An efficient confidentiality-preserving Proof of Ownership for Deduplication

Lorena González-Manzano, Agustin Orfila

Universidad Carlos III de Madrid, Leganes, Spain

[a]*Avenida de la Universidad 30, Leganes, Spain*

Abstract

Data storage in the cloud is becoming widespread. Deduplication is a key mechanism to decrease the operating costs cloud providers face, due to the reduction of replicated data storage. Nonetheless, deduplication must deal with several security threats such as honest-but-curious servers or malicious users who may try to take ownership of files they are not entitled to. Unfortunately, state-of-the-art solutions present weaknesses such as not coping with honest-but-curious servers, deployment problems, or lacking a sound security analysis. In this paper we present a novel Proof of Ownership scheme that uses convergent encryption and requires neither trusted third parties nor complex key management. The experimental evaluation highlights the efficiency and feasibility of our proposal that is proven to be secure under the random oracle model in the bounded leakage setting.

Keywords: Deduplication, Proof of Ownership, Convergent encryption, Cloud computing

An efficient confidentiality-preserving Proof of Ownership for Deduplication

Lorena González-Manzano, Agustin Orfila

Universidad Carlos III de Madrid, Leganes, Spain

[b]*Avenida de la Universidad 30, Leganes, Spain*

1. Introduction

Cloud computing has emerged as a computational paradigm to provide services for cloud users. A total of three layers can be distinguished in cloud computing, namely, Platform, Infrastructure and Software as a Service (PaaS, IaaS and SaaS respectively) [1]. SaaS is the layer that users are more likely to interact with because it corresponds to applications running on the cloud infrastructure that are accessible from client devices, e.g. web browsers. Many of these applications allow outsourcing data to the cloud, e.g. Dropbox[1] or Box[2]. Service providers try to avoid the upload and storage of replicated data, in an effort to maximize bandwidth and minimize storage space. Deduplication is a key technique to keep from storing multiple copies of the same data. Accordingly, target-based, source-based and cross-user deduplication are distinguished [2]. Target-based enforces deduplication at server side. Clients are not involved in the process and only storage space is saved. By contrast, source-based deduplication is enforced at client side. Hence, each client deduplicates his data what saves storage space and bandwidth. More challenging, cross-user deduplication helps saving both storage space and bandwidth to a greater extent. A file is only uploaded to the server if it has not been uploaded by other user.

Deduplication involves security threats which pose new challenges. In par-

[1] www.dropbox.com
[2] www.box.com

ticular, service providers have to ensure that stored data is only available for the right set of clients. Files are typically indexed in the server by its digest. Originally, the upload of a file digest by a client was interpreted as a proof that the client actually owns the file. However this procedure spots security threats [2]. For instance, a cloud storage service could be used as a content-distribution network (CDN). Hence, a CDN attack can be performed by exchanging among different users the file digests of large files, e.g. movies. This attack was carried out in the wild by Dropship [3]. In addition, an attacker can compromise the server and can get access to the internal cache, which includes file hashes. Then, in the possession of hashes the attacker is able to download the corresponding files.

In order to mitigate the aforementioned threats the Proof of Ownership (PoW) concept was introduced by Halevi et al. [4]. A PoW scheme is a security protocol used by a server to verify that a client owns a particular file. If the probability of fraudulently proving the ownership of a file is negligible given a security parameter, the PoW is assumed to be secure even if the adversary knows part of the uploaded file [5]. Besides, PoW schemes look for being computationally efficient at client and at server side in terms of I/O and bandwidth. Additionally, PoW schemes should not require the server to load large portions of the file from its back-end storage at each execution of PoW. To enhance client-side efficiency, Di Pietro et al. [5] proposed s-PoW, an alternative challenge-response PoW scheme to the one introduced by Halevi et al [4]. s-PoW proofs refer to client responses to server challenges, which consist of particular random bits of the requested file. Also in the context of PoW, Blasco et al. [6] proposed a PoW scheme based on bloom filters (bf-PoW) that provides flexibility and scalability. bf-PoW has been proven to be more efficient than Halevi et al.'s approach at client side and more than Di Pietro et al.'s proposal at the server side.

PoW is considered a part of remote data auditing (RDA) procedures, a novel research topic that refers to a group of protocols that verify the correctness of the data over a cloud managed by untrusted providers [7]. RDA involves security

protocols such as Provable Data Possession (PDP) [8], Proof of Retrievability (POR) [9], and the inner Proof of Ownership (PoW) [4]. PDP allows a cloud client to verify that the server possesses the original data without retrieving it. POR is a type of cryptographic Proof of Knowledge that ensures the privacy and integrity of outsourced data without having to download the files. The main difference between PDP and POR is the security features they provide. Whereas in the POR approach the client's data is completely stored on the server, PDP solutions only guarantee that most of the client's data is kept in the server. Thus, in PDP schemes a small portion of the data may be lost by the server.

Clients expect service providers to manage their deduplicated data appropriately. However, data privacy preservation requires taking a step further towards honest-but-curious servers. It is assumed that these servers 1) do not tamper data, 2) honestly execute the proposed scheme and 3) try to learn the content of stored files [10]. In this respect, it is important to note that cloud servers may be hacked or make careless technical mistakes, which may expose client data to unauthorized users. Protection mechanisms against this kind of adversary focus on the use of convergent encryption (CE) cryptography [11]. CE symetrically encrypts a file (or data block) with a key that is computed from the contents of the file (or block). The rationale for using CE is to provide a deterministic encryption scheme that makes decryption keys available to file owners. These keys are unknown by the server which only stores the encrypted file.

Deduplication faces threats that have been studied from different perspectives. Some proposals have worked on the verification of the ownership of uploaded data [5, 6, 12], others have developed protection measures against honest-but-curious servers [13, 14, 15, 16] and some others have provided solutions to both problems [17, 18, 19, 20]. Indeed, the latter are especially noteworthy as they combine encryption and PoW schemes. Unfortunately, they either lack a formal security analysis, are difficult to deploy or do not present an evaluation of their efficiency. For instance, [20, 19, 18] involve additional entities apart from the server and the client, what diminishes the practicability of the proposal; and

[21] does not analyse client and server side efficiency.

CE is pointed out as insecure [11, 17] in its original form [22] and modifications have been proposed [17, 18, 19, 20, 13]. CE applied in a straightforward manner suffers from poisoning attacks. File contents may be susceptible to this attack if the association between the plain text file and the encrypted stored data can not be verified by the server. Thus, a malicious client could upload a forged encrypted file that does not correspond with the file identifier. Consequently, a legitimate client could deduplicate a file according to this identifier and delete the local copy. Whenever the legitimate client subsequently downloads the encrypted file, he will download whatever the malicious client wanted. This attack is also known as Target Collision attack [11]. In addition, CE is not secure in the bounded leakage setting because the encryption key (i.e. file hash) is generated from the input file in a deterministic way, and it is a short piece of information that can be leaked.

Contributions. This paper presents ce-PoW, a novel PoW scheme that is resilient to honest-but-curious-servers and poisoning attacks, making use of CE. ce-PoW is proven to be secure in the bounded leakage setting [4]. In addition, our experimental results show that ce-PoW is as efficient as the top PoW proposals, i.e. s-PoW [5] and bf-PoW [6], it can be compared against. This fact is remarkable because in contrast to s-PoW and bf-PoW, ce-PoW faces honest-but-curious servers. Finally, contrary to other proposals [15, 23], ce-PoW does not require complex key management or trusted third parties different from the client and the server.

Roadmap. The paper is structured as follows. Section 2 introduces the related work. In Section 3 the objectives and the design of ce-PoW scheme are presented. Subsequently, Section 4 shows the security analysis and the computational complexity of ce-PoW. Then, Section 5 describes the experimental evaluation that compares ce-PoW with top solutions that it can be compared against. Finally, conclusions and future work are outlined in Section 6.

2. Related Work

The primary goal of deduplication is to save data storage avoiding data replication. As a consequence of using this technique security issues emerge, particularly the necessity of guaranteeing the ownership of files. The concept of Proof of Ownership (PoW) was introduced by Halevi et al. [4] and they proposed a scheme named b-PoW. In b-PoW the server computes a Merkle tree of a precomputed buffer from each uploaded file. Then, clients prove the ownership of a file computing the corresponding sibling paths based on a server request. If clients correctly compute requested paths, it is assumed that they are in the possession of the file. The security of b-PoW is admittedly based on assumptions that are hard to verify. On the basis of this proposal, Di Pietro et al. proposed s-PoW [5] in which security relies on information theoretical assumptions. The server computes a set of challenges per file composed of random bits of the uploaded file. Clients prove ownership of a file by delivering the requested challenge to the server which refers, namely, to an array of file bits. At the client side, in terms of efficiency and bandwidth consumption, this approach is superior to the one proposed by Halevi et al. but less efficient at the server side. Blasco et al. proposed a PoW scheme based on bloom filters (bf-PoW) that is both flexible and scalable [6]. bf-PoW is more efficient at client side than Halevi et al.'s approach and more efficient at the server side than Di Pietro et al.'s scheme.

Zheng and Xu combined the concepts of PoW and PDP proposing Proof of Storage with Deduplication (POSD) to achieve efficiency and security [12]. POSD consists of four phases, namely, key generation, uploading, auditing and deduplication. The security weakness of POSD is that clients have to be reliable in the key generation process. Shin et al. identified this security problem and proposed an enhancement of POSD: keys are blended with random values provided by the storage server.

Honest-but-curious servers are a threat to manage under the deduplication scenario. CE is the most widely used technique in this regard. In the context

of file systems, the combination of data confidentiality and data deduplication was introduced by Douceur et al. [22]. Given that hashes are public data, the original CE scheme is not secure [11, 21] and thus, different approaches apply variations to the original scheme.

Following we describe the remaining worth mentioning proposals that use CE for deduplication and we compare them with our approach in Section 2.1. Xu et al. proposed a CE scheme in which files are encrypted with an AES random key τ, which is in turn encrypted using the file hash as key obtaining C_τ [21]. In the first upload of a file, the file hash, $hash(F)$, the encrypted file C_F and C_τ are provided to the server. Subsequently, the server stores $hash(F)$, C_τ and $hash(C_F)$ in a look up table. C_τ is delivered to clients to prove file ownership. A client passes the proof if he is able to construct the appropriate i-th leaf of a Merkle hash tree created through the file F. Unfortunately, performance at client and at server sides is not compared with state of the art proposals and the leakage setting is not compatible with them.

Other proposals involve some third party apart from the server and the client. Li et al. presented Dekey [18] that is focused on the enhancement of key management. Files are convergently encrypted and decryption keys are also encrypted by means of a master key which is managed by multiple servers. Decryption keys will be only delivered once a PoW scheme is passed. Stanek et al. presented a convergent threshold public key cryptosystem together with the use of a PoW scheme [20]. Accordingly, a pair of trusted entities are involved, an identity provider to prevent sybil attacks and an indexing service to prevent the leakage of information of unpopular files. Jin et al. presents a scheme to keep clients anonimity[19]. A trusted third party, called intermediator, is involved. Cloud providers store encrypted files and intermediators enforce a PoW scheme over plain text files. They guarantee client anonymity hiding the relationship between clients and their files. ClouDedup provides deduplication at file and at block level using CE [14]. It relies on a pair of trusted third parties, a metadata manager to allow key-management and block-level deduplication and an additional server to guarantee data confidentiality. Based on Message-Locked

Encryption, DupLESS is proposed in [15]. Data is encrypted using keys provided by a semi-trusted key server after authentication. Besides, again applying CE, Li et al. presented a deduplication storage system where searches are enforced over encrypted content [16]. Contrary to the use of CE, Ng et al. proposed the combination of public key cryptography together with PoW [23]. Files are encrypted at client side and decryption keys are distributed among a particular group of users. It is based on devising privacy preserving proofs of ownership.

2.1. Comparison

To identify the novelty of our proposal a comparison of related works is depicted in Table 1. Analysed features are: addressing honest-but-curious servers; development of a PoW scheme; use of third parties; tackling formal security analysis; addressing bandwidth efficiency; addressing server storage efficiency; applying block and/or file level deduplication; and practicability of the approach.

On the one hand, this analysis shows that several approaches [5, 6, 12] tackle the ownership problem of uploaded data but they do not address honest-but-curious servers. Just some of them [5, 6, 12] explore server storage efficiency and client-server bandwidth efficiency as well. In addition, these works present a formal security analysis and are practical. However, amongst these three, [12] involves a third party, thus increasing management complexity.

On the other hand, other proposals develop countermeasures against honest-but-curious servers [14, 11]. [15] is the one which provides bandwidth efficiency together with honest-but-curious servers protection. Moreover, the practicability of the approach and the use of third parties is also considered in [15], as well as in [14].

By contrast, more challenging proposals address both problems, namely, they build a PoW scheme and offer protection against honest-but-curious servers [21, 18, 20, 19, 23]. Furthermore, [21, 20, 23] provide a formal security analysis and [20] focuses on server storage efficiency as well. Nonetheless, this last proposal, along with many others [19, 14, 15, 23], needs third parties and it

Table 1: Related work comparison

Proposals	Honest-but-curious servers	PoW scheme	Third parties	Formal security analysis	Bandwidth efficiency	Server space efficiency	Block level dedup.	File level dedup.	Pract.
[5]	-	√	-	√	√	√	-	√	√
[6]	-	√	-	√	√	√	-	√	√
[12]	-	√	Auditor entity	√	√	√	√	-	√
[20]	Threshold cryptosystem	√	Identity provider, Indexing service	√	-	√	√	-	√
[21]	Modified convergent encryption scheme	√	-	√	-	-	√	-	*Partial*
[18]	Convergent encryption	√	Key management cloud server provider	-	-	-	√	√	√
[19]	Not specified - use of public keys	√	Intermediator	-	-	-	-	√	
[14]	Convergent encryption	-	Metadata manager, Additional server.	-	-	-	√	-	√
[15]	Messsage-lock encryption	√*	Key server	-	√	-	-	√	√
[23]	Public key encryption	√	Third party	√	-	-	√	-	-
[11]	Convergent encryption	-	Chunk store, Metadata store	-	-	-	-	√	-
ce-PoW	Convergent encryption	√	-	√	√	√	-	√	√

*: mentioned but not applied

makes management more complex. Indeed, [21] is the only approach which deals with honest-but-curious servers without additional entities. However, it does not consider bandwidth and server space efficiency. Besides, its practicability is partially considered because it is not compared with other proposals and just some parts of the scheme are analysed, namely hash functions and the proposed PoW scheme at client side. Likewise, file content guessing attacks are more challenging in ce-PoW than in [21] (see Section 4.3).

This paper proposes ce-PoW, a novel PoW scheme that incorporates a variation of CE. Apart from dealing with honest-but-curious servers, ce-PoW is supported by a formal security analysis and looks for bandwidth efficiency and server space efficiency. Additionally, the practicability of our approach is shown by means of an experimental work.

It is very well known that CE in its original form is insecure because encrypted files can be decrypted by means of public data (i.e. the hash of the files [22]). Moreover, Xu et al. exposed that the combination of a PoW scheme and CE is incompatible [21]. PoW schemes assume that some data can be leaked and consequently CE becomes insecure if hashes are leaked. On the bases of these statements, it should be noticed that CE is applied at chunk level instead of at file level. Then, the encryption of each file requires as many keys (hashes) as the number of chunks. Besides, chunck size depends on the desired level of security (see Section 4.1).

3. Convergent Encryption compatible Proof of Ownership Scheme

In this Section we describe our proposal. First, we provide an overview of ce-PoW phases. Second, we introduce the adversarial model. Then, the design goals of ce-PoW are described. Finally, ce-PoW is described in a detailed way.

3.1. System overview

Our scheme describes the interactions between a server $\mathcal{S}$ and clients, as depicted in Figure 1. Each client $\mathcal{C}$ has a unique identifier $id(\mathcal{C})$ and uploads

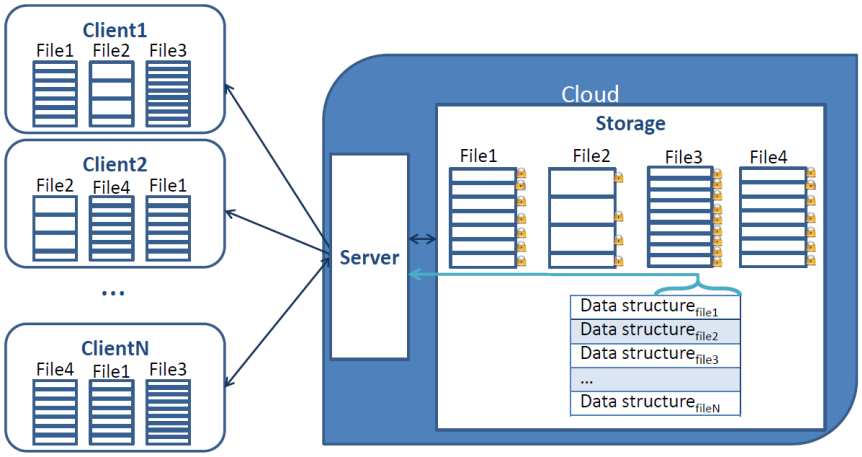

Figure 1: System overview. Each file is divided in chunks of different sizes.

an arbitrary number of files, convergently encrypted in chunks, to a storage provider using $\mathcal{S}$.

The behaviour of the system is different for the *first* and for *subsequent* uploads of any given file f. The server identifies each file according to a digest h_c that the client computes over the convergently encrypted file chunks. First, the user uploads the digest h_c that allows the server to distinguish the first from subsequent uploads. In particular, the upload is treated as initial if the server receives a particular digest for the first time; the upload is treated as subsequent if the server has already received the digest. For the first upload of file f, the server requires the client to upload the file convergently encrypted in chunks (Client Initialization Phase, Figure 2). Then, the server initializes a set of data structures to be used during subsequent uploads of the same file (Server Initialization Phase, Figure 2). When the file is going to be uploaded again by a different client, he is challenged by the server to prove the ownership of the file (Challenge Phase, Figure 2). The server verifies the responses provided by the client against the information computed in the Server Initialization Phase. A client passes the PoW if he correctly responds to the challenges sent by the server. If this is the case, the client legitimately owns the file and the ownership

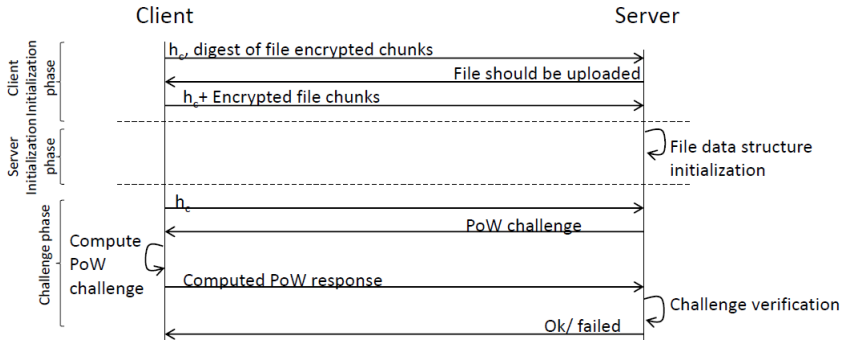

Figure 2: Overview of the ce-PoW phases

is stored in the corresponding file data structure. Accordingly, he can later request the download of the file with no further need to pass a challenge phase.

3.2. Adversarial model

The main goal of a malicious client is to pass a PoW execution in regard to a file he does not own. The legitimate owner of the file may collude with a malicious client leaking a bounded part of the file. However, in a PoW scheme it is assumed that the exchange of information does not take place interactively during the PoW challenge. In other words, a PoW does not protect against an adversary who uses the rightful owner of a file as an interactive oracle to obtain the correct responses to a PoW challenge. It is also assumed that 64MB is a size big enough to discourage collusion [4].

3.3. Objectives

The design objectives of the ce-PoW scheme can be summarized as follows:

security: the probability that a malicious client $\tilde{\mathcal{C}}$, who does not own a complete file f, succeeds in a PoW is negligible given a security parameter κ;

collusion resistance: in order to be able to engage in a successful PoW a malicious client $\tilde{\mathcal{C}}$, who does not posses file f, must exchange a minimum amount S_{min} of information with the legitimate owner of f;

bandwidth efficiency: the number of exchanged bytes between client and server along a PoW execution should be minimized;

space efficiency: while a PoW is running, the server should only be required to load in memory a small piece of information, whose size is independent of the input file size.

The first two objectives tackle the security requirements of the scheme: a malicious client succeeds in a PoW for file f with negligible probability or by exchanging ahead of time at least S_{min} bytes with the colluding owner of f. The concept of S_{min} is inspired by the work of Halevi et al. [4] and accordingly, it is set to 64MB. It acts as a deterrent for a legitimate owner $\mathcal{C}$ to collude with a malicious client $\tilde{\mathcal{C}}$. $\mathcal{C}$ is forced to issue a large enough data transferred over the network to $\tilde{\mathcal{C}}$. The last two objectives are related to performance requirements. They look for minimizing network bandwidth and memory consumption (on the server side).

3.4. Our Scheme: ce-PoW description

Let $\mathcal{H}_2 : \{0,1\}^B \rightarrow \{0,1\}^l$ be a cryptographic hash function: the two system parameters B and l represent the chunk size and the token size respectively. They play an important part in the security and performance of the scheme as described in Section 4.1. Let also $\mathcal{H}_1 : \{0,1\}^* \rightarrow \{0,1\}^n$ be a cryptographic hash function where n is a positive integer. In our scheme, the file is split in chunks and the i-th chunk of file f is identified as $f[i]$.

Our scheme has two separate phases (see Figure 3): In the **initialization phase**, the client sends the file size to the server. The server sends to the client the number of *chunks* the file should be split into. This number is computed according to the analysis performed in Section 4.1. Then, the client convergently encrypts each chunk, computes $\mathcal{H}_2$ over each encrypted chunk and, finally, computes $\mathcal{H}_1$ over the resulting hashes obtaining h_c (see Algorithm 1). Thereafter, the client sends h_c and the encrypted chunks to the server. The server then computes h_c from the received encrypted chunks and compares the result with the received h_c, in order to avoid poisoning attacks. If the comparison is successful, the server creates an associative array $\mathcal{A}$ that maps strings of finite size to 4-tuples; we use the dot notation to refer to components of tuples. The hash

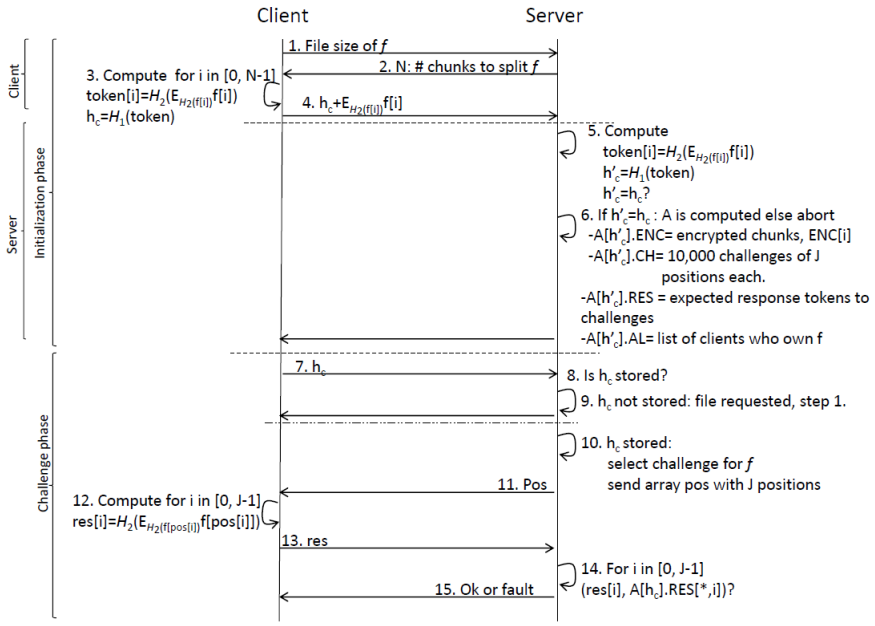

Figure 3: ce-PoW scheme

of the encrypted chunks (h_c) is the lookup key for $\mathcal{A}$: $\mathcal{A}[h_c].ENC$ contains the convergently encrypted file chunks, $\mathcal{A}[h_c].CH$ stores 10,000 challenges (with J random positions each), $\mathcal{A}[h_c].RES$ keeps the expected response tokens that correspond to the challenges and $\mathcal{A}[h_c].AL$ contains a list of identifiers of clients who own f (see Algorithm 2). The number of challenges has been selected according to s-PoW, in order to perform a fair comparison. As in s-PoW or bf-PoW, new challenges are generated when computed ones are consumed. This process can be done when the workload of the server is low.

Algorithm 1: First client upload. Client side

Input: The number of chunks N and a file f
Output: The hash h_c of the convergently encrypted file chunks; and the convergently encrypted chunks $E_{\mathcal{H}_2(f[i])}f[i]$
for $i \leftarrow 0$ ***to*** $N-1$ **do**
  $token[i] \leftarrow \mathcal{H}_2(E_{\mathcal{H}_2(f[i])}f[i])$;
end
$h_c \leftarrow \mathcal{H}_1(token)$;
return h_c and $E_{\mathcal{H}_2(f[i])}f[i]$;

Algorithm 2: First client upload. Server-side

Input: Encrypted chunks $ENC[i] = E_{\mathcal{H}_2(f[i])} f[i]$ and h_c uploaded by client $\mathcal{C}$.
Output: The entry $\mathcal{A}[h_c]$
for $i \leftarrow 0$ ***to*** $N-1$ **do**
  Compute array *token* from received $ENC[i]$
  $token[i] \leftarrow \mathcal{H}_2(ENC[i])$;
end
$h_c \leftarrow \mathcal{H}_1(token)$;
if $\neg Match(h_c, \mathcal{H}_1(token))$ **then**
  return $\perp$;
end
Store 10,000 random challenges CH with J indexes each
for $x \leftarrow 0$ ***to*** 9999 **do**
  for $y \leftarrow 0$ ***to*** $J-1$ **do**
    $pos[y] \leftarrow PRF(seed)$;
    $CH[x, y] \leftarrow pos[y]$;
    $RES[x, y] \leftarrow token[pos[y]]$;
  end
end
$\mathcal{A}[h_c].ENC \leftarrow ENC$;
$\mathcal{A}[h_c].CH \leftarrow CH$;
$\mathcal{A}[h_c].RES \leftarrow RES$;
$\mathcal{A}[h_c].AL \leftarrow \{id(\mathcal{C})\}$;
return $\mathcal{A}[h_c]$;

In the **challenge phase**, the server receives an h_c value. If h_c entry is not found, the server requests the client to upload the file size, coming back to step 1 of the ce-PoW scheme (see Figure 3). Alternatively, if an entry for h_c is found in $\mathcal{A}$, the server loads in memory the first unused challenge (an array *pos* with J random chunk indexes) together with the corresponding responses and sends the challenge to the claiming client. The client then performs his part of the *challenge phase* (Algorithm 3): in particular, the client computes the response token for each of the J chunk indexes and sends the array of response tokens to the server. Subsequently, the server can execute its part of the challenge phase (Algorithm 4). It checks whether the array of response tokens matches the array of response tokens loaded in memory. If this is the case, the server considers the PoW successful and assigns f to $\mathcal{C}$. Otherwise, the client has failed the PoW.

Algorithm 3: Challenge phase – client side.

Input: A file f and an array pos of J indexes
Output: An array res of J response tokens
for $i \leftarrow 0$ ***to*** $J-1$ **do**
  $res[i] \leftarrow \mathcal{H}_2(E_{\mathcal{H}_2(f[pos[i]])} f[pos[i]])$;
end
return res;

Algorithm 4: Challenge phase – server side.

Input: h_c of a file f; two arrays pos and res of J indexes and client response tokens, respectively
Output: The outcome of the challenge
for $i \leftarrow 0$ ***to*** $J-1$ **do**
  if $\neg Match(res[i], \mathcal{A}[h_c].RES[*, i]))$ **then**
    return $\perp$;
  end
end
$\mathcal{A}[h_c].AL \leftarrow \mathcal{A}[h_c].AL \bigcup \{id(\mathcal{C})\}$;
return $\top$;

Finally, a client $\mathcal{C}$ may request the download of a particular file f by sending h_c to the server; if the file exists in the server, the latter will check whether $id(\mathcal{C}) \in \mathcal{A}[h_c].AL$ and $\mathcal{A}[h_c].ENC$ will be sent to $\mathcal{C}$ if the check is successful.

4. Theoretical analysis

This Section presents first the security of our PoW scheme. Then we perform a complexity analysis to compare it with s-PoW and bf-PoW. The security analysis provides the rationale to define the settings of the experimental setup. Finally the execution of content guessing and poisoning attacks is discussed.

4.1. Security

The security of the proposed scheme is analysed in this section. ce-PoW relies on the information theoretical assumptions that were established by Di Pietro et al. [5] for s-PoW, and assumed by Blasco et al. [6]. Given a file f, the objective of the adversary $\tilde{\mathcal{C}}$ is to engage in a successful PoW, without actually possessing the entire file. PoWs are not designed against an adversary that uses

a legitimate file owner as a real-time oracle to provide the correct responses to the PoW challenges. However, the adversary can collude with clients prior to the PoW challenge. In order to model this interaction and taking into account that $\tilde{\mathcal{C}}$ may know parts of the file, the adversary's knowledge of the target file can be bounded to a percentage p. In other words, the probability the adversary knows a byte of the file at a randomly chosen position is p. We also assume that if the adversary does not posses a particular byte, he will be able to guess it with probability g. It is easy to prove that the best strategy for the adversary is to cluster his knowledge of the file into contiguous and aligned chunks of size B to obtain an optimal probability of success. Accordingly, we will refer to p as the probability that the adversary knows the whole content of a chunk of size B whose position is chosen at random.

The ce-PoW challenge phase requires from the adversary the knowledge of J tokens that correspond to J random chunks. Once received by the server, each token is checked against the one computed locally. Let us consider the generic i-th position out of the J random positions requested by the server. The PoW is passed if the $\tilde{\mathcal{C}}$ produces J tokens tok_i correctly. Therefore, we can compute the probability of success (let us call the event $succ$) of the adversary as:

$$P(succ) = P(tok_i)^J \tag{1}$$

Let us notice that the adversary cannot answer with the token that corresponds to another chunk (unless the two are equal). Let us now analyse the probability of the event tok_i, i.e. the probability that the adversary can successfully produce the bits of the i-th token. Let also $know_i$ be the event that the adversary knows the i-th chunk—recall that the probability of this event is p. At this point, the adversary either knows the chunk, and can therefore compute the token (convergently encrypting it using $\mathcal{H}_2$), or does not; in the latter case, the adversary can either guess the y unknown bytes that compose the chunk (the probability of a correct guess being g^y under our simplifying assumptions) or guess the l-bit output of $\mathcal{H}_2$ that is used to generate the token (the probability of a correct guess being 0.5^l, where 0.5 stems from the random oracle model and

the assumption that each bit outputted by $\mathcal{H}_2$ is truly random). Given that the token is always shorter than the chunk, we postulate that – in the absence of other information – it is easier for the adversary to guess the token. That is, we assume that $g^y << 0.5^l$; then we can write:

$$\begin{aligned} P(tok_i) &= P(tok_i \cap (know_i \cup \overline{know_i})) \\ &= P(tok_i|know_i)P(know_i) + P(tok_i|\overline{know_i})P(\overline{know_i}) \\ &= p + 0.5^l(1-p) \end{aligned} \tag{2}$$

The adversary is challenged on J independent chunk positions. Consequently, the probability of success of the adversary is:

$$\begin{aligned} P(succ) &= P(tok_i)^J \\ &= (p + 0.5^l(1-p))^J \end{aligned} \tag{3}$$

From Equation 3 we can derive a lowerbound for J that ensures $P(succ) \leq 2^\kappa$, where κ is the security parameter, as

$$J \geq \frac{\kappa \ln 2}{(1-p)(1-(0.5^l))} \tag{4}$$

The first security requirement highlighted in Section 3.3 is satisfied by ce-PoW according to Equation 4. In order to satisfy the second security requirement, collusion resistance, we need to ensure that a legitimate client $\mathcal{C}$ needs to exchange at least S_{min} bytes with a malicious client $\tilde{\mathcal{C}}$ to allow him to run a successful PoW for an unknown file. Given that tokens are typically shorter than the entire file chunks, the best strategy for the adversary is to request all tokens from the colluding client[3]. Taking into account that there are $\frac{F}{B}$ tokens in a file f of size F, the token length l can be set as:

$$l \geq S_{min}\frac{B}{F} \tag{5}$$

[3]We do not consider the case of files with an extremely low entropy, whose chunks may be compressed down to a size smaller than the corresponding chunk.

Note that Equation 5 holds only in the case of files whose size is bigger than S_{min}. For smaller files the adversary would circumvent the restriction by exchanging the file's content instead. Consequently, for smaller files, S_{min} must be scaled down to the file size itself. To discourage collusion, Halevi et al. proposed a value of 64MB for S_{min}. This value was also considered in s-PoW and bf-PoW and, accordingly, we take it into account as well.

According to the previous analysis, given that clients have to store the decryption keys in order to decrypt stored files, the required storage space for decryption keys is of 64MB for files bigger than 64MB. For instance, encrypting a movie of 1.280 GB requires the client to store as many decryption keys as 5% of the original file size. By contrast, for files smaller than 64 MB, the client needs to store as much information as the file size. Nevertheless, it is very likely that there is a lot of redundancy in client files. For instance, different versions of a document will share multiple chunks and the corresponding decryptions keys can be stored only once. Accordingly, source-based block level deduplication at client side would greatly reduce the storage needs or alternatively, decryption keys could be stored in a different cloud storage [14, 18].

4.2. Complexity

In this section, the bandwidth and space complexity of our scheme is analysed, along with its computational and I/O requirements. We compare our solution against Di Pietro et al. [5] and Blasco et al. [6] proposals. Already used, s-PoW and bf-PoW refer to these two schemes respectively. In all cases, we only analyse the upper bounds and we consider all hash functions to have the same computational cost. Results are summarized in Table 2. s-PoW and bf-PoW require hashing the entire file f for identification purposes while ce-PoW requires hashing the encrypted chunks. Although Di Pietro et al. reduce this complexity by using a function with similar properties but a smaller computational footprint [5], we choose to factor out this optimization as it can be applied to all three schemes alike.

At the client side, the computational cost is higher in ce-PoW due to en-

cryption and computation of the two hashes. Nevertheless, the I/O in the client is similar in the three proposals and it just depends on the file size. The server initial computation is also higher for ce-PoW due to the fact that two hashes have to be computed.

In the case of bf-PoW, given that each time a client $\mathcal{C}$ issues a request to $\mathcal{S}$ he must calculate $\mathcal{H}_1$ over the entire file f, we conclude that the overall computational cost is $O(F)$. In the case of ce-PoW the computational cost depends on the time to calculate two hashes and these hashes are in turn dependent on the chunk size and the cost of computing the CE of the chunks. At the server side, for bf-PoW the computational cost of the initialization phase is dominated by the cost of hashing elements to be inserted in the bloom filter. Overall, this requires k hash operations over the file. In the case of ce-PoW, the incurred cost depends on computing two hashes. For the server regular computation ce-PoW and s-PoW have a comparable complexity.

Regarding memory consumption, initialization involves loading the entire file into memory. During regular execution, the server in ce-PoW just checks if the responses received from the client match the precomputed tokens loaded in memory. The server is only required to read from disk the list of J precomputed challenges and responses, similarly to what is done in s-PoW.

In terms of bandwidth, ce-PoW requires J tokens to be sent to the server. This number increases roughly linearly as the security parameter κ increases. This is somehow similar to what happens with bf-PoW, but in bf-PoW J decreases proportionally when the BF's false positive rate increases p_f. Finally, s-PoW requires only K bits of the file chosen at random positions to be sent to the server, where K is a superlinear function of the security parameter κ.

4.3. Content guessing and poisoning attacks

CE does not provide semantic security as it is vulnerable to content-guessing attacks. Bellare et al. [13] formalized CE under the name message-locked encryption. Their security analysis highlights that message-locked encryption offers confidentiality for unpredictable messages only. This fact may be a threat

	s-PoW	**bf-PoW**	**ce-PoW**
Client computation	$O(F) \cdot hash$	$O(F) \cdot hash$	$O(B) \cdot CE \cdot hash \cdot hash$
Client I/O	$O(F)$	$O(F)$	$O(F)$
Server init computation	$O(F) \cdot hash$	$O(F) \cdot hash$	$O(B) \cdot hash \cdot hash$
Server regular computation	$O(n \cdot \kappa) \cdot PRF$	$O\left(\frac{l \cdot \kappa \cdot (log1/p_f)}{p_f}\right) \cdot hash$	$O(n \cdot l \cdot \kappa) \cdot PRNG$
Server init I/O	$O(F)$	$O(F)$	$O(F)$
Server regular I/O	$O(n \cdot \kappa)$	$O(0)$	$O(0)$
Server memory usage	$O(n \cdot \kappa)$	$O\left(\frac{log(1/p_f)}{l}\right)$	$O(n \cdot l \cdot \kappa)$
Bandwidth	$O(\kappa)$	$O\left(\frac{l \cdot \kappa}{p_f}\right)$	$O(l \cdot \kappa)$

Table 2: Complexity analysis of the ce-PoW against the other proposals. F is the file size, κ is the security parameter, n is the number of precomputed challenges in s-PoW. l is the token size and p_f is the false positive rate of the BF.

to user privacy. CE might be susceptible to offline brute-force dictionary attacks if the file has low min-entropy. Knowing that the file f underlying a target ciphertext C is drawn from a dictionary $S = \{f_1, \ldots, f_n\}$ of size n, the attacker can recover f in the time for $|S|$ off-line encryptions. The elements of S are convergently encrypted and the result compared to C. Whenever a match is found, the attacker knows f. Consequently, security against an honest-but-curious server depends on the length of S. If the number of elements in S is long enough, content guessing attacks are not feasible. Otherwise, it is a serious threat.

Proposals that deal with content guessing attacks over CE schemes are all based on the use of trusted third parties (TTPs) [14, 15]. In the case of ce-PoW, the worst scenario is faced whenever the size of the chunks is 16 bytes. If the content of the chunks is random (i.e the highest entropy), the dictionary S would have 2^{128} elements. Given that AES is the encryption algorithm, if

we consider an encryption rate of 61 MiB/sec[4], a content guessing attack over an encrypted chunk would take 10^{24} years on the average case. Likewise, if S is composed of just 2^{80} elements this time would be 10^{17} years. Nonetheless, for chunks with very low min-entropy ce-PoW would become vulnerable. For such a case, our scheme could be directly adapted to use a TTP [14] where every file is reencrypted by the TTP with the same key prior to be uploaded to the server. However, our proposal in its current form presents a trade-off between usability and security concerning content guessing attacks. Similar to ce-PoW, Xu et al. [21] presents a PoW scheme under the bounded leakage setting that faces honest-but-curious servers avoiding TTPs. Their work does not address content guessing attacks. Indeed, their proposal is vulnerable in a straightforward manner. In [21] file hashes are stored in the server and then, an honest-but-curious server can directly compute the hashes over the dictionary of guessed files $S = \{f_1, \ldots, f_n\}$ to verify matches. By contrast, in ce-PoW the server stores h_c, a digest over encrypted chunks, what hinders the attack to a certain extent. Assuming $S = \{f_1, \ldots, f_n\}$, an attacker would have to convergently encrypt the chunks of each element in S, then compute $\mathcal{H}_2$ over each encrypted chunk and finally compute h_c over the result to verify each possible match.

To successfully perform a poisoning attack on ce-PoW, a collision over hash h_c has to be found due to the check done by the server (see step 5 in Figure 3). An adversary has to find two different plain text files whose hash over the digests of convergently encrypted chunks is the same. Thus, assuming hash $\mathcal{H}_1$ is collision resistant, ce-PoW becomes resilient to poisoning attacks.

5. Experimental Results

This Section presents an experimental comparison of the proposed scheme, ce-PoW, with the schemes proposed by Di Pietro et al. [5] and by J. Blasco et al. [6], referred to as s-PoW and bf-PoW respectively. We exclude the Halevi et al.

[4] http://www.cryptopp.com/benchmarks.html

approach [4] from our experiments because its security is admittedly based on assumptions that are hard to verify. All these schemes are implemented in C++ using OpenSSL for cryptographic operations. Particularly, used cryptographic primitives are AES in counter mode and SHA-1. Besides, $\mathcal{H}_1$ corresponds to SHA-1 and $\mathcal{H}_2$ corresponds to the application of SHA-1 over each encrypted chunk, applying RC4 to extend the length of the hash to l.

The experiments have been performed on a AMD Athlon(tm) II x2 220 processor with 4GB of RAM. Input files have been randomly generated and their sizes range from 4MB to 2GB doubling the size at each step.

On the bases of [5] [6], the following parameters are applied: the security parameter is set to k=66 and the min entropy to S_{min}=64MB ; the size of the tokens (l) is set to {16, 64, 256,1024} bytes; and the probability (p) that an adversary knows a chunk of a file is established to {0.5; 0.75; 0.9; 0.95}. In this regard, the size of chunks (B) is calculated according to Equation 5 given the values of l, S_{min} and the input file size. The correspondence is depicted in Table 3. Similarly, the number of requested challenges, J, is computed according to Equation 4 leading to these values: {91, 182, 457, 914}.

Table 3: Chunk sizes (B) in bytes, computed from the file size, the token size and S_{min}

l (bytes)	16	64	256	1024	File size (Mbytes)
	16	64	256	1024	4
	16	64	256	1024	8
	16	64	256	1024	16
	16	64	256	1024	32
	16	64	256	1024	64
	32	128	512	2048	128
	64	256	1024	4096	256
	128	512	2048	8192	512
	256	1024	4096	16384	1024
	512	2048	8192	32768	2048

5.1. Server side

The time the server initialization takes has been measured and compared. In s-PoW the server precomputes n challenges composed of J random file bits

by means of a pseudorandom number generator F (Algorithm 1 of [5]). In bf-PoW the server initializes a bloom filter, divides the input files in chunks of a fixed size, creates a token per chunk and inserts a function of each token in the bloom filter (Algorithm 1 of [6]). In the case of ce-PoW, the server computes a hash per each encrypted chunk and generates n challenges (see Algorithm 2). A comparison between s-PoW, bf-PoW and ce-PoW of the server initialization phase is depicted in Figure 4 taking n=10,000. The size of file chunks in ce-PoW has been computed according to Table 3. It can be noticed that bf-PoW is the fastest approach. Nonetheless, ce-PoW with l=256B is comparable with bf-PoW with l=16B and faster than s-PoW with l=256B for all file sizes. In general, ce-PoW behaves quite well for small files (smaller than 32MB) and for big files the achieved results are comparable with s-PoW and bf-PoW, if it is properly tuned. One of the heaviest task for ce-PoW in the server is the computation of hashes to provide file integrity (avoiding poisoning attacks). Thus, removing the time to create chunk hashes, ce-PoW is the fastest approach in all cases except for ce-PoW l=256B and files between 4-32MB, as Figure 5 depicts. The computational cost of ce-PoW is particularly affected by computations to protect against poisoning attacks and honest-but-curious servers. However, s-PoW and bf-PoW do not deal with this kind of adversary.

Additionally, the server verifies the response of challenges. In bf-PoW the server calls a pseudo random function to enforce the verification process for each token involved in the requested challenge. This task is performed before the bloom filter is checked (Algorithm 3 of [6]). Conversely, in s-PoW the server has to verify that the received response matches the precomputed one. Similarly, in ce-PoW the server verifies that received tokens match precomputed ones. As a result, the time to enforce this computation is negligible for both ce-PoW and s-PoW, but not for bf-PoW.

5.2. Client side

The time spent by clients to respond to challenges is relevant. s-PoW clients look for the requested bits until they complete the challenge (Algorithm 2 of

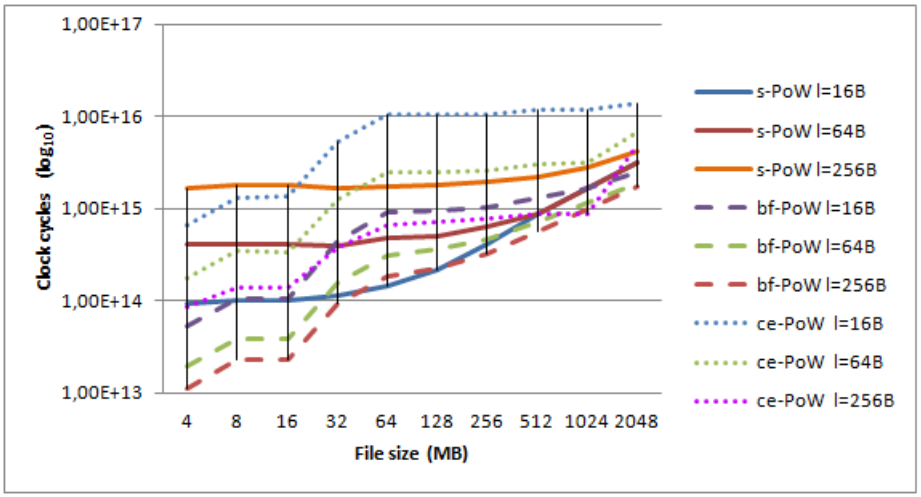

Figure 4: Clock cycles of the server initialization phase. The number of precomputed challenges is set to n=10,000.

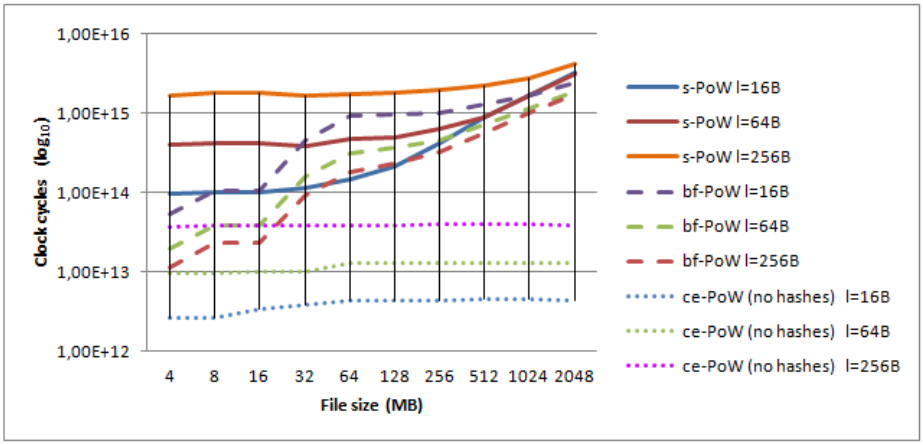

Figure 5: Clock cycles of the server initialization phase. ce-PoW hash calculations to provide file integrity are not considered. The number of precomputed challenges is set to n=10,000

[5]). Similarly, bf-PoW clients compute a token for each J chunk index using a hash function (Algorithm 2 of [6]). By contrast, ce-PoW clients first apply CE to the chunks defined by J indexes, and then compute a hash over each encrypted chunk. We have compared the client response times for a different number of requested challenges, J={91, 182, 457, 914}. Results are depicted in Figures 6-9. Regardless of J, ce-PoW achieves better results than s-PoW for

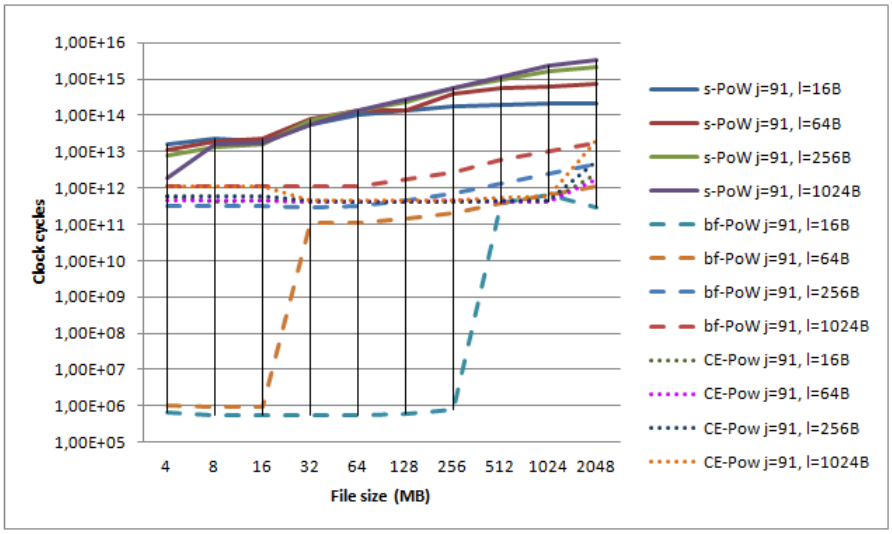

Figure 6: Client response creation clock cycles for J=91 challenges.

all file sizes and it is comparable with bf-PoW for most settings. The strong point of ce-PoW is that the time remains constant for every file size, except those greater than 2048MB. In fact, ce-PoW is much better than s-PoW and comparable with bf-PoW.

It is noteworthy that the time to compute challenges in ce-PoW in independent of chunk sizes and it remains constant. Recall from Table 3 that chunk size depends on file size. Thus, the same amount of encryptions and hashes are performed for every file. For instance, the time to encrypt file chunks of 32B is comparable with the time to encrypt chunks of size 512MB. Nonetheless, the time increases for files bigger than 2048MB because encryption is affected by big chunks, particularly for l=1024B.

In sum, bf-PoW is the fastest scheme for client response creation except for l=1024B and s-PoW is the slowest one, while ce-PoW is in the middle of both. It is particularly noticeable that in our scheme the time remains constant for files between 4-1024MB due to the different size of chunks. Besides, results achieved by ce-PoW are comparable with those of bf-PoW for big files.

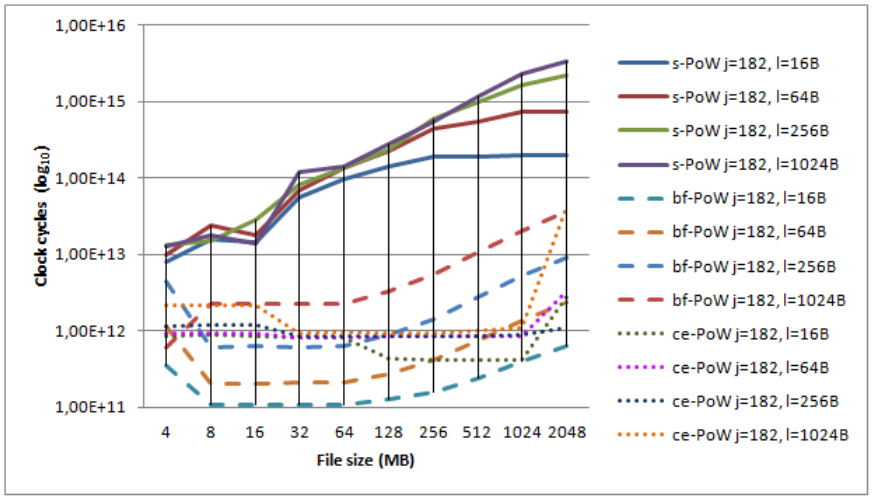

Figure 7: Client response creation clock cycles for J=182 challenges.

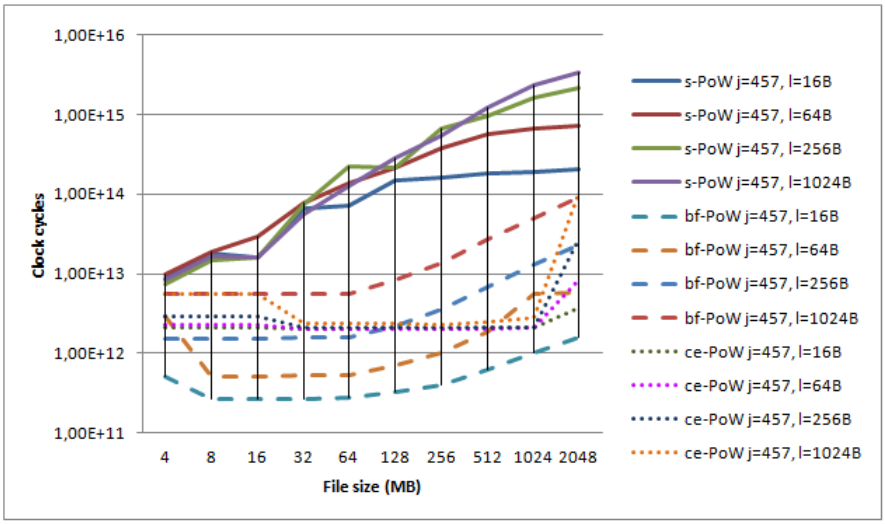

Figure 8: Client response creation clock cycles for J=457 challenges.

5.3. Summary

ce-PoW is specially efficient for files between 4-32MB. In this regard, as the average size of files uploaded to cloud providers like Amazon or Azure is 350kb[5], our scheme is as efficient as top solutions in average cases. Moreover, ce-PoW

[5] `http://www.nasuni.com/blog/57-whats_the_cost_of_a_gb_in_the_cloud`

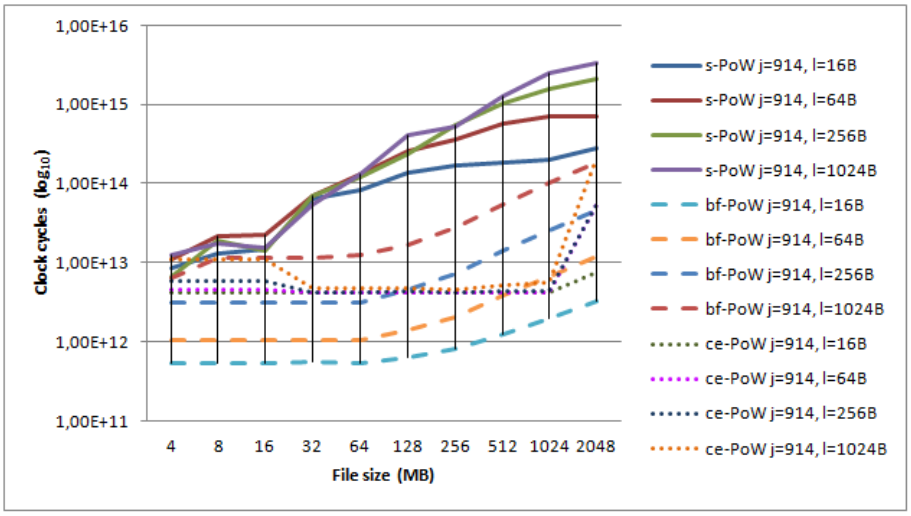

Figure 9: Client response creation clock cycles for J=914 challenges.

is analysed to be faster than s-PoW at client side and it can be compared with both s-PoW and bf-PoW at server side.

It is noteworthy that the performance comparison among ce-PoW and the other two proposals is not straightforward. ce-PoW offers protection against honest-but-curious-servers by the use of a CE scheme. Therefore, the successfulness of ce-PoW is to achieve results comparable with top proposals that do not deal with threat.

6. Conclusions

Cross-user deduplication is commonly applied by cloud providers to save storage space and bandwidth. However, if it is performed in a straightforward manner security flaws arise. This paper presents ce-PoW, a proposal that combines a Proof of Ownership scheme with Convergent Encryption to protect cloud stored data against honest-but-curious servers and outside adversaries. ce-PoW is proven to be secure under the bounded leakage setting. Weaknesses of state-of-the solutions such as the dependency on trusted third parties, have been overcome. ce-PoW is as efficient as top PoW schemes that do not deal with honest but curious servers.

Future work involves the reduction of client storage space in regard to decryption keys. Likewise, content guessing attacks against very low min-entropy files remains as an open research problem. Finally, exploring other cryptographic approaches such as Attribute Based Encryption [24] and Proofs of Work [25] could lead to outstanding new solutions.

7. Acknowledgements

We would like to thank Alessandro Sorniotti and Roberto Di Pietro for providing us the source code of their proposal and encourage us to write this paper.

References

[1] D. Zissis, D. Lekkas, Addressing cloud computing security issues, Future Generation Computer Systems 28 (3) (2012) 583–592.

[2] D. Harnik, B. Pinkas, A. Shulman-Peleg, Side channels in cloud services: Deduplication in cloud storage, Security & Privacy, IEEE 8 (6) (2010) 40–47.

[3] W. V. der Laan, Dropship, `https://github.com/driverdan/dropship` (2013).

[4] S. Halevi, D. Harnik, B. Pinkas, A. Shulman-Peleg, Proofs of ownership in remote storage systems, in: Proceedings of the 18th ACM conference on Computer and communications security, ACM, 2011, pp. 491–500.

[5] R. Di Pietro, 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, ACM, 2012, pp. 81–82.

[6] J. Blasco, A. Orfila, R. D. Pietro, A. Sorniotti, A tunable proof of ownership scheme for deduplication using bloom filters, in: Proceedings of the IEEE Conference on communications and network security, CNS, 2014.

[7] M. Sookhak, H. Talebian, E. Ahmed, A. Gani, M. K. Khan, A review on remote data auditing in single cloud server: Taxonomy and open issues, Journal of Network and Computer Applications 43 (2014) 121–141.

[8] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, Provable data possession at untrusted stores, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, 2007, pp. 598–609.

[9] A. Juels, B. S. Kaliski Jr, PORs: Proofs of retrievability for large files, in: Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp. 584–597.

[10] Q. Chai, G. Gong, Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers, in: Proceedings of the IEEE International Conference on Communications (ICC), IEEE, 2012, pp. 917–922.

[11] M. W. Storer, K. Greenan, D. D. Long, E. L. Miller, Secure data deduplication, in: Proceedings of the 4th ACM international workshop on Storage security and survivability, ACM, 2008, pp. 1–10.

[12] Q. Zheng, S. Xu, Secure and efficient proof of storage with deduplication, in: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY '12, 2012, pp. 1–12.

[13] M. Bellare, S. Keelveedhi, T. Ristenpart, Message-locked encryption and secure deduplication, in: Advances in Cryptology–EUROCRYPT 2013, Springer, 2013, pp. 296–312.

[14] P. Puzio, R. Molva, M. Önen, S. Loureiro, Block-level de-duplication with encrypted data, Open Journal of Cloud Computing (OJCC) 1 (1) (2014) 10–18.

[15] M. Bellare, S. Keelveedhi, T. Ristenpart, Dupless: server-aided encryption for deduplicated storage, in: Proceedings of the 22nd USENIX conference on Security, USENIX Association, 2013, pp. 179–194.

[16] J. Li, X. Chen, F. Xhafa, L. Barolli, Secure deduplication storage systems with keyword search, in: Proceedings of the IEEE 28th International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2014, pp. 971–977.

[17] J. Xu, E.-C. Chang, J. Zhou, Weak leakage-resilient client-side deduplication of encrypted data in cloud storage, in: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM, 2013, pp. 195–206.

[18] J. Li, X. Chen, M. Li, J. Li, P. Lee, W. Lou, Secure deduplication with efficient and reliable convergent key management, Parallel and Distributed Systems, IEEE Transactions on 25 (6) (2014) 1615–1625.

[19] X. Jin, L. Wei, M. Yu, N. Yu, J. Sun, Anonymous deduplication of encrypted data with proof of ownership in cloud storage, in: Proceedings of the 13th IEEE/CIC International Conference on Communications in China (ICCC), 2013, pp. 224–229.

[20] J. Stanek, A. Sorniotti, E. Androulaki, L. Kencl, A secure data deduplication scheme for cloud storage, Tech. rep., IBM (2013).

[21] J. Xu, J. Zhou, Leakage resilient proofs of ownership in cloud storage, revisited, in: I. Boureanu, P. Owesarski, S. Vaudenay (Eds.), Applied Cryptography and Network Security, Vol. 8479 of Lecture Notes in Computer Science, 2014, pp. 97–115.

[22] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, M. Theimer, Reclaiming space from duplicate files in a serverless distributed file system, in: Proceedings of the 22nd International Conference on Distributed Computing Systems, IEEE, 2002, pp. 617–624.

[23] W. K. Ng, Y. Wen, H. Zhu, Private data deduplication protocols in cloud storage, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, ACM, 2012, pp. 441–446.

[24] J. Bethencourt, A. Sahai, B. Waters, Ciphertext-policy attribute-based encryption, in: Security and Privacy, 2007. SP'07. IEEE Symposium on, IEEE, 2007, pp. 321–334.

[25] D. Liu, L. J. Camp, Proof of work can work., in: WEIS, 2006.