			
	$\xrightarrow{\hat{h}}$		
		$\hat{s} \leftarrow (\hat{h})^{\kappa}$	
	$\xleftarrow{\hat{s}}$		
$s \leftarrow \hat{s} \cdot y_{KS,1}^{-\alpha}$			
$\tilde{s} \leftarrow s \cdot g_1^{\beta}$			
	$\xrightarrow{\tilde{s}}$	$\longrightarrow$	
			$\tilde{c} \leftarrow (\tilde{s})^{\gamma}$
	$\xleftarrow{\tilde{c}}$	$\longleftarrow$	
$c \leftarrow \tilde{c} \cdot y_{CS,1}^{-\beta}$			
ret: $K_F \leftarrow H(c)$			

Figure 3: ML-KeyGen- Protocol Description

locked key generation protocol, ML-KeyGen, that will be used by ML-PoR to generate the keying material for PoR. Compared to related work, this new solution offers better security guarantees, as it protects against dictionary attacks that could be launched by the key server as well. ML-KeyGen extends the solution in [4] by generating the *message-locked* key using two secret keys each of them generated by $\mathcal{KS}$ and $\mathcal{CS}$ respectively. Thanks to this solution, neither $\mathcal{KS}$ nor $\mathcal{CS}$ can solely mount dictionary attacks.

The proposed protocol is executed among a data owner $\mathcal{DO}$, the key server $\mathcal{KS}$ and the cloud storage server $\mathcal{CS}$. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two groups of prime order p with g_1 and g_2 as their respective generators, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear pairing. We also define two cryptographic hash functions $H^* : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $H : \mathbb{G}_1 \rightarrow \{0,1\}^{\tau}$ with τ being a security parameter. During a setup phase, $\mathcal{KS}$ and $\mathcal{CS}$, respectively choose a private key $\kappa \in \mathbb{Z}_p^*$ and $\gamma \in \mathbb{Z}_p^*$ and publish their corresponding public keys ($y_{\mathsf{KS},1} = g_1^{\kappa}$, $y_{\mathsf{KS},2} = g_2^{\kappa}$) and ($y_{\mathsf{CS},1} = g_1^{\gamma}$, $y_{\mathsf{CS},2} = g_2^{\gamma}$). As depicted in Figure 3, $\mathcal{DO}$ computes an initial *message-locked* key h for file F by simply computing the hash of F: $h \leftarrow H^*(F)$. This key is further blinded using a pseudo randomly generated value $\alpha \in \mathbb{Z}_p^*$ and the result denoted by $\hat{h}$ is sent to $\mathcal{KS}$. Thanks to the underlying blinded signature, $\mathcal{KS}$ computes the signature of $\hat{h}$ using its private key κ. Upon reception of this blinded signature, data owner $\mathcal{DO}$ unblinds this value to derive the signature of h. Hence: $s = h^{\kappa} g_1^{\alpha\kappa} g_1^{-\alpha\kappa}$. Once s is successfully verified by checking if $e(s, g_2) = e(h, y_{\mathsf{KS},2})$, $\mathcal{DO}$ blinds s using a second random value $\beta \in \mathbb{Z}_p^*$ and sends the result $\tilde{s}$, this time, to cloud server $\mathcal{CS}$. Similarly to $\mathcal{KS}$, $\mathcal{CS}$ signs $\tilde{s}$ and returns this signature to $\mathcal{DO}$. $\mathcal{DO}$ in turn, unblinds $\tilde{c}$ to derive the signature $c = h^{\kappa\gamma} g_1^{\beta\gamma} g_1^{-\beta\gamma}$ and verifies the signature using $y_{CS,2}$. If the verification succeeds, the *message-locked* key K_F equals $H(c) = H(h^{\kappa\gamma})$. Thanks to this new key generation solution neither party can perform offline dictionary attacks. More details on the security of ML-KeyGen are provided in Section 6.1.

4.4 ML-Encode: Message-Locked POR Encoding

Generally speaking, the preprocessing step of PoR schemes consists of the following operations: (i) applying an error-correcting code to the file to be uploaded so as to allow $\mathcal{DO}$ to recover from accidental errors; (ii) encrypting and permuting the file to hide the dependencies between data blocks and redundancy (ECC) blocks; (iii) incorporating some integrity values to authenticate the file. It follows that for message-locked PoR to work, users should not only generate the same secret for the same file, but also agree on the the ECC parameters, the encryption and the permutation algorithms and the integrity mechanisms. Accordingly in our scheme, we let $\mathcal{KS}$ choose and advertise these parameters and algorithms before any data owner joins the system. In this manner, we ensure that the encoding operation yields the same output for a fixed input file regardless of the data owner carrying it out.

To summarize, ML-Encode runs in the same way as a regular Encode, except for the following: The secret keys are generated using ML-KeyGen and all the parameters related to PoR (e.g. ECC algorithm and cryptographic functions) are provided by $\mathcal{KS}$.

5. ML-POR INSTANTIATIONS

To illustrate the feasibility of *message-locked* PoRs, we describe in what follows two instantiations that build upon the PoR schemes from [7, 23]. Before the detailed description of our instantiations, we note that like a regular PoR a message locked PoR comprises five algorithms: ML-KeyGen, ML-Encode, ML-Challenge, ML-ProofGen and ML-ProofVerif. We note that algorithms ML-Challenge, ML-ProofGen and ML-ProofVerif are executed in the same way as their counterparts in the original protocols. Therefore, in the following we only describe ML-KeyGen and ML-Encode algorithms. The interested reader may refer to [7, 23] for more details on the original algorithms.

5.1 ML-Stealthguard: A message-locked PoR scheme based on watchdogs

This section describes a *message-locked* PoR that extends an existing watchdog-based solution named StealthGuard [7]. In StealthGuard, data retrievability is achieved thanks to the oblivious insertion of pseudo-random generated blocks

Table 1: ML-StealthGuard's Public Parameters

Public Parameter	Description
τ	security parameter of StealthGuard
l	size of a block
m	number of blocks in a split S_i
v	number of watchdogs in one split
ρ	ECC code rate $\rho = \frac{m}{m+d}$
(D, m, d)-ECC	ECC correcting up to $\frac{d}{2}$ errors per split
$\mathsf{Enc} : \{0,1\}^\tau \times \{0,1\}^* \to \{0,1\}^*$	encryption algorithm
$\Phi : \{0,1\}^\tau \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^l$	watchdog generator
$\Pi_F : \{0,1\}^\tau \times [\![1, n \cdot D]\!] \to [\![1, n \cdot D]\!]$	pseudo-random file-level permutation
$\Pi_S : \{0,1\}^\tau \times [\![1, D+v]\!] \to [\![1, D+v]\!]$	pseudo-random split-level permutation
$H_{\mathsf{permF}} : \{0,1\}^* \to \{0,1\}^\tau$	file-level permutation key generator
$H_{\mathsf{enc}} : \{0,1\}^* \to \{0,1\}^\tau$	encryption key generator
$H_{\mathsf{wdog}} : \{0,1\}^* \to \{0,1\}^\tau$	watchdog key generator
$H_{\mathsf{permS}} : \{0,1\}^* \times [\![1, n]\!] \to \{0,1\}^\tau$	split-level permutation key generator

named *watchdogs*. Furthermore, StealthGuard leverages a privacy-preserving word search scheme in order to query the watchdogs without leaking any information about their value or their position within the data.

5.1.1 Overview of StealthGuard

A data owner $\mathcal{DO}$ wishing to outsource a file F proceeds as follows:

- $\mathcal{DO}$ calls the algorithm KeyGen to derive a secret key K that is used to process F before outsourcing it to cloud server $\mathcal{CS}$ as well as to verify its retrievability later.
- $\mathcal{DO}$ then calls algorithm Encode, which is the algorithm that prepares file F for upload. Hence, algorithm Encode applies ECC to F, and then using secret key K determines the value and location of the watchdogs within the encoded file and permutes and encrypts the file.

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

5.1.2 ML-StealthGuard

In order for StealthGuard to become compatible with secure deduplication, it needs to be slightly modified such that all data owners who outsource the same file F to cloud storage server $\mathcal{CS}$ compute *the same* verifiable version of that file. In particular, we make StealthGuard *message-locked* by leveraging the newly proposed ML-KeyGen. Additionally, $\mathcal{KS}$ publishes some parameters which should be common for all data owners in the system in order for ML-Encode to be deterministic and return the same output for each of its execution over the same file. These public parameters are listed in Table 1.

Given file F, $\mathcal{DO}$ first runs the ML-KeyGen protocol to derive a *message-locked* key K_F from F. Subsequently, with secret key K_F and the public parameters advertised by $\mathcal{KS}$, $\mathcal{DO}$ invokes ML-Encode that performs the following operations:

1. *Error-correction*: As a first step, ML-Encode divides file F into n splits $\{S_1, S_2, ..., S_n\}$ where each split S_i with $1 \leq i \leq n$, regroups m blocks of size l. ML-Encode further applies the error-correcting code published by $\mathcal{KS}$ to each file split S_i. This yields a new encoded file $\dot{F}$.

2. *File block permutation*: At this step, ML-Encode first generates a permutation key K_{permF} such that $K_{\mathsf{permF}} = H_{\mathsf{permF}}(K_F)$. This key and the published pseudo-random permutation Π_F are used to shuffle all the blocks in file $\dot{F}$. Let $\ddot{F}$ denote the resulting file.

3. *File encryption*: Given secret key K_F, ML-Encode derives an encryption key $K_{\mathsf{enc}} = H_{\mathsf{enc}}(K_F)$, and with this encryption key ML-Encode further encrypts the blocks in file $\ddot{F}$ using the semantically secure encryption algorithm Enc published by $\mathcal{KS}$. We denote by $\tilde{F}$ file $\ddot{F}$ after encryption.

4. *Watchdog insertion*: ML-Encode computes a watchdog generation key $K_{\mathsf{wdog}} = H_{\mathsf{wdog}}(K_F)$ and uses this key and the published pseudo-random function Φ to generate $n \times v$ watchdogs $w_{ij} = \Phi(K_{\mathsf{wdog}}, i, j)$ for $1 \leq i \leq n$ and $1 \leq j \leq v$. Since the watchdogs are pseudo-randomly generated and the blocks in the splits are encrypted using a semantically secure encryption, $\mathcal{CS}$ cannot differentiate between watchdogs and data blocks. Once all watchdogs are generated, ML-Encode appends v watchdogs $\{w_{i1}, ..., w_{iv}\}$ to each $\tilde{S}_i$[1] in $\tilde{F}$. Each split is permuted using their corresponding newly generated permutation key $K_{\mathsf{permS,i}} = H_{\mathsf{permS}}(K_F, i)$ and the published pseudo-random permutation Π_S. Without loss of generality, we denote $\hat{F}$ file $\tilde{F}$ after the insertion of watchdogs.

At the end of its execution, algorithm ML-Encode returns file $\hat{F}$. Thereupon, $\mathcal{DO}$ selects a unique file identifier fid for file $\hat{F}$ and outsources file $\hat{F}$ together with identifier fid to $\mathcal{CS}$. Finally, $\mathcal{DO}$ securely keeps the secret key K_F and file identifier fid in her local storage.

Using secret key K_F, file identifier fid and the public parameters of the system, $\mathcal{DO}$ is able to run the algorithms ML-Challenge and ML-ProofVerif (which are the same as the

[1]We note here that the size of split $\tilde{S}_i$ is $D = m + d - 1$, where d is the number of redundancy blocks of the ECC algorithm.

Table 2: ML-CompactPoR's Public Parameters.

Public Parameter	Description
τ	security parameter of Compact PoR
l	size of a block
m	number of blocks in a split S_i
p	large prime with $\lvert p \rvert \approx l$
ρ	ECC code rate $\rho = \frac{m}{m+d}$
(D, m, d)-ECC	ECC correcting up to $\frac{d}{2}$ errors per split
$\mathsf{Enc} : \{0,1\}^\tau \times \{0,1\}^* \rightarrow \{0,1\}^*$	encryption algorithm
$\Pi_F : \{0,1\}^\tau \times [\![1, n \cdot D]\!] \rightarrow [\![1, n \cdot D]\!]$	pseudo-random file-level permutation
$f : \{0,1\}^\tau \times \{0,1\}^* \rightarrow \mathbb{Z}_p$	pseudo-random function
$H_{\mathsf{perm}} : \{0,1\}^* \rightarrow \{0,1\}^\tau$	file-level permutation key generato
$H_{\mathsf{enc}} : \{0,1\}^* \rightarrow \{0,1\}^\tau$	encryption key generator
$H_{\mathsf{mac}} : \{0,1\}^* \rightarrow \{0,1\}^\tau$	MAC key generator
$H_\alpha : \{0,1\}^* \times [\![1, m]\!] \rightarrow \mathbb{Z}_p$	MAC coefficient generator

original Challenge and ProofVerif of StealthGuard) to check the retrievability of her outsourced file F.

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

5.2 ML-CompactPoR: A message-locked PoR scheme based on homomorphic tags

As in the case of watchdog-based PoRs, tag-based PoR schemes need to be adapted in order to be compatible with secure deduplication. To show how a tag-based PoR can be transformed into a *message-locked* PoR, we instantiate the private Compact POR scheme proposed in [23].

5.2.1 Overview of Compact PoR

A data owner $\mathcal{DO}$ wishing to outsource a file F proceeds as follows:

- $\mathcal{DO}$ calls algorithm KeyGen in order to generate a secret key K that will be used to prepare F for upload and to verify its retrievability later.
- $\mathcal{DO}$ then calls algorithm Encode, which is in charge of preparing the file for upload. Accordingly, algorithm Encode applies ECC to F and then uses the secret key K to encrypt, shuffle and authenticate the output of the ECC. The authentication consists of generating a homomorphic MAC for each split within the file.

At any time $\mathcal{DO}$ wishes to check the retrievability of file F, she calls the Challenge algorithm to generate a query chal that includes the indices of randomly-chosen splits together with some randomly-generated coefficients. On receiving the challenge chal, cloud storage server $\mathcal{CS}$ calls algorithm ProofGen to generate a proof of retrievability. Such a proof consists of a linear combination of the randomly chosen splits using the randomly-generated coefficients and the corresponding MAC. The latter is computed by applying the same linear combination over each split's homomorphic tag.

$\mathcal{DO}$ decides that F is retrievable if the proof produced by $\mathcal{CS}$ is correctly formed: She uses her secret key to check if the MAC in the proof successfully authenticates the linear combination of the file splits.

5.2.2 ML-CompactPoR

Similarly to StealthGuard, the original compact PoR becomes compatible with deduplication whenever it uses a *message-locked* key together with some other parameters that are common for all users in the system. Therefore ML-CompactPOR builds upon the newly proposed ML-KeyGen and assumes that all users fetch the public parameters from a key server $\mathcal{KS}$ that are listed in Table 2.

Assume that data owner $\mathcal{DO}$ intends to outsource a file F. Accordingly, $\mathcal{DO}$ prepares F for upload by first invoking the ML-KeyGen protocol. Without loss of generality, we denote K_F the resulting *message-locked* key.

Given secret key K_F and the public parameters advertised by $\mathcal{KS}$, $\mathcal{DO}$ calls algorithm ML-Encode that performs the following operations:

1. *Error-correction*: Algorithm ML-Encode applies the error-correcting code published by $\mathcal{KS}$ to file F. This yields file $\dot{F}$.

2. *File block permutation*: At this step, ML-Encode computes a permutation key $K_{\mathsf{perm}} = H_{\mathsf{perm}}(K_F)$ which together with the published pseudo-random permutation Π_F is used to permute all the blocks in file $\dot{F}$. Without loss of generality, we denote the resulting permuted file $\ddot{F}$.

3. *File encryption*: Having K_F, ML-Encode derives an encryption key $K_{\mathsf{enc}} = H_{\mathsf{enc}}(K_F)$, and uses this encryption key and the semantically secure encryption algorithm Enc published by $\mathcal{KS}$ to encrypt the blocks in file $\ddot{F}$. Let $\hat{F}$ denote the encrypted file.

4. *Tag generation*: $\hat{F}$ is further divided into n equally-sized splits each comprising m blocks. We denote $\hat{b}_{ij}$ the j^{th} block of the i^{th} split where $1 \leq i \leq n$ and $1 \leq j \leq m$. ML-Encode then generates a MAC key $K_{\mathsf{mac}} = H_{\mathsf{mac}}(K_F)$ and generates m random numbers $\alpha_j = H_\alpha(K_F, j)$ where $1 \leq j \leq m$. Then, for each split $\hat{S}_i$, ML-Encode computes the following homomorphic MAC σ_i:

$$\sigma_i = f(K_{\mathsf{mac}}, i) + \sum_{j=1}^{m} \alpha_j \hat{b}_{ij}$$

Once ML-Encode is executed, $\mathcal{DO}$ picks a unique file identifier fid for file $\hat{F}$ and uploads file $\hat{F}$ together with its identifier fid and the MACs $\{\sigma_1, ..., \sigma_n\}$. We note that $\mathcal{DO}$ keeps the secret key K_F and file identifier fid in her local storage.

Given secret key K_F, file identifier fid and the public parameters proposed by $\mathcal{KS}$, $\mathcal{DO}$ is able to run the algorithms ML-Challenge and ML-ProofVerif to verify the retrievability of her outsourced file F.

Notice that if there is another data owner $\mathcal{DO}'$ wishing to outsource the same file F, $\mathcal{CS}$ will easily detect the duplicate copy if she executes the proposed ML-CompactPOR and consequently remove it to save storage space. Thanks to the deterministic nature of the underlying algorithms, $\mathcal{DO}'$ will still be able to check the retrievability of F.

6. SECURITY ANALYSIS

Based on previous work on message-locked key generation [8, 4], we sketch the security analysis of ML-KeyGen and further briefly show that a *message-locked* PoR assures storage correctness as long as the *message-locked* key is not compromised.

6.1 Message-Locked Key Generation

The work of [8, 4] on secure deduplication relies on the assumption that the key server $\mathcal{KS}$ is not interested in learning the content of the files kept at the cloud storage server $\mathcal{CS}$; and thus the key generation protocol only involves the user –having as input the file to be uploaded– and $\mathcal{KS}$–having as input its secret key. Yet, it is easy to see that there is no real countermeasure to deter $\mathcal{KS}$ from running offline dictionary attacks to compromise the confidentiality of the uploaded files. Our solution addresses this problem by having $\mathcal{CS}$ take part in the ML-KeyGen forcing as a result $\mathcal{KS}$ to go online and connect to the $\mathcal{CS}$ whenever it makes a guess for an outsourced file. Luckily, such online attacks can be obstructed using rate-limiting measures that bound the number of ML-KeyGen runs (and therewith the number of online attacks) that a given user can initiate within a given time period. Hence, as long as $\mathcal{CS}$ and $\mathcal{KS}$ do not collude, none of them can perform offline dictionary attacks.

6.2 Message-Locked PoR

6.2.1 *Security Guarantees of PoR*

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

- hiding the dependencies between ECC blocks and data blocks through semantically-secure encryption and secure permutations;
- authenticating the outsourced files either by inserting (unforgeable) watchdogs (cf. [7, 18]) or by computing (unforgeable) tags (cf. [23, 26]).

6.2.2 *Security of Message-Locked PoR and File Unpredictability*

Note that the security of our *message-locked* PoR is assured so long as the key derived in the key generation phase is not compromised. By having access to the secret key used during upload, the $\mathcal{CS}$ not only can compromise the confidentiality of the file, but can also corrupt the uploaded file without being detected. Notably, $\mathcal{CS}$ can mount the following types of attacks:

- Use the secret key to find the dependencies between ECC blocks and data blocks by decrypting the file and inverting the permutations used to shuffle the blocks. In this fashion, $\mathcal{CS}$ can corrupt the file in such a way that the file becomes irretrievable while ensuring that the probability of detecting the tampering is negligible.
- In the case of [7, 18], the secret key enables $\mathcal{CS}$ to discover which blocks are data blocks and which are watchdogs, and accordingly, it can keep only the watchdogs and get rid of the data blocks.
- In the case of [23, 26], given the secret key $\mathcal{CS}$ can modify the data block and compute the corresponding tags correctly.

Taking these attacks into account, we conclude that similarly to previous work on secure deduplication, the security of our scheme is closely tied to how well $\mathcal{CS}$ guesses the content of the uploaded file (i.e. the unpredictability of the uploaded file). Nevertheless thanks to our ML-KeyGen protocol, $\mathcal{CS}$ cannot run offline dictionary attacks, as it must go online and connect to the key server $\mathcal{KS}$ to generate the secret key and test the correctness of its guesses.

6.3 Rate Limiting

The confidentiality and the integrity guarantees of our scheme depend on the *unpredictability of the file* and *the number of message-locked key generation queries* a user is allowed to issue. Intuitively, the more predictable the file is, the less the number of key generation queries an adversary (e.g. storage server $\mathcal{CS}$) has to make to divulge its content; inversely, the more key generation queries an adversary makes, the more predictable the file becomes (by ruling out the files that do not match). It follows that in order to contain the threat of online attacks, it is important to limit the number of such queries a given user makes. This in reality will be implemented through authentication mechanisms: Both $\mathcal{CS}$ and $\mathcal{KS}$ will authenticate and identify (and therewith keep track of) users submitting upload queries. Still as rightly pointed out by [19], $\mathcal{CS}$ or $\mathcal{KS}$ for instance can masquerade as any number of users voiding thus the benefits of any rate limiting countermeasure. This illustrates that in order for rate limiting to work, we need to put in place mechanisms to verify the identity of users engaging in the key derivation protocol. One approach to achieve this is to link the user's identity to the user's IP address; however, this approach can be insufficent if users mount spoofing attacks. A more secure alternative, albeit more costly is to have an *identity manager* – as in [25] – that verifies the identity of the users and provides these with the credentials necessary to operate the functionalities of the storage service.

7. PERFORMANCE EVALUATION

Thanks to the newly proposed *message-locked* PoR scheme, $\mathcal{CS}$ succeeds in saving storage space using deduplication techniques and data owners are provided with some guarantees with respect to the integrity of their data. The generation of a *message-locked* key and the use of deterministic operations for the pre-processing (encoding) of the outsourced files enable $\mathcal{CS}$ to discover redundant data and further perform deduplication while still being able to produce cryptographic proofs for data retrievability. The only additional cost added by the newly designed ML-PoR schemes, compared with their original versions is the one originating from the server-aided message-locked key generation protocol which is mandatory to ensure the security of PoR. Concerning the server-aided message-locked key generation protocol, the proposed ML-KeyGen protocol, compared to existing solutions [8, 4], relaxes the trust towards $\mathcal{KS}$ at the cost of one more communication round. However, in both cases the overhead of the key generation protocol can be considered as minimal compared to the computational cost of the remaining algorithms.

The currently proposed *message-locked* PoR schemes can achieve even more savings by implementing a client-side deduplication strategy whereby a cloud customer uploads a file F only if not already stored. Consequently, both cloud customers and the cloud provider benefit from bandwidth savings. To enable client-side deduplication, a data owner $\mathcal{DO}$ only needs to execute ML-KeyGen and ML-Encode and to check whether the resulting encoded file is already stored at $\mathcal{CS}$ (e.g. by hashing this file and comparing it with a hash table stored at $\mathcal{CS}$). However, as noted in [17], client-side deduplication is exposed to some attacks launched by potentially malicious users: An adversary that discovers the identifier of a file can claim possession of it. To circumvent such attacks, solutions in the literature [13, 4] propose to combine client-side deduplication mechanisms with proofs of ownership (PoW) [16] which help $\mathcal{CS}$ verify that a user *owns* a file without the need to upload it.

A solution implementing client-side deduplication using PoW, would require $\mathcal{DO}$ to first execute the most costly algorithm of *message-locked* PoR (namely, ML-Encode), and thereafter to prove to $\mathcal{CS}$ the ownership of the verifiable file $\hat{F}$. Given that the computational cost of current PoW schemes is linear to the size of the file (see Table 2 in [15]), data owners have to consider the trade-off between the bandwidth savings and the computational overhead of PoW before deciding to use client-side deduplication.

When using client-side deduplication, one might argue that if data owner $\mathcal{DO}$ does not upload the file, then she should not be burdened with the execution of the ML-Encode algorithm, and instead compute the proof of ownership using the original file F. Unfortunately, this approach would break the security of PoR since to be able to verify a PoW, $\mathcal{CS}$ needs to process the original file F. Hence, $\mathcal{CS}$ would immediately derive the secret *message-locked* key K_F, thus, a PoW should be computed using the verifiable version $\hat{F}$.

8. RELATED WORK

Authors in [12] propose a secure data deduplication mechanism and observe that it is inherently compatible with the PoR scheme proposed in [27] without providing any details.

Moreover, in [28], a new proof of storage with deduplication (POSD) was introduced whereby thanks to a publicly verifiable proof of data possession scheme, users can verify the correct storage of deduplicated data using the public key of the first user actually storing the data. This solution has been proved insecure in [24] and additionally does not prevent the cloud from cheating.

Very recently, the authors in [3] introduce a multi-tenant PoR framework that marries well with deduplication and propose a solution that relies on the homomorphic properties of the signature in [23]. Contrary to our solution, the proposed scheme considers a stronger and more realistic security model in which users can be corrupted by the cloud server. This however comes at a higher cost at the user side in terms of bandwidth and computation; namely, the verification complexity of the storage of a file grows linearly with the number of users outsourcing that file.

9. CONCLUSION

In this paper, we identified an inherent incompatibility of existing proof of retrievability schemes with data deduplication which, nowadays, is inevitable to real-life cloud storage systems to maximize their storage space savings. Similarly to secure deduplication solutions, we proposed the *message-locked* PoR approach which makes sure that all algorithms in a PoR scheme are deterministic and therefore enables file-based deduplication. The two described instantiations of existing PoR solutions (ML-StealthGuard and ML-CompactPOR) mainly implement a new encoding algorithm, ML-Encode, that basically differs from the original one in one aspect: instead of pseudo-randomly generated, the required keying material is derived from the file itself. Thus, for a given file, ML-Encode will always output the same encoded file irrespective of the identity of the cloud customer executing it. As such *message-locked* keys are exposed to dictionary attacks that could be launched by potentially malicious cloud providers, the proposed ML-PoR solutions initially call for a server-aided key generation technique (ML-KeyGen) which helps in protecting the secrecy of such keys. Thanks to the newly proposed ML-KeyGen solution that involves both the key server and the cloud provider, none of these parties can discover the *message-locked* key alone.

10. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers. This work was partially supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

11. REFERENCES

[1] Amazon drive. https://www.amazon.com/gp/drive/landing/everything/\\buy?tag=bisafetynet-20. Accessed: 2016-09-15.

[2] Google Drive. https://apps.google.com/driveforwork/. Accessed: 2016-09-15.

[3] F. Armknecht, J.-M. Bohli, D. Froelicher, and G. Karame. SPORT: Sharing Proofs of Retrievability

across Tenants. Cryptology ePrint Archive, Report 2016/724, 2016. http://eprint.iacr.org/2016/724.

[4] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent Data Deduplication in the Cloud. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 886–900, 2015.

[5] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 598–609, 2007.

[6] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and Privacy in communication networks (SecureComm)*, 2008.

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

[8] M. Bellare, S. Keelveedhi, and T. Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security (USENIX SEC)*, pages 179–194, 2013.

[9] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-Locked Encryption and Secure Deduplication. In *Proceedings of Eurocrypt*, pages 296–312, 2013.

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

[11] J. Camenish, G. Neven, and A. Shelat. Simulatable adaptive oblivious transfer. In *Proceedings of EUROCRYPT*, pages 573–590, 2007.

[12] R. Chen, Y. Mu, G. Yang, and F. Guo. Bl-mle: Block-level message-locked encryption for secure large file deduplication. *IEEE Transactions on Information Forensics and Security*, 10(12):2643–2652, 2015.

[13] R. di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

[14] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[15] L. Gonzales-Manzano and A. Orfila. An efficient confidentiality-preserving Proof of Ownership for deduplication. *Journal on Network and Computer Applications*, 50:49–59, 2015.

[16] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of ACM CCS*, pages 491–500, 2011.

[17] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. In *Proceedings of the 4th ACM International Wokshop on Storage Security and Survivability (StorageSS)*, 2008.

[18] A. Juels and B. S. K. Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 584–597, 2007.

[19] J. Liu, N. Asokan, and B. Pinkas. Secure Deduplication of Encrypted Data Without Additional Independent Servers. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 874–885, 2015.

[20] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14:1–14:20, 2012.

[21] P. Puzio, R. Molva, M. Önen, and S. Loureiro. ClouDedup: Secure Deduplication with Encrypted Data for Cloud Storage. In *PRoceeings of the , 5th IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM)*, 2013.

[22] P. Puzio, R. Molva, M. Önen, and S. Loureiro. PerfectDedup: Secure data deduplication. In *10th International Workshop on Data Privacy Management (DPM)*, 2015.

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

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

[25] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A Secure Data Deduplication Scheme for Cloud Storage. In *18th International Conference on Financial Cryptography and Data Security (FC)*, pages 99–118, 2014.

[26] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 229–238, 2012.

[27] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(5):847–859, 2011.

[28] Q. Zheng and S. Xu. Secure and efficient proof of storage with deduplication. In *Proceedings of the 2nd ACM conference on Data and Application Security and Privacy (CODASPY)*, 2012.

APPENDIX

A. STEALTHGUARD

In what follows, we briefly describe StealthGuard [7], a PoR scheme that achieves data retrievability thanks to the oblivious insertion of pseudo-random generated blocks named *watchdogs*.

- $\mathsf{KeyGen}(1^\tau) \rightarrow (K, \mathsf{param})$: Data owner $\mathcal{DO}$ calls this algorithm in order to generate a secret key K and a set of parameters param (such as ECC rate, size of split, etc.) that will be used to prepare F for upload and to verify its retrievability later.

- $\mathsf{Encode}(K, F) \rightarrow (\mathsf{fid}, \hat{F})$: $\mathcal{DO}$ calls this algorithm to prepare her file F for outsourcing to cloud server $\mathcal{CS}$. It applies ECC to F and divides the encoded file into equally-sized splits. Subsequently, Encode uses the secret key K to determine the value and position of the watchdogs, inserts them within the splits accordingly and further permutes and encrypts the file.
- $\mathsf{Challenge}(K, \mathsf{fid}) \rightarrow \mathsf{chal}$: $\mathcal{DO}$ calls this algorithm which chooses a split and a watchdog within this split and generates a challenge chal consisting of a privacy preserving search query for the selected watchdog.
- $\mathsf{ProofGen}(\mathsf{fid}, \mathsf{chal}) \rightarrow \mathcal{P}$: Upon receiving the challenge chal $\mathcal{CS}$ invokes this algorithm which processes the relevant split and generates a proof $\mathcal{P}$ that consists of a correct response to the search query. Thanks to the underlying privacy preserving word search scheme, $\mathcal{CS}$ can never learn the content of the query (hence discover the watchdog) and the corresponding response.
- $\mathsf{ProofVerif}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P}) \rightarrow b \in \{0, 1\}$: $\mathcal{DO}$ calls this algorithm which processes the received proof $\mathcal{P}$ and outputs a bit equal to 1 if the proof is valid or 0 otherwise. Only $\mathcal{DO}$ can verify this proof using secret key K.

In order to decide if F is retrievable, $\mathcal{DO}$ needs to issue at least γ PoR queries (γ depending on param) from randomly selected splits: if $\mathcal{DO}$ receives γ valid PoR responses then, she concludes that F can be retrieved, otherwise, she concludes that $\mathcal{CS}$ has corrupted part of the file.

B. PRIVATE COMPACT POR

In what follows, we briefly describe the private compact PoR scheme proposed in [23]. This scheme takes advantage of a pseudo-random function f.

- $\mathsf{KeyGen}(1^\tau) \rightarrow (K, \mathsf{param})$: Data owner $\mathcal{DO}$ calls this algorithm in order to generate a secret key K and a set of parameters param (such as ECC rate, size of split, etc.) that will be used to prepare F for upload and to verify its retrievability later.
- $\mathsf{Encode}(K, F) \rightarrow (\mathsf{fid}, \hat{F})$: $\mathcal{DO}$ calls this algorithm to prepare her file F for outsourcing to cloud server $\mathcal{CS}$. It applies ECC to F and then uses the secret key K to permute and encrypt the encoded file, and further outputs the result $\hat{F}$. Subsequently, Encode divides $\hat{F}$ in n equally-sized splits each comprising m blocks. We denote $\hat{b}_{ij}$ the j^{th} block of the i^{th} split where $1 \leq i \leq n$ and $1 \leq j \leq m$. Encode then generates a MAC key K_{mac}, chooses m random numbers α_j where $1 \leq j \leq m$ and computes for each split the following homomorphic MAC σ_i:

$$\sigma_i = f(K_{\mathsf{mac}}, i) + \sum_{j=1}^{m} \alpha_j \hat{b}_{ij}$$

  $\hat{F}$ together with the homomorphic MACs $\{\sigma_i\}$, $1 \leq i \leq n$, is then outsourced to $\mathcal{CS}$.
- $\mathsf{Challenge}(K, \mathsf{fid}) \rightarrow \mathsf{chal}$: $\mathcal{DO}$ calls this algorithm which outputs a challenge chal, consisting of a random l-element set $I \subset [1, n]$ and l random coefficients v_i.
- $\mathsf{ProofGen}(\mathsf{fid}, \mathsf{chal}) \rightarrow \mathcal{P}$: Upon receiving the challenge chal, $\mathcal{CS}$ invokes this algorithm which computes the proof $\mathcal{P} := (\beta_j, \sigma)$, for $1 \leq j \leq m$ as follows:

$$\beta_j = \sum_{(i,v_i)\in\mathsf{chal}} v_i \hat{b}_{ij} \ , \ \sigma = \sum_{(i,v_i)\in\mathsf{chal}} v_i \sigma_i$$

- $\mathsf{ProofVerif}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P}) \rightarrow b \in \{0, 1\}$: $\mathcal{DO}$ calls this algorithm which checks if the received proof $\mathcal{P}$ is well formed as follows:

$$\sigma \stackrel{?}{=} \sum_{(i,v_i)\in\mathsf{chal}} v_i f(K_{\mathsf{mac}}, i) + \sum_{j=1}^{m} \alpha_j \beta_j$$

  ProofVerif outputs a bit equal to 1 if the proof valid or 0 otherwise.

Thanks to the unforgeability of homomorphic MACs, a malicious cloud cannot corrupt a file without being detected.