RZ 3854 (# Z1310-007) 10/15/2013
Computer Science 26 pages

Research Report

Proof of Ownership for Deduplication Systems: a Secure, Scalable, and Efficient Solution

Roberto DI Pietro‡, Alessandro Sorniotti*

‡Università di Roma Tre,
Italy

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

Proof of ownership for deduplication systems: a secure, scalable, and efficient solution

Roberto DI Pietro
Università di Roma Tre, Italy
dipietro@mat.uniroma3.it

Alessandro Sorniotti
IBM Research – Zurich, Switzerland
aso@zurich.ibm.com

October 8, 2013

Abstract

Deduplication is a technique used to reduce the amount of storage needed by service providers. It is based on the intuition that several users may want (for different reasons) to store the same content. Hence, storing a single copy of these files would be sufficient. Albeit simple in theory, the implementation of this concept introduces many security risks. In this paper, we address the most severe one: an adversary, possessing only a fraction of the original file, or colluding with a rightful owner who leaks arbitrary portions of it, becomes able to claim possession of the entire file. The paper's contributions are manifold: first, we review the security issues introduced by deduplication, and model the security threats our scheme addresses; second, we introduce a novel Proof of Ownership (POW) scheme that has all the features of the state-of-the-art solution, but incurs only a fraction of the overhead experienced by the competitor. We also show that the security of the proposed mechanisms relies on information-theoretical rather than computational assumptions, and propose viable optimization techniques that further improve the scheme's performance. Finally, the quality of our proposal is supported by extensive benchmarking.

1 Introduction

The rapid surge in cloud service offerings has resulted in a sharp drop in prices of storage services, and in an increase in the number of customers. Through popular providers, like Amazon s3 and Microsoft Azure, and backup services, like Dropbox and Memopal, storage has indeed become a commodity. Among the reasons for the low prices, we find a strong use of multitenancy, the reliance on distributed algorithms run on top of simple hardware, and an efficient use of the storage backend thanks to compression and deduplication. Deduplication is widely adopted in practice for instance by services such as Bitcasa [7] and Ciphertite [8].

Deduplication is the process to avoid having to store the same data multiple times. It leverages the fact that large data sets often exhibit high redundancy. Examples include common email attachments, financial records, with common headers and semi-identical fields, and popular media content—such as music, videos—likely to be owned (and stored) by many users.

There are four different deduplication strategies, depending on whether deduplication happens at the client side (i.e. before the upload) or at the server side, and whether it happens at a block level or at a file level. Deduplication is most efficient when it is triggered at the client side, as it also saves upload bandwidth. For these reasons, deduplication is a critical enabler for a number of popular and successful storage services (e.g. Dropbox, Memopal) that offer cheap remote storage to the broad public by performing client-side deduplication, thus saving both the network bandwidth and the storage costs associated with processing the same content multiple times. However, new technologies introduce new vulnerabilities, and deduplication is no exception.

Security Threats to Deduplication Harnik *et al.* [15] have identified a number of threats that can affect a storage system performing client-side deduplication. These threats, briefly reviewed in the following, can be turned into practical attacks by any user of the system.

A first set of attacks targets the privacy and confidentiality of users of the storage system. For instance, a user can check whether another user has already uploaded a file by trying to upload it as well, and by checking—e.g. by monitoring local network traffic—whether the upload actually takes place. This attack is particularly relevant for rare files that may reveal the identity of the user who performed the original upload. This attack, as shown in [15], can also be turned into an attack targeting the discovery of the content of a file. Suppose a document has standard headings, text and signature, contains mostly public information, and has only a small subset of private information. A malicious user can forge all possible combinations of such a document, upload them all and check for the ones that undergo deduplication.

A different type of attack can turn deduplication features into a covert channel. Two users who have no direct connectivity could try to use the storage system as a covert channel. For instance, to exchange a bit of information, the two users would pre-agree on two files. Then the transmitting user uploads one of the two files; the receiving user detects which one gets deduplicated and outputs either 0 (for the first file) or 1 (for the second one).

Finally, users can abuse a storage system by turning it into a content-distribution network: users who wish to exchange large files leveraging the large bandwidth available to the servers of the storage systems could upload only a single copy of such a file and share the short token that triggers deduplication (in most cases, the hash digest of the file) among all users who wish to download the content. Real-world examples of such an approach include Dropship [25].

Proof of Ownership (POW) To remedy the security threats mentioned above, the concept of Proof of Ownership (POW) has been introduced [13]. POW schemes essentially address the root-cause of the aforementioned attacks to deduplication, namely, that the proof that the client owns a given file (or block of data) is solely based on a static, short value (in most cases the hash digest of the file), whose knowledge automatically grants access to the file.

POW schemes are security protocols designed to allow a server to verify (with a given degree of assurance) whether a client owns a file. The probability that a malicious client engages in a successful POW run must be negligible in the security parameter, even if the malicious client knows a (relevant) portion of the target file. A POW scheme should be efficient in terms of CPU, bandwidth and I/O for the both the server and all legitimate clients: in particular, POW schemes should not require the server to load the file (or large portions of it) from its back-end storage at each execution of POW.

Additional assumptions about POW schemes are that they should take into account the fact that a user wishing to engage in a successful POW run with the sever may be colluding with other users who possess the file and are willing to help in circumventing POW checks. These latter users however, are neither assumed to always be online (i.e. they cannot answer the POW challenges on behalf of the malicious user), nor are they willing to exchange very large amounts of data with the malicious user. Both assumptions are arguably reasonable, as such users would have no strong incentive in helping the free-riders.

Halevi *et al.* [13] have introduced the first practical cryptographic protocol that implements

POW. Their seminal work, however, suffers from a number of shortcomings that might hinder its adoption. The first is that the scheme has extremely high I/O requirements at the client-side: it either requires clients to load the entire file into memory or to perform random block accesses with an aggregate total I/O higher than the size of the file itself. Secondly, the scheme takes a heavy computational toll on the client. Thirdly, its security is admittedly based on assumptions that are hard to verify. Finally, its good performance on the server side depend strongly on setting a hard limit (64 MiB) on the amount of bytes an adversary is entitled to receive from a legitimate file owner.

Contributions In this paper, inspired by [9], we propose a novel family of schemes for secure Proof of Ownership. The different constructions attain several ambitious goals: i) their I/O and computational costs do *not* depend on the input file size; ii) they are *very efficient* for a wide range of systems parameters; iii) they are *information-theoretically secure*; and, iv) they require the server to keep a per-file state that is a *negligible fraction* of the input file size. Finally, they explore different optimization strategies that do not compromise security.

Roadmap The remainder of this paper is organised as follows: Section 2 reviews the state of the art. Section 3 defines system and security models. Section 4 presents the basic scheme and three optimizations. Section 5 describes the implementation and benchmarks. Section 6 contains a discussion on the performance and potential for further optimizations, while Section 7 presents our conclusions.

2 Related Work

Several deduplication schemes have been proposed by the research community [20, 18, 2] showing how deduplication allows very appealing reductions in the use of storage resources [11, 14].

Douceur *et al.* [10] study the problem of deduplication in a multitenant system where deduplication has to be reconciled with confidentiality. The authors propose the use of convergent encryption. Convergent encryption of a message consists of encrypting the plaintext using a deterministic (symmetric) encryption scheme with a key that is deterministically derived solely from the plaintext. Clearly, when two users independently attempt to encrypt the same file, they will generate the same ciphertext which can easily be deduplicated. Unfortunately, convergent encryption does not provide semantic security as it is vulnerable to content-guessing attacks. Bellare *et al.* [6] formalize convergent encryption under the name *message-locked encryption*. As expected, the security analysis presented in [6] highlights that message-locked encryption offers confidentiality for unpredictable messages only, clearly failing to achieve semantic security. Storer *et al.* [24] point out some security problems related to the way convergent encryption is used in [10] and propose a security model and two protocols for secure data deduplication.

The seminal work of Harnik *et al.* [15] first discusses the shortcomings of client-side deduplication, and presents some basic solutions to the problem. In particular, attacks on privacy and confidentiality can be addressed without a full-fledged POW scheme by triggering deduplication only after a small, but random, number of uploads.

Halevi *et al.* [13] first introduce the concept of Proof of Ownership as a solution to the inherent weaknesses associated with client-side deduplication. A detailed description of their schemes is the subject of the next section. Their seminal work has been extended in a number of other works [22, 27, 26]. However, these extensions either do not challenge the key design choices of the original scheme or focus on other problems related to data outsourcing, such as integrity, auditing or proof of retrievability, and are thus unable to address the shortcomings of the original scheme.

Whereas POW deals with the assurance that a client indeed possesses a given file, Provable Data Possession (PDP) and Proof of Retrievability (PoR) deal with the dual problem of ensuring—at the client-side—that a server still stores the files it ought to. PDP is formally introduced by Ateniese and colleagues [4, 3]. A number of earlier works already address remote integrity checking, see the Related Work section of [3] for more details. The protocol of Ateniese *et al.* is based on asymmetric cryptography: the data owner computes – prior to the upload – a tag for each data block. Any verifier can later prompt the storage service to answer a challenge on a subset of blocks: the computation of the response involves both data blocks and tags. Ateniese *et al.* [5] present a dynamic PDP scheme based on symmetric cryptography, and show how relaxing the requirement of public verifiability allows a much more lightweight scheme. The scheme is dynamic in that it allows data blocks to be appended, modified and deleted. Erway *et al.* [12] present formal definitions of Dynamic PDP together with two protocols allowing also block insertion. PoR schemes, introduced by Juels and Kaliski [16] combine message authentication code-based data verification with error-correcting codes (ECC) to allow a client to download pre-determined subsets of blocks and check whether their MAC matches the pre-computed one: the use of ECC ensures that small changes in the data are detected with high probability.

2.1 The State-of-the-Art Solution

Next, we describe in detail the POW scheme presented by Halevi *et al.* [13], as it represents the state-of-the-art solution our solution will be compared with.

The authors present three schemes that differ in terms of security and performance. All three involve the server challenging the client to present valid sibling paths for a subset of leaves of a Merkle tree [21]. Both the client and the server build the Merkle tree; the server only keeps the root and challenges clients that claim to possess the file. This approach meets the requirements on limited I/O and computation at the server side, as the server is only required to build the tree once and only needs to store the root of the tree. The Merkle tree is built on a buffer, whose content is derived from the file, and pre-processed in three different ways for the three different schemes.

The first scheme applies erasure coding on the content of the original file; the erasure-coded version of the file is the input for construction of the Merkle tree. Informally, the rationale for such an approach is that erasure coding "spreads" with significant probability the (possibly limited) blocks of the file that are unknown to the user on a high number of blocks of its erasure coded version. The server then needs to challenge the user on a super-logarithmic number of leaves. However, this first scheme suffers from a number of shortcomings: first of all, the input to the Merkle tree construction phase is a buffer whose size is greater than the file itself; secondly, erasure coding is not I/O efficient.

The second scheme pre-processes the file with a universal hash function instead of erasure coding, to the same end: the file is hashed to an intermediate reduction buffer whose size is sufficiently large to discourage its sharing among colluding users, but not too big to be impractical. The authors settle for a size of 64 MiB. This buffer is then used as input for the construction of the Merkle tree. This scheme, although better than the previous, suffers from a very high computational cost linked to the hashing.

The third scheme, which is the one we will compare our solution with, follows the same approach as the previous, but substitutes universal hashing with mixing and a reduction phases that "hash" the original file into the reduction buffer mentioned above. Also, this variant is built on the assumption that requiring to share 64 MiB will discourage collusion attacks. In the remainder of this paper, we shall refer to this scheme as b-POW.

ALGORITHM 1: Mixing and Reduction phases of b-POW.

```
Input: An $M$-bit file $f$ split into $m = \frac{M}{512}$ 512-bit blocks.
Output: A 64 MiB reduction buffer.
$l \leftarrow \min(2^{20}, 2^{\lceil \log_2 m \rceil})$;
let $buf[]$ be an array of $l$ 512-bit blocks;
let $IV[]$ be an array of $m$ 256-bit blocks;
let $idx[][]$ be a view of IV as a $m \times 4$ indexes into buf;
$IV[0] \leftarrow SHA256\_IV$;
// Reduction Phase
for $i \in [0, m-1]$ do
    $IV[i] \leftarrow \mathrm{SHA256}(IV[i-1], f[i])$;
    for $j \in [0, 3]$ do
        $block \leftarrow$ cyclic shift of $f[i]$ by $j \cdot 128$ bits;
        $buf[IV[i][j]] \leftarrow buf[IV[i][j]] \oplus block$;
    end
end
// Mixing Phase
for $k \in [0, 4]$ do
    for $i \in [0, l-1]$ do
        for $j \in [0, 3]$ do
            $block \leftarrow$ cyclic shift of $buf[i]$ by $j \cdot 128$ bits;
            if $i \neq IV[i][j]$ then
                $buf[IV[i][j]] \leftarrow buf[IV[i][j]] \oplus block$;
            end
        end
    end
end
return $buf$;
```

Let us review both phases (see Algorithm 1) in more detail. The first phase populates the reduction buffer by xoring each block of the original file in four random positions in the buffer (after performing a bit shift). The mixing phase amplifies the confusion and diffusion in the reduction buffer by xoring together random positions of the reduction buffer.

3 System Model

The system is composed of two main principals, the client $\mathcal{C}$ and the server $\mathcal{S}$. Both $\mathcal{C}$ and $\mathcal{S}$ are computing nodes with network connectivity. $\mathcal{S}$ has a large back-end storage facility and offers its storage capacity to $\mathcal{C}$; $\mathcal{C}$ uploads its files and can later download them. During the upload process, $\mathcal{S}$ attempts to minimize the bandwidth and to optimize the use of its storage facility by determining whether the file the client is about to upload has already been uploaded by another user. If so, the file does not need to be uploaded and we say it undergoes deduplication.[1] Note that a trivial solution would be to transfer the file from the client to the server, and later have

[1]The privacy issues raised by this solution are out of the scope of this paper; some preliminary solutions have been proposed in [15].

the checks performed on the server-side. However, this solution is highly bandwidth demanding, and also sacrifices another benefit of deduplication: the reduction of the completion time on both the client and the server side.

A further requirement on the server-side is to minimize accesses to its back-end storage system: for example, a protocol that requires the server to access the file content at each interaction with a client to evaluate the potential for deduplication would not meet this requirement. We assume, however, that $\mathcal{S}$ has a front-end storage facility, whose capacity is a small fraction of the capacity of the back-end one, and that can be used to store per-file information. We finally assume that server-side computational power is abundant and cheap, especially if the required computation does not need to be executed immediately, but can be deferred to moments of low system load.

$\mathcal{C}$ is assumed to have limited resources in terms of computational power and I/O capability, and therefore one of the design guidelines of the scheme is to minimize the scheme's client-side computational and I/O footprint. $\mathcal{C}$ and $\mathcal{S}$ engage in an interactive protocol. As previously mentioned, minimization of the latency of this protocol is another important objective.

3.1 Adversarial Model

In the context of POW protocols, $\mathcal{S}$ is considered to be a trusted entity that abides by the rules of the protocol as its correct execution is in $\mathcal{S}$'s best interest. $\mathcal{C}$, in contrast, is considered to be a malicious principal and consequently it cannot be assumed that it is bound by the rules of the protocol.

Given a target file f_*, the objective of a malicious client $\mathcal{A}$ is to convince the server that $\mathcal{A}$ owns f_*, despite this not being the case. It is assumed that $\mathcal{A}$ does not know f_* in its entirety; however, we assume that $\mathcal{A}$ knows an arbitrarily large fraction of it. The estimated upper bound on the fraction of f_* known to $\mathcal{A}$ will be one of the inputs of the system, playing a role in the scheme's security analysis. Several malicious clients can collude and share information, for instance, about past protocol rounds. $\mathcal{A}$ may even collude with other clients that indeed possess f_*, and may receive arbitrary information about the file from them, including its content—but never in its entirety. However we assume that $\mathcal{A}$ cannot interact with clients that possess f_* during the challenge between $\mathcal{S}$ and $\mathcal{A}$ over f_*, as otherwise such clients could easily circumvent the security of the protocol by answering *in lieu* of $\mathcal{A}^2$.

This protocol is extremely efficient on the server side, from both a storage and a computational perspective. This comes at the expenses of poorer performance at the client side. From a security perspective, the scheme is secure as long as the adversary does not collude with a malicious user who legitimately possesses a file, and is willing to exchange more than a fixed amount of bytes. This threshold is the file size for files up to 64 MiB, and is fixed at 64 MiB for larger files. This restriction effectively voids the protection afforded by the scheme, especially for very large files (e.g. several gigabytes) where the threshold to file size ratio is very small.

4 Our Scheme

In this section, we shall describe our POW solution. Our scheme consists of two separate phases: in the first phase, the server receives a file for the first time and pre-computes the responses for a number of POW challenges related to the file. Computation of POW challenges for a given file is carried out both upon receiving an upload request for a file that is not yet present at

[2]Protection from this attack exceeds the scope of this paper.

the server side, and when the stock of the previously computed challenge/responses has been depleted. The number of challenges to be precomputed is a tunable system parameter.

The second phase is triggered by the client when it sends the server a unique identifier for a file it wishes to prove possession of. The server chooses an unused challenge from the precomputed ones for that file and sends it to the client; the client derives the response based on its knowledge of the file and sends the response to the server. The server then checks whether the client's response matches the precomputed one.

In the following sections, we will detail our scheme. We will do so incrementally, starting with an initial scheme (s-POW), and later presenting three improved variants: s-POW1, s-POW2 and s-POW3. After presenting the outcome of our benchmarking in Section 5 and introducing additional considerations in Section 6, Section 6.1 will discuss how the different variants may be combined together for practical deployments.

4.1 s-POW

The basic idea behind s-POW is that the probability that a malicious user is able to output the correct value of K bits of the file, with each bit selected at a random position in the file, is negligible in the security parameter k – assuming an upperbound on the subset of bits of the file known to the attacker. Therefore, a response to a challenge will be a bit string of size K, constructed as the concatenation of K bits of the original file, picked at K random positions.

Let us describe s-POW in more detail. The server keeps a hash-map data structure $\mathfrak{F}$ that maps strings of finite size to 4-tuples; these tuples contain a file pointer *ptr*, an array of responses $res[]$ and two indexes, id_c and id_u. The first index keeps track of the highest challenge computed so far, whereas the second index counts the number of challenges consumed. By default, both indexes are initialized to zero; $res[]$ is initialized with an array of empty strings, and *ptr* is associated with an unassigned pointer. The search key into the hash-map is the hash digest of the file; given a digest d, $\mathfrak{F}[d]$ represents the tuple mapped to d: $\mathfrak{F}[d] =\perp$ if d has not yet been associated with any tuple.

ALGORITHM 2: Server-side algorithm: the server precomputes the challenges for a file.

```
Input: A hash digest $d$; a number $n$ of responses that need to be pre-computed and a response bit
       length $K$.
Output: An updated response vector.
begin
    $f_d \leftarrow \mathfrak{F}[d]$;
    for $i \in [0, n-1]$ do
        $ctr \leftarrow f_d.id_c + i$;
        $s \leftarrow F_{S_{MK}}(d||ctr)$;
        for $j \in [0, K-1]$ do
            $pos \leftarrow F_s(j)$;
            $res[i] = res[i]||get\_bit(f_d.ptr, pos)$;
        end
    end
    $f_d.id_c = ctr + 1$;
    return $\perp$;
end
```

Also, let H be a cryptographic hash function and F_s a pseudo-random number generator taking s as seed. For the sake of simplicity, we assume that F_s generates integers ranging from zero to the size of the file in bits minus one. *get_bit* is a macro taking as input a file pointer and a bit position and producing as output the corresponding bit value. Finally, let S_{MK} be the server master secret.

Algorithm 2 describes the operations that occur at the server-side when either a new file has been uploaded, or the precomputed responses of an old file have been exhausted and new ones need to be generated. The server computes n challenges at a time: this allows optimization of the I/O operations. For each challenge, a fresh new random seed is computed using index id_c, the digest d of the file, and the server master key S_{MK}. Then, K random positions are generated using F, and the bits in the corresponding positions are concatenated to form the response to the id_c-th challenge.

Algorithm 3 describes how a client replies when challenged by the server; the client essentially uses the challenge seed s received from the server that is needed to generate the K random positions over the file, and collects the bit-value of the file at these K random positions to form the response *resp*.

ALGORITHM 3: Client-side algorithm: the client computes the response to a challenge posed by the server.

```
Input: A file $f$ and a challenge seed $s$.
Output: A response bit string.
begin
    let $res$ be an empty string;
    for $j \in [0, K-1]$ do
        $pos \leftarrow F_s(j)$;
        $res = res || get\_bit(f, pos)$;
    end
    return $res$;
end
```

Algorithm 4 shows the overall protocol executed between client and server. The protocol starts with the client computing the hash of the file and sending it to the server with a request to store the associated file. The server checks whether the file already exists in the hash map. If not, the file needs to be uploaded and no challenge takes place. If a challenge is required, the server picks the first unused challenge for the given file, computes the associated seed and sends it to the client. The client is then able to invoke Algorithm 3 and compute the response, which is sent back to the server. The server checks the response for equality with the precomputed one and outputs success or failure based on this check. At this stage, the server will assign that particular file to the set of files belonging to the client, so that later on the client can access it. Finally, if all precomputed challenges have been used up, the server invokes Algorithm 2 to repopulate the response vector.

4.1.1 Security Analysis of s-POW

As introduced in Section 3, the goal of the adversary $\mathcal{A}$ is to pass the check performed by $\mathcal{S}$ during the file uploading phase, while not owing the file in its entirety. In this way, $\mathcal{A}$ could later gain access to the file actually stored on the server. In the following, we analyse the security of our solution that is based on challenging the client on the value of K bits randomly chosen over the file that $\mathcal{A}$ claims to possess. Before exploring the security of s-POW, we remind the reader that the cryptographic digest d of the file f does not play any role in the security of the scheme, as we assume that this short value can be obtained by $\mathcal{A}$.

ALGORITHM 4: The protocol of s-POW, expressed as a distributed algorithm run between the client $\mathcal{C}$ and the server $\mathcal{S}$.

```
$\mathcal{C}$ : upon upload of file $f$ do
    $d \leftarrow H(file)$;
    send to $\mathcal{SRV}$ a store file request with $d$;
end
$\mathcal{S}$ : upon receipt of a store file request do
    if $\mathfrak{F}[d] \neq \perp$ then
        $s \leftarrow F_{S_{MK}}(d||\mathfrak{F}[d].id_u)$;
        send to $\mathcal{CLI}$ a challenge request with $s$;
    else
        initialize $\mathfrak{F}[d]$;
        receive $f$ from $\mathcal{CLI}$;
        $\mathfrak{F}[d].ptr \leftarrow f$;
    end
end
$\mathcal{C}$ : upon receipt of a challenge request do
    invoke Algorithm 3 on input $file$ and $s$ to get $res$;
    send to $\mathcal{SRV}$ the challenge response $res$;
end
$\mathcal{S}$ : upon receipt of a challenge response do
    if $resp = \mathfrak{F}[d].res[\mathfrak{F}[d].id_u \bmod n]$ then
        $\mathcal{CLI}$ succeeds;
    else
        $\mathcal{CLI}$ fails;
    end
    $\mathfrak{F}[d].id_u = \mathfrak{F}[d].id_u + 1$;
    if $\mathfrak{F}[d].id_u \equiv 0 \bmod 0$ then
        invoke Algorithm 2;
    end
end
```

In accordance with the working hypothesis given in Section 3, we can assume that $\mathcal{A}$ owns (or has access to) a fraction $p = (1-\epsilon)$ of the file. When confronted with a single-bit challenge posed by the server, two cases can occur: the requested bit belongs to the portion of the file available to $\mathcal{A}$ – let us indicate this event with w. This can happen with probability: $P(w) = (1-\epsilon)$. Otherwise, we can assume $\mathcal{A}$ performs a (possibly educated) guess that results in a success probability g. Therefore, $\mathcal{A}$ can succeed on a single-bit challenge ($P(succ_1)$), under the assumption that $\epsilon > 0$, with probability

$$\begin{aligned} P(succ_1) &= P(succ_1 \wedge (w \vee \bar{w})) \\ &= P(succ_1|w)P(w) + P(succ_1|\bar{w})P(\bar{w}) \\ &= P(w) + gP(\bar{w}) \\ &= (1-\epsilon) + g(1-(1-\epsilon))) \\ &= 1 - \epsilon(1-g) \end{aligned}$$

However, $\mathcal{A}$ is confronted with K challenges, each being i.i.d. from the others. Therefore, the probability that $\mathcal{A}$ can successfully pass the check ($P(succ)$) is

$$P(succ) = (1-\epsilon(1-g))^K \tag{1}$$

Equation 1 completely characterizes our security model. Indeed, once reasonable values have been set for ϵ and g, and given a security parameter k, an appropriate value of the parameter K of s-POW, assuring that $P(succ) \leq 2^{-k}$, can simply be derived as:

$$K = \left\lceil \frac{k \ln 2}{\epsilon(1-g)} \right\rceil \tag{2}$$

Note that in its present formulation (cf. Algorithm 2) – s-POW trades information-theoretical security for improved space efficiency, by deriving challenge seeds from a master secret. A simple way of achieving information-theoretical security would be to generate a fresh random seed s for each new challenge in Algorithm 2, and to save it together with its pre-computed response.

Finally, note that equations 1 and 2 also highlight that K is not affected by the length of the file, because the only parameters involved are ϵ, the fraction of the file *unknown* to $\mathcal{A}$, and the value g.

Let us now focus on how the scheme tolerates data leakage from colluding file owners. Recall that Halevi *et al.*'s scheme [13] can only tolerate exchanges between the adversary and a colluding file owner of up to 64 MiB (for large files), and of up to the file size (for smaller ones); indeed, if the adversary could receive more than this amount of data, it could require the colluding file owner to send the reduction buffer, thus making it able to run a successful PoW exchange. s-POW (and its subsequent optimizations) capture this aspect by means of the system parameter $p = (1-\epsilon)$ described above. Notice also that, as this parameter represents a fraction of the total file size, it grows with the file size as opposed to the fixed threshold chosen in [13], whose ratio to the file size decreases as the file size increases.

4.2 s-POW1: challenging at the block-level

In most modern operating systems and file systems, the cost of reading a single bit from a file is comparable (if not equal) to the cost of reading an entire disk sector – usually an aligned, contiguous 512 byte block of data. Consequently, s-POW1, a natural extension of s-POW, is a scheme in which the user is challenged to prove knowledge of K_{B} data blocks chosen in random positions. Intuitively, this solution allows the client to provide a stronger proof of its knowledge of the file to the server at the same I/O cost, or, alternatively, to provide comparable assurance at a lower cost.

Algorithms 2 through 4 can be adapted to describe s-POW1 in the following simple manner: first of all, let B be the block-size of a file with total size F. Then in s-POW1, the pseudo-random number generator F_s should generate integers ranging from zero to $\lceil \frac{F}{B} \rceil - 1$. The *get_bit* macro is replaced with *get_block*, taking the same arguments and returning the content of the block of the file indexed by the second argument. These changes would entail that the size of the response res is increased by a factor B; this change impacts both the size of responses and the storage required at the server side. Whenever this is not acceptable, the content of res can be processed with a cryptographic hash function H' prior to i) storing it in Algorithm 2; and, ii) returning it in Algorithm 3.

4.2.1 Security Analysis of s-POW1

In the security analysis of s-POW we assume that the adversary knows a fraction $p = (1-\epsilon)$ of the original file and is challenged on K_{b} single bits at random positions (we will use the subscript b to refer to bits and the subscript B to refer to blocks).

Let us now analyze the security of a scheme in which the adversary still knows only a fraction $p = (1-\epsilon)$ of the file and is now challenged on K_{B} contiguous blocks of size B of the file (out of a file with total size F).

To better capture the behaviour of the adversary, let us introduce two integers n_{b} and n_{B}: we assume that the adversary knows n_{B} blocks of the file and that n_{b} single bits amongst the blocks that are still unknown. To compare the security of s-POW1 with that of s-POW, the portion of the file known to the adversary should not change. Hence, the choice of n_{B} and n_{b} is subject to

$$p = (1-\epsilon) = \frac{n_{\mathsf{B}}B}{F} + \frac{n_{\mathsf{b}}}{F}$$

We also define $p_{\mathsf{B}} = \frac{n_{\mathsf{B}}B}{F}$ as the probability that the adversary knows a given block, and $p_{\mathsf{b}} = \frac{n_{\mathsf{b}}}{F}$ as the probability that the adversary knows a single bit of an unknown block. Hence we can write $p = (1-\epsilon) = p_{\mathsf{B}} + p_{\mathsf{b}}$.

Let $\mathtt{succ}_{\mathsf{B}}$ and $\mathtt{succ}_{\mathsf{b}}$ be the events that the adversary successfully answers the query for a random block of the file and for a random bit drawn from within an unknown block, respectively. Let also $\mathtt{know}_{\mathsf{B}}$ and $\mathtt{know}_{\mathsf{b}}$ be the events that the adversary knows a block and a bit from the remaining blocks, respectively. Then

$$\begin{aligned}
P(\mathtt{succ}_{\mathsf{B}}) &= P\left(\mathtt{succ}_{\mathsf{B}} \cap \left(\mathtt{know}_{\mathsf{B}} \cup \overline{\mathtt{know}_{\mathsf{B}}}\right)\right) \\
&= P\left(\mathtt{succ}_{\mathsf{B}} | \mathtt{know}_{\mathsf{B}}\right) P\left(\mathtt{know}_{\mathsf{B}}\right) + P\left(\mathtt{succ}_{\mathsf{B}} | \overline{\mathtt{know}_{\mathsf{B}}}\right) P\left(\overline{\mathtt{know}_{\mathsf{B}}}\right) \\
&= p_{\mathsf{B}} + (1-p_{\mathsf{B}}) P\left(\mathtt{succ}_{\mathsf{b}}\right)^{B} \qquad (3) \\
&= p_{\mathsf{B}} + (1-p_{\mathsf{B}}) \left(P\left(\mathtt{succ}_{\mathsf{b}} | \mathtt{know}_{\mathsf{b}}\right) P\left(\mathtt{know}_{\mathsf{b}}\right) + P\left(\mathtt{succ}_{\mathsf{b}} | \overline{\mathtt{know}_{\mathsf{b}}}\right) P\left(\overline{\mathtt{know}_{\mathsf{b}}}\right)\right)^{B} \qquad (4) \\
&= p_{\mathsf{B}} + (1-p_{\mathsf{B}}) \left(p_{\mathsf{b}} + (1-p_{\mathsf{b}})g\right)^{B} \qquad (5)
\end{aligned}$$

where, as before, g is the probability that the attacker guesses a single unknown bit. Step 3 follows from the fact that answering the challenge for an unknown block successfully boils down to answering the challenge for its B single bits successfully; steps 4 and 5 follow from the way p_{B} and p_{b} have been defined, in particular, from the fact that p_{b} has already been defined as the probability of knowing a bit given that the block it belongs to is unknown to the adversary.

The adversary is then challenged over K_{B} different blocks in random positions; hence, the probability of success of the adversary can be defined as

$$\begin{aligned}
P(\mathtt{succ}) &= P(\mathtt{succ}_{\mathsf{B}})^{K_{\mathsf{B}}} \\
&= \left(p_{\mathsf{B}} + (1-p_{\mathsf{B}}) \left(p_{\mathsf{b}} + (1-p_{\mathsf{b}})g\right)^{B}\right)^{K_{\mathsf{B}}} \qquad (6)
\end{aligned}$$

From Equation 6 we can derive a lower bound for the parameter K_{B} expressed as

$$K_{\mathsf{B}} = \left\lceil \frac{k \ln 2}{(1-p_{\mathsf{B}})\left(1 - \left(p_{\mathsf{b}} + (1-p_{\mathsf{b}})\, g\right)^{B}\right)} \right\rceil \qquad (7)$$

Note that by setting $B = 1$ and $n_{\mathsf{B}} = 0$ in Equation 6, we reach the same conclusion as that of Equation 2.

Through Equation 6 we can study the different strategies of the adversary, whether it decides to cluster its knowledge of the file in contiguous blocks or to disperse it across single scattered bits. In practice, we shall then choose $\max(K_{\mathsf{B}})$ subject to $p = (1-\epsilon) = p_{\mathsf{B}} + p_{\mathsf{b}}$ to obtain

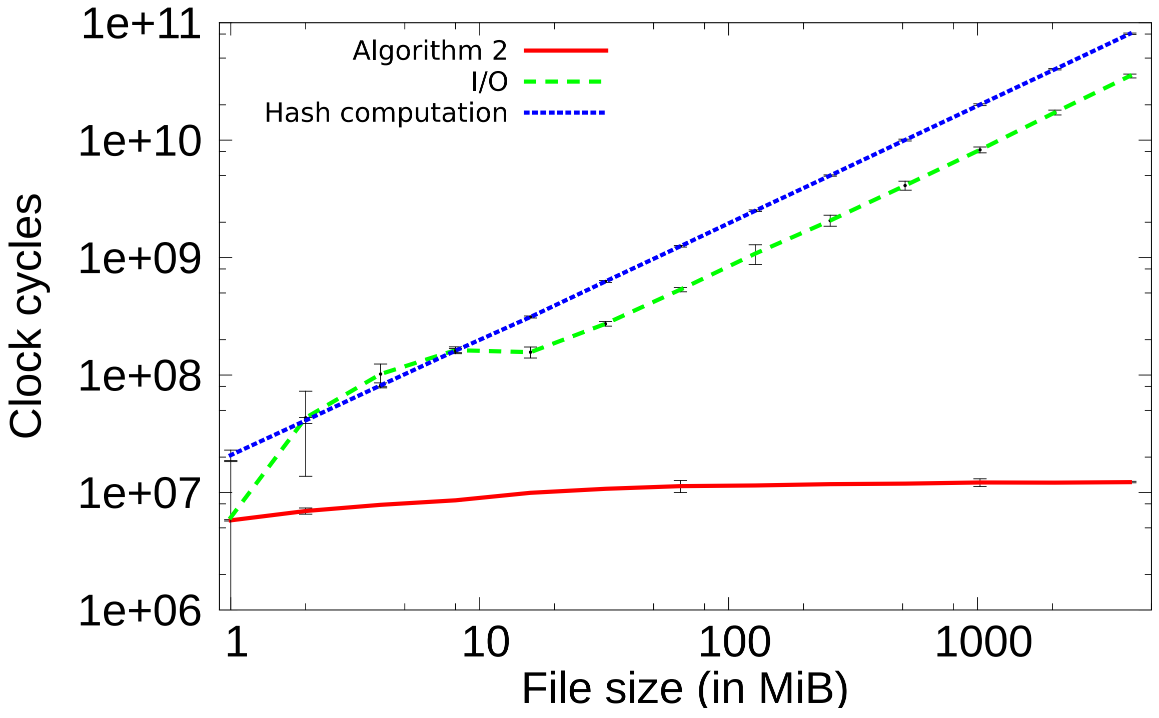

Figure 1: Comparison of the running time of each of the three main operations executed in s-POW as the input file size grows.

optimal security against the best strategy of the adversary. The optimal strategy against s-POW1, i.e. the one that requires the highest value of K_{B}, corresponds to the one in which the attacker clusters knowledge of the file into blocks of size B and, if challenged on an unknown block, attempts to guess its content; this can be modelled by setting $p_{\mathsf{b}} = 0$ and consequently $p_{\mathsf{B}} = p = (1 - \epsilon)$. If we hold the security parameter constant and assume that the cost of reading a bit is comparable to that of reading the entire block that contains it, we can compare equations 2 and 7 and see that in s-POW1 the I/O cost is a factor of $\frac{1}{(1-g)}$ lower.

4.3 s-POW2

In this section, we propose a further improvement of s-POW. Given the extreme simplicity of s-POW, it may seem that there is very little room for optimization. Indeed, as we have seen in the previous sections, the number K of bits (or blocks) used to challenge the client is a function of the security parameter of the scheme. Therefore, any reduction in K will inevitably alter the overall security of the scheme.

Section 5 describes the results of the benchmarking on our scheme. We will, however, already mention one of the results here to give the reader an idea of our improved scheme: Figure 1 shows the evolution of the clock cycles spent by the client in the execution of the three main components of s-POW as the size of the file grows. The three components are the I/O time, i.e., the time spent to access the file on disk and to load it into main memory, the hash time, i.e., the time spent in the computation of the hash digest of the file, and finally the time spent in the execution of Algorithm 3. Notice that the I/O and the hash time are by far more expensive than that of Algorithm 3. It is natural therefore to try to reduce the cost of these two components.

Let us recall that the computation of the hash is required because the server needs to be able to uniquely identify the file being uploaded among those already stored (if this is the case) to compute the appropriate challenge seed and to compare the response of the client with the pre-computed one. However, the cryptographic properties of standard hash functions (one-wayness, preimage resistance and second preimage resistance) are not strictly needed in this setting. Indeed, the properties we are looking for are such that the hash function may be replaced with another function that: i) has a small probability of producing the same output given different files as input; ii) is computationally less expensive than a hash; and, iii) minimizes the required I/O.

Algorithm 3 is an excellent candidate for such a function, as it has a very small computational footprint and requires only a minimum number of I/O operations to retrieve the bits that constitute the output of a challenge. Consequently, we modify the overall protocol as shown in Algorithm 5.

ALGORITHM 5: Changes in the protocol of s-POW to achieve s-POW2.

```
Input: A file $f$.
$\mathcal{C}$ : upon upload of file $f$ do
    invoke Algorithm 3 on input $file$ and $s_{pub}$ to get $d$;
    send to $\mathcal{SRV}$ a store file request with $d$;
end
```

The main difference with respect to the original version of the protocol for s-POW as shown in Algorithm 4 is that – at the client-side – Algorithm 3 (on input of a public seed s_{pub}, randomly generated and published as a parameter of the system) is invoked instead of the hash function H to generate the file digest d. As we shall see in Section 5, this change achieves a significant improvement in the performance of the scheme, especially at the client-side. Indeed, it is no longer necessary to scan the entire file to compute its hash; it suffices to perform a relatively small number of random accesses to its content.

To tolerate the scenario in which multiple files have the same digest d (produced by the invocation of s-POW on input of the public seed), the server has to keep a one-to-many map (instead of the previously used one-to-one map) between the output of the indexing function and the tuple containing indexes, file pointer, and array of pre-computed responses. In this scenario, the server would receive a single *resp* and compare it with all the pre-computed responses for all files indexed by the same value of d. If none of them matches, the client has to upload the new file. If one matches, the server concludes that the client owns the file associated with the matching precomputed response, and deduplication can take place. However, this approach comes at the expense of a slightly higher usage rate of the precomputed challenges. Indeed, imagine that there are two files *f1* and *f2* with the same digest d: a client owning *f1* engages in the POW protocol with the server and receives a challenge seed $F_{S_{MK}}(d||i)$ for some value i of the current counter; that seed can no longer be used for a client owning *f2* because if the challenge is leaked, a user colluding with $\mathcal{A}$ could precompute the correct response and send it to $\mathcal{A}$. Not reusing challenges that have been disclosed implies that the usage rate of challenges for files indexed by the same key d is equal to the sum of the rates of requests for each of these files. However, this does not constitute a problem, because the server has abundant computational power and can regularly schedule the pre-computation of challenges in periods of low system load.

4.3.1 Security Analysis of s-POW2

The security of the scheme is unchanged: indeed, even if an attacker were able to produce the correct value d for a given file (for instance, by receiving it from an accomplice), it would still need to generate the correct response to the challenge of the server. However, we have shown in Section 4.1.1 that the probability of this happening is negligible in the security parameter.

The cryptographic hash function H previously used for indexing was collision resistant by definition. As explained above, H has been replaced with an s-POW invocation on input of a public seed. We therefore need to quantify the collision probability of such an indexing function, i.e. the probability that the digests $d1$ and $d2$ of two different files $f1$ and $f2$ are equal. We can derive this probability by assuming that two files are similar with a given probability z (z expresses the percentage of the bits of the two files in the same position that show the same value). Hence, for M files we have the probability of collision $P(coll)$:

$$P(coll) \leq \binom{M}{2} P(d1 = d2) \leq \frac{M^2}{2} P(d1 = d2) = \frac{M^2}{2} z^K \tag{8}$$

where K is the parameter of s-POW mentioned in the previous section. The above probability can still be considered negligible for practical instantiations of the scheme. For instance, for $M = 10^9$, $z = .95$ and[3] $K = 1830$, $P(coll) \leq 2^{-75}$.

However, let us take a conservative stance and assume that collisions *do* happen. Then, as explained above, an invocation of the mapping $\mathfrak{F}$ on d would return a set of m files. We then need to quantify the additional advantage that an adversary might have in passing the proof of ownership, given that the server has to compare the client's response with m pre-computed ones instead of a single one. Let r_i be the event that $resp_*$, the response received by $\mathcal{A}$, equals $resp_i$, the i-th precomputed response of one of the m files in the set. Then, it follows that the probability $P(succ)$ of $\mathcal{A}$ to pass the check over at least one out of the m files is:

$$\begin{aligned} P(succ) &= P(r_1 \vee \ldots \vee r_m) \leq mP(succ_{r_i}) \\ &= m(1 - \epsilon(1 - g))^K \end{aligned} \tag{9}$$

where the term $(1 - \epsilon(1 - g))^K$ comes from Equation 1. From 9 we conclude that m can become an additional parameter of the system and can contribute to the determination of the parameter K, even though its effect over K is scaled down by a logarithmic factor: major changes in m will have very little effect on the value of K.

Another aspect we need to consider for the case where an invocation of the mapping $\mathfrak{F}$ on d returns a set of m files, is the probability that there are collisions among the m pre-computed responses for a given value of the index $\mathfrak{F}[d].id_u$; that is, the $\mathfrak{F}[d].id_u$-th pre-computed response for file f_i is equal to the $\mathfrak{F}[d].id_u$-th pre-computed response of file f_j, $i \neq j$, and the digest d_i of f_i is equal to the digest d_j of f_j. However, this happens with a negligible probability as shown in Equation 8, by substituting M with m; moreover, $m << M$.

4.4 Distribution of File Sizes and s-POW3

Further improvements might be achieved if another, less expensive candidate for the indexing function of the file could be found. Here, we consider using the size $f.size$ of a file f as a candidate for the indexing function. This approach clearly meets the last two requirements outlined in Section 4.3, because it optimizes both I/O and computation.

[3]See Section 5 on the sizing of the parameter K.

ALGORITHM 6: Changes in the protocol of s-POW to obtain s-POW3.

```
Input: A file $f$.
$\mathcal{C}$ : upon upload of file $f$ do
    $d \leftarrow f.size$;
    send to $\mathcal{SRV}$ a store file request with $d$;
end
```

Algorithm 6 shows the changes to the client-side introduced in this version of the protocol. As we can see, no computation – besides determining the size of the file – is required on the client side.

We have explained in Section 4.3 how to cope with collisions in the indexing. However, we still need to verify whether the file size constitutes a good indexing function, i.e. whether in practice the likelihood that two different files have the same size is tolerably small.

To this end, we have studied the distribution of file sizes of the entire Agrawal *et al.* dataset [1], containing information about over 4.2 billion files. The dataset captures (among other information) the sizes of the files observed in the computers of a large corporation.

The objective of our analysis is to verify the intuition that – especially for large files, i.e. those for which computing another indexing function is more expensive – the size of a file can become a very effective file indexing function. To this end, we have extracted from the dataset a unique file identifier (namely, the hash of the filename) and the respective file size. After purging doubles, we counted the number of files with equal size. Figure 2 shows the results of the analysis. The figure shows four curves: one curve plots the total number of files per bin (with power-of-two bins); the other three curves plot, respectively, the minimum, median, and maximum number of files with the same size per bin. Notice that we have only counted *different* files with the same size: we used the hash of the file to establish whether two files were the same.

First of all, we can observe that the minimum and median number of files with the same size are relatively constant up to around 10 KiB; after this threshold both curves plunge. The size starts to behave (on average) as a unique identifier for files larger than approximately 1 MiB, even though the number of files in the considered bin is still relatively high: for example, in the 1 MiB to 2 MiB range, we have 39,514,848 files, and the median number of files with the same size is 4. Clearly, the number of files with the same size decreases also because each bin is wider and less and less populated: however, Figure 2 portrays what can (very likely) be considered the distribution of the files a file-storage service may receive as input from its customers—and it is therefore of high significance for this paper. Far from claiming to be exhaustive, our study nonetheless strongly supports the use of the file size for indexing purposes in our scenario.

4.4.1 Security Analysis of s-POW3

Similar considerations as those made in Section 4.3.1 apply to s-POW3: the influence of the number of files with the same size on the choice of the system's parameters has been captured in Equation 9. Intuitively, the approach suggested in s-POW3 is particularly effective for files with large size, because: i) as shown, the probability of collision is low; and, ii) avoiding the computation of another indexing function on a very large file is particularly cost-effective.

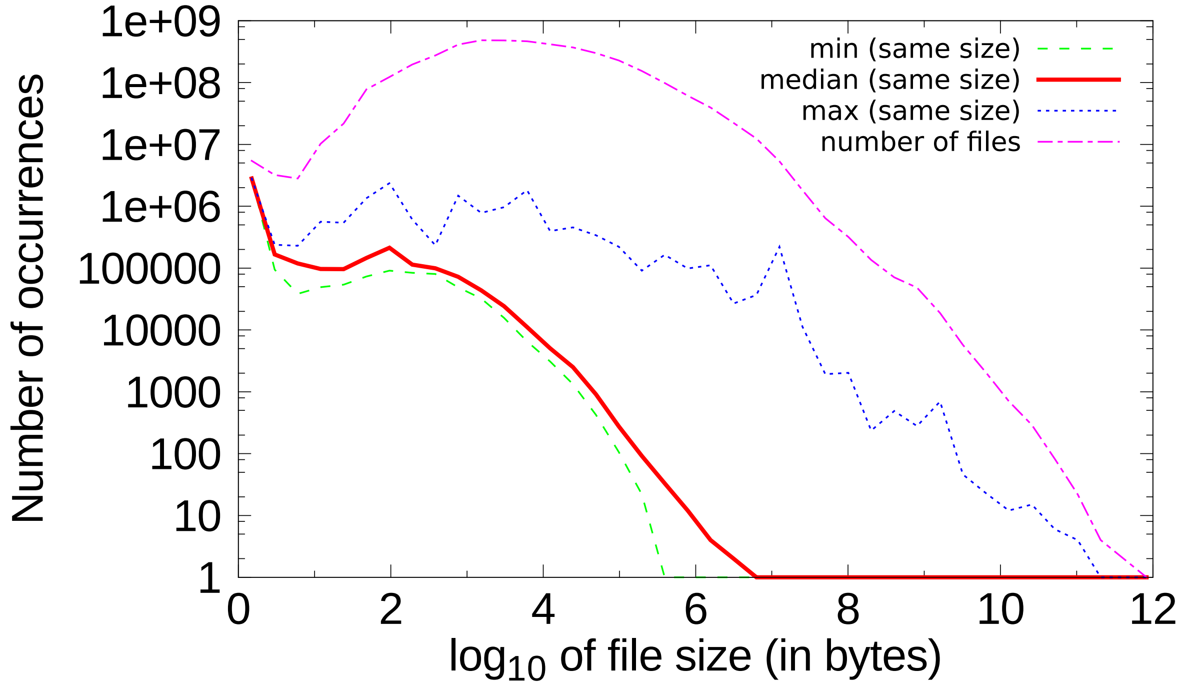

Figure 2: Plot of the number of files for each bin (with power-of-two bins), and of the minimum, median and maximum number of different files with the same size per bin (with power-of-two bins) for the Agrawal *et al.* dataset [1].

5 Running POW

To evaluate the effectiveness of our proposals, we have implemented both b-POW and s-POW and its two variants. The code has been developed in C++ using the OpenSSL crypto library for all cryptographic operations and using the Saito-Matsumoto implementation [23] of the Mersenne twister [19] for the pseudo random number generator. The code implements both the client-side and the server-side of all schemes. The interactions between client and server as well as the data exchange have been virtualised so as to not consider networking-related delays and to focus only on local (client and server) I/O and computation.

5.1 Experimental Settings

We have run our experiments on a 64-bit RedHat box with an Intel Xeon 2.27GHz CPU, 18 GiB of RAM and an IBM 42D0747 7200 RPM SATA hard disk drive. All schemes operate on input files with randomly generated content; the input file size ranges from 1 MiB to 4 GiB, with the size doubled at each step. The files are reasonably well defragmented, with a maximum of 34 different extents on the 4 GiB file.

The parameters for b-POW have been chosen in strict adherence with the choices made in [13]. We have also used the same security parameter $K = 66$. Our scheme has two parameters, $\epsilon = (1 - p)$ and g. The values of these parameters are needed to derive a value for K in Equation 2. We have chosen p, the upper bound on the fraction of the file known to the adversary, as $p \in \{0.5, 0.75, 0.9, 0.95\}$. The parameter g measures the probability that the adversary successfully guesses the value of a bit without knowing it. To assign a reasonable

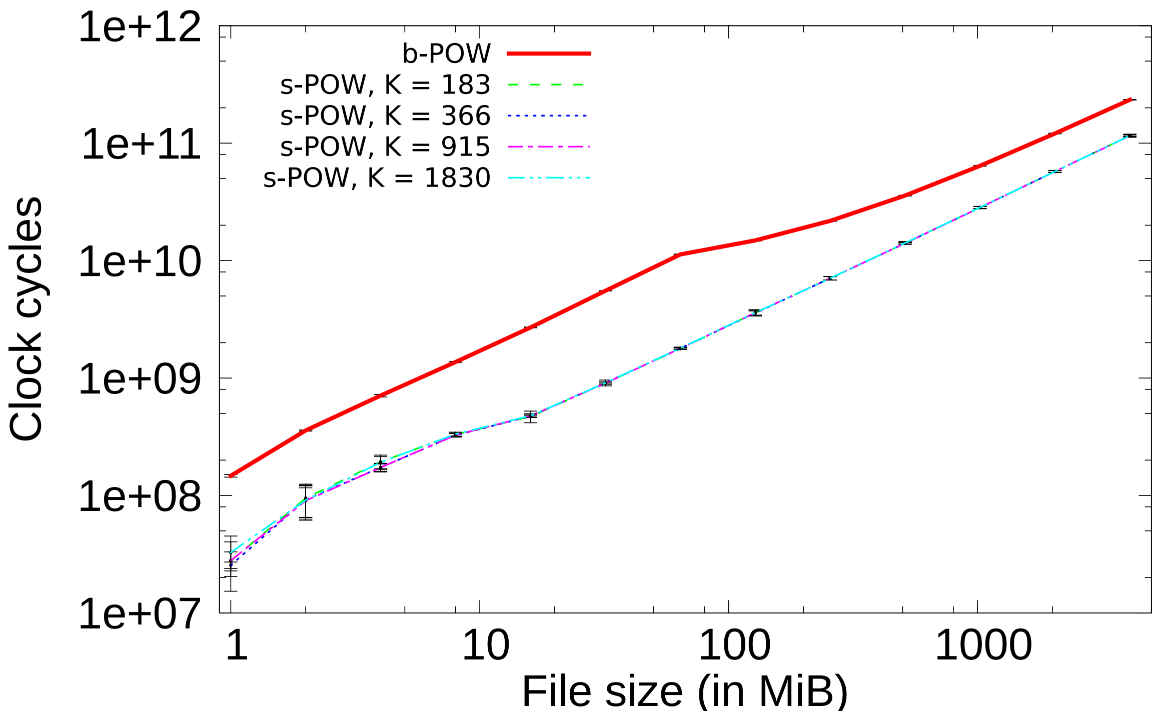

Figure 3: Comparison of the running time for the client side b-POW with that of s-POW for different values of K as the input file size grows.

value to g, we have analysed what the probability of guessing a bit in an ASCII file with lower-case letters written in the English language is, arguably a relatively conservative case with low entropy in the input distribution. Given the letter frequency analysis in [17], the probability that a given bit equals one is 0.52731. In addition, Equation 2 shows that slight changes in the value of g do not noticeably affect the value K. We have therefore chosen $g = 0.5$.

Each configuration has been run at least 200 times; before each repetition, cached data, dentries and inodes have been flushed (at both the client and the server-side) to ensure accurate measurements. To perform the comparison of the different schemes, the code has been instrumented by surrounding relevant code blocks and functions with calls to extract the Intel Time Stamp Counter through the RDTSC assembly instruction. The figures below have been generated by reporting the mean value and the standard deviation (using a box plot) of the extracted clock cycle count.

5.2 Client-Side

Here we compare the client-side performance of b-POW with those of s-POW and s-POW2[4]. The implementation of b-POW first loads the file into main memory, where the various phases of the scheme are performed: the reduction phase (which also results in the computation of the SHA256 hash digest of the file), the mixing phase, and the calculation of a binary Merkle tree on the resulting reduction buffer.

On the client-side of s-POW, the input file is loaded into memory, the hash digest is com-

[4]We have chosen not to clutter the graphs by also including a comparison with s-POW1 and s-POW3 because the performance of these two schemes is identical to that of s-POW, modulo a constant.

puted and then Algorithm 3 is executed. Figure 3 shows the results of the experiments assessing the performances of the two competing solutions: s-POW is faster than b-POW– from ten times to twice as fast. The complexity of both schemes grows at an approximately equal rate as the input file size grows. The reason for this is that – as mentioned – reading the file and computing the hash are by far the predominant operations for both schemes. The discontinuity in the curve of b-POW– noticeable around 64 MiB – is due to the fact that 64 MiB is the maximum size for the reduction buffer. Therefore, the computational cost of the reduction phase reaches its maximum at 64 MiB and remains constant afterwords.

For s-POW2, the computation of the hash is replaced by an initial invocation of Algorithm 3. Note that, as access to the entire content of the file is no longer needed, the file is no longer loaded into memory. Indeed, only random disk accesses are needed to fetch the required bits. Figure 4 shows that this second version improves the scheme's performance with respect to that of b-POW. We can see how the computational cost of our scheme reaches a plateau for sufficiently large files, because – regardless of the input file size – the computation required is essentially constant. The growth rates of the two schemes are now markedly different: b-POW grows linearly with the input file size, whereas s-POW2 is asymptotically constant. In addition, the influence of the parameter K starts to become appreciable.

5.3 Server-Side

At the server-side, we have identified two main phases: the initialization phase and the regular execution phase. The initialization phase corresponds to the first upload of the file; in both schemes, this phase starts with the computation of a hash digest of the file. Then come the reduction and mixing phases for b-POW, or Algorithm 2 (with n set to 10,000) for all variants of s-POW: note that all variants of s-POW are indeed equivalent on the server side, since Algorithm 2 never changes. We therefore only compare b-POW with s-POW. The implementation of Algorithm 2 has been optimized by pre-computing all bit position indexes at once (for all n pre-computed challenges) and by sorting them before performing the file access operations to fetch the corresponding bits. This optimization (simple but effective) allows us to only have to scan the file at most once, thus avoiding the performance penalty associated with random, non-sequential file accesses.

The regular execution phase includes the operations that have to be executed by the server upon each interaction with the client. In b-POW, this phase requires verification of the correctness of the sibling path in the computed Merkle tree for a super-logarithmic number of leaves (we have picked this number to be 20 as in [13]). In contrast, in our scheme, if we factor out the table lookup required to retrieve (from the file index received) the correct data structure holding the state for the given file, our protocol only needs to verify the equality of two short bit strings. The related overhead is therefore negligible. However, our scheme also requires regular re-executions of Algorithm 2 to pre-compute new challenges: we will therefore include this in the regular execution phase.

Figure 5 shows the performance of the initialization phase: the cost of b-POW grows – as explained in the previous section – at the same rate as the cost of reading the entire file. Our scheme exhibits an essentially constant computational cost up to a certain point, and a cost similar to that of b-POW (linear with the cost of reading the entire file) from that point on. The reason for this is that the overhead of generating the $n \cdot K$ challenges, sorting them and maintaining the data structure with all bit vectors, is constant: for small files, this overhead is higher than the cost of reading the entire file and thus prevails. However, once the input file has reached a critical size, the cost of reading the file becomes dominant. The reason for

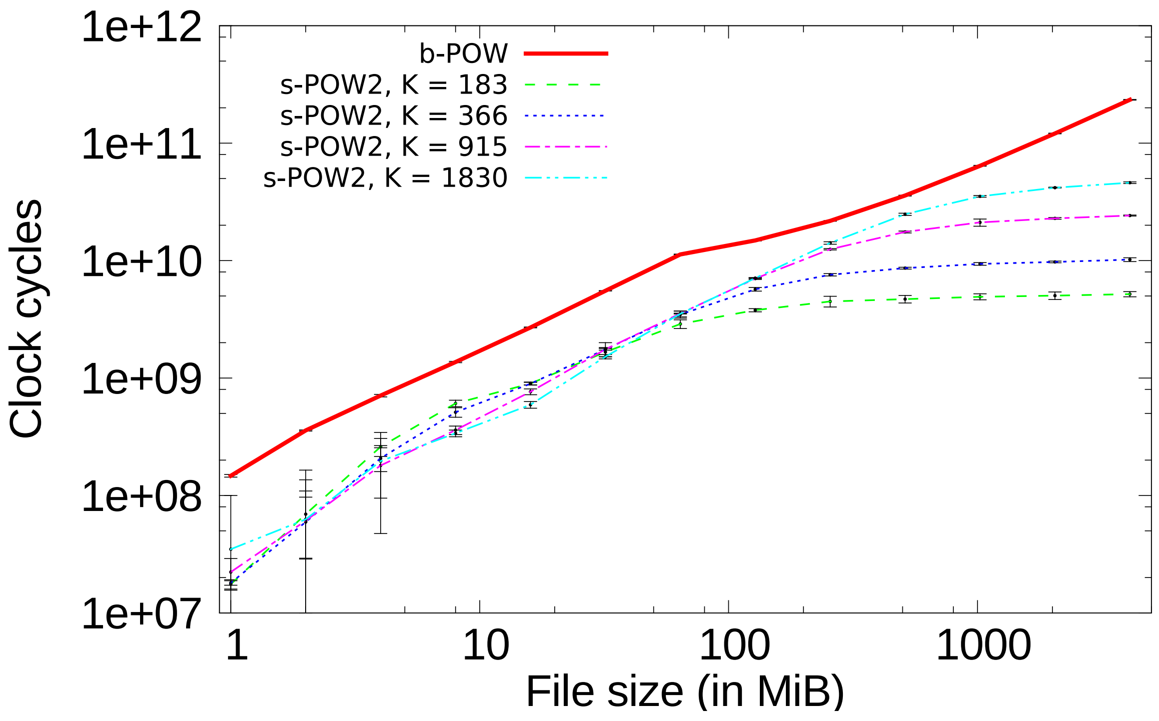

Figure 4: Comparing the running time of b-POW with that of s-POW2 for different values of K as the input file size grows (client-side).

this asymptotic behaviour is that, with high probability, reading $n \cdot K$ bit positions in the file requires fetching most data blocks of the file, which is roughly equivalent to reading the entire file.

Figure 6 compares the performance of 10,000 repetitions of the regular execution phase for both schemes. b-POW exhibits an essentially constant computational cost as the number of leaves of the Merkle tree is relatively low and does not grow past 64 MiB. The computational costs for this phase of our scheme are the same as those shown in Figure 5 minus the hash computation which is no longer required. We would like to emphasize however that Figure 6 shows a comparison between the *on-line* computation required by the verification phase of b-POW and the *off-line* computation required to generate the challenges: the former requires readily available computation power regardless of the load of the system (because delaying client requests is usually not acceptable), whereas the latter is a computation that can be carried out when the system load is low.

5.4 Benchmarking on SSDs

To assess the influence of faster storage technologies on the performance of the scheme, we have performed a further analysis in which we replaced the HDD with an OCZ Vertex 2 SATA II *solid state* drive. In this way, the effects of the I/O bottleneck can be minimized.

5.4.1 Client side

On the client side we have chosen to compare b-POW with s-POW2: a comparison with s-POW, with both schemes (b-POW and s-POW) requiring the client to read the input file in its

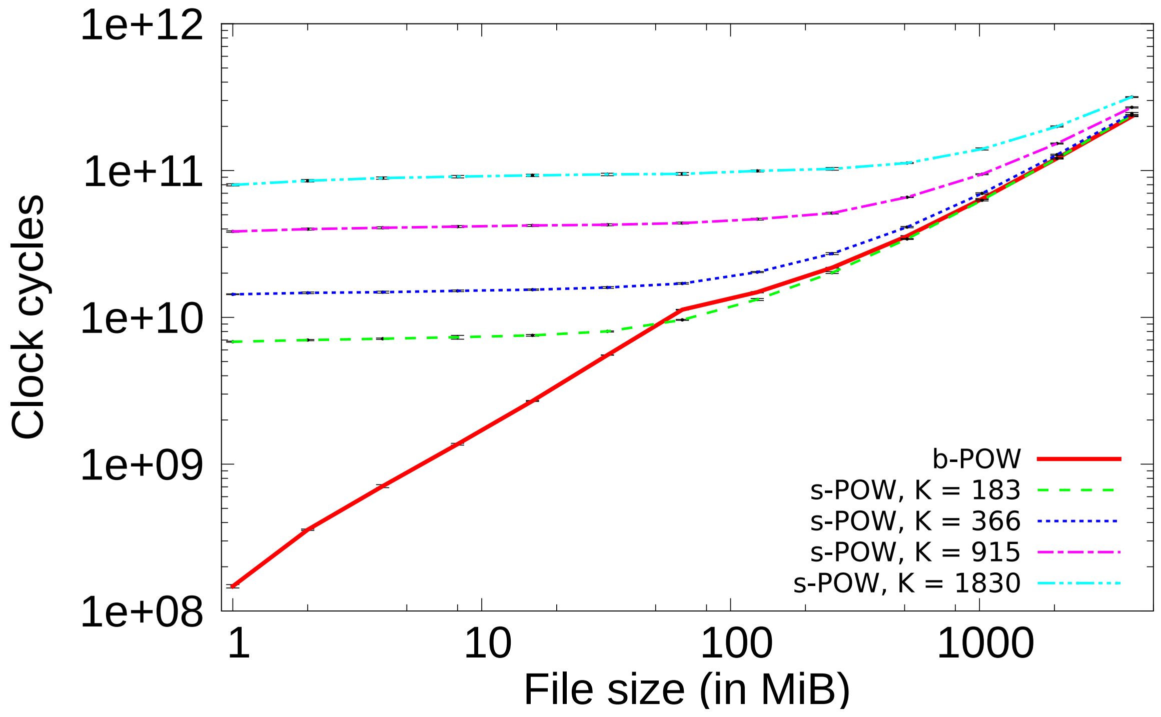

Figure 5: Comparing the running time of the server initialization phase of b-POW with that of s-POW for different values of K as the input file size grows.

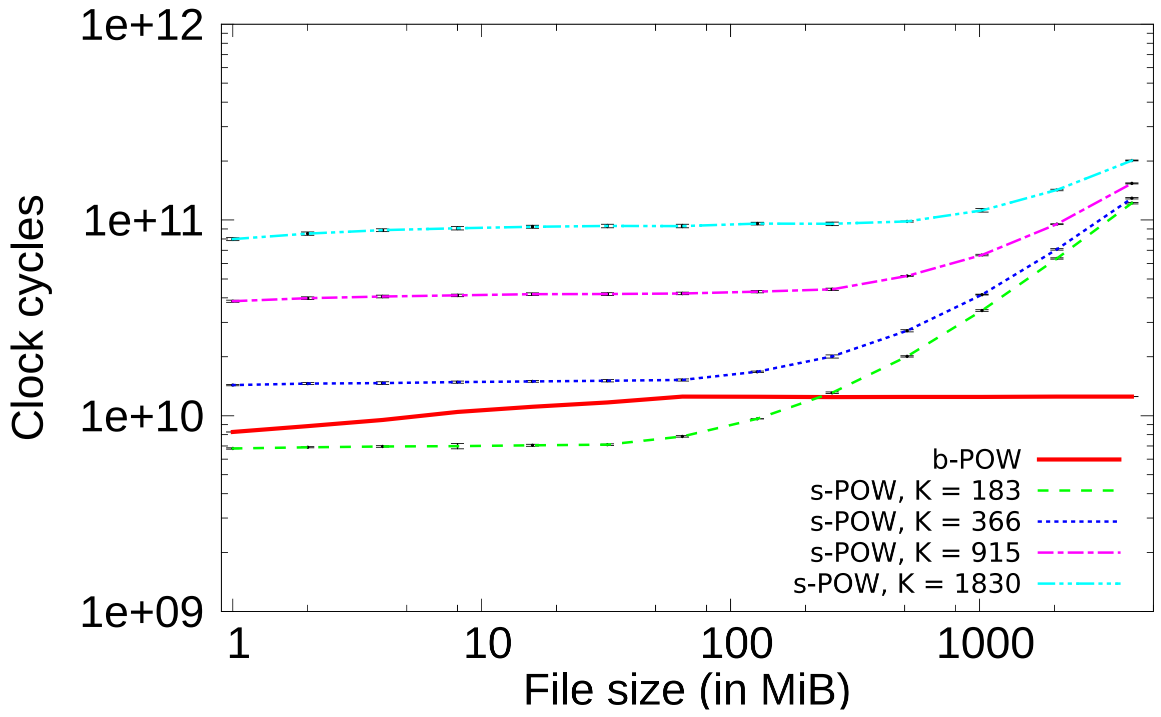

Figure 6: Comparing the running time of the server regular execution phase of b-POW with that of s-POW for different values of K as the input file size grows.

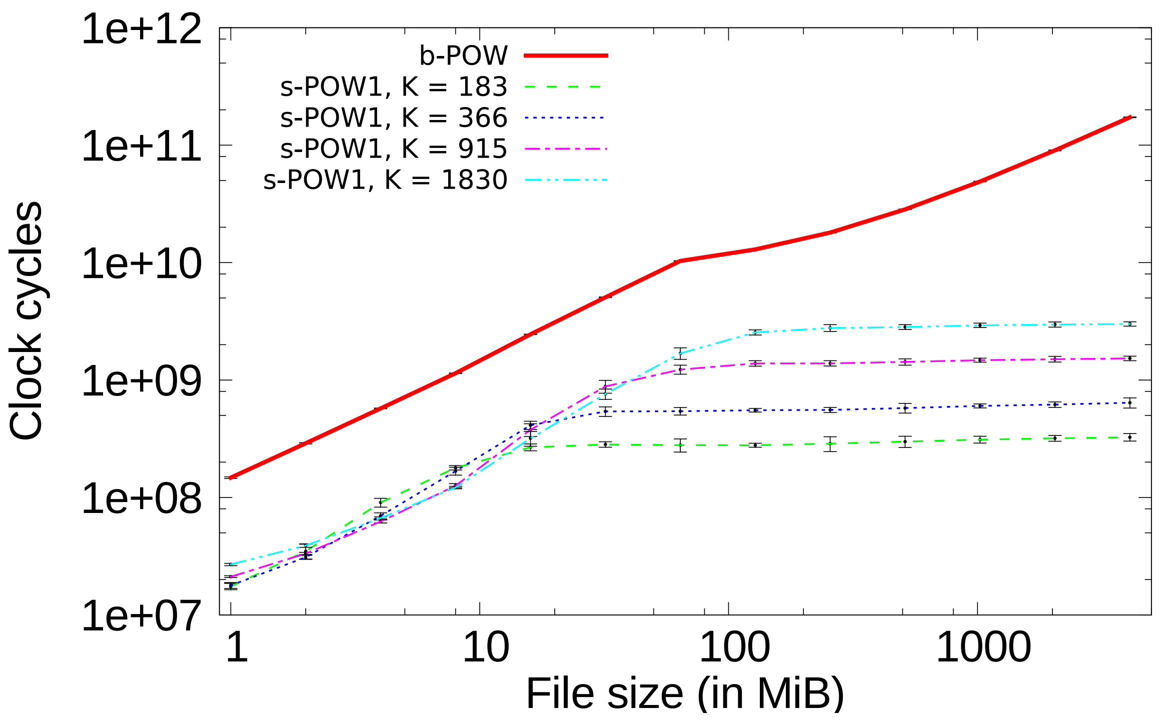

Figure 7: Comparison of the running time of b-POW against that of s-POW2 on Solid State Drives, for different values of K, as the input file size grows.

entirety, does not provide any real additional insight. As s-POW2 is the only scheme variant that does not require the client to access the file in its entirety, benchmarks on SSDs can show the real impact of this design choice.

Figure 7 shows the results of the experiment for the client side: the gap between the two schemes becomes considerably more pronounced as a relatively small number of (uncached) random accesses – the ones performed by s-POW2– is much faster on SSDs than full sequential access – required by b-POW. Hence, the effects of the lower I/O demands of our scheme become more noticeable. This analysis is particularly relevant as SSDs are becoming cheaper and cheaper and have already become standard components of commodity PCs. Furthermore, Figure 7 confirms that past a certain threshold, the cost of our scheme becomes independent of the input file size and essentially constant.

5.4.2 Server Side

Figures 8 and 9 show the results of the comparison between the server-side of b-POW and s-POW when run on solid state drives (recall that, on the server side, there are no major differences between s-POW and s-POW2). As remarked in the previous section, our scheme becomes much faster because random accesses are much faster than sequential ones on SSDs. In particular, Figure 8 shows that the growth rate of the two schemes are noticeably different in the initialization phase. For the regular execution phase, we can draw similar conclusions as for the case with standard HDD.

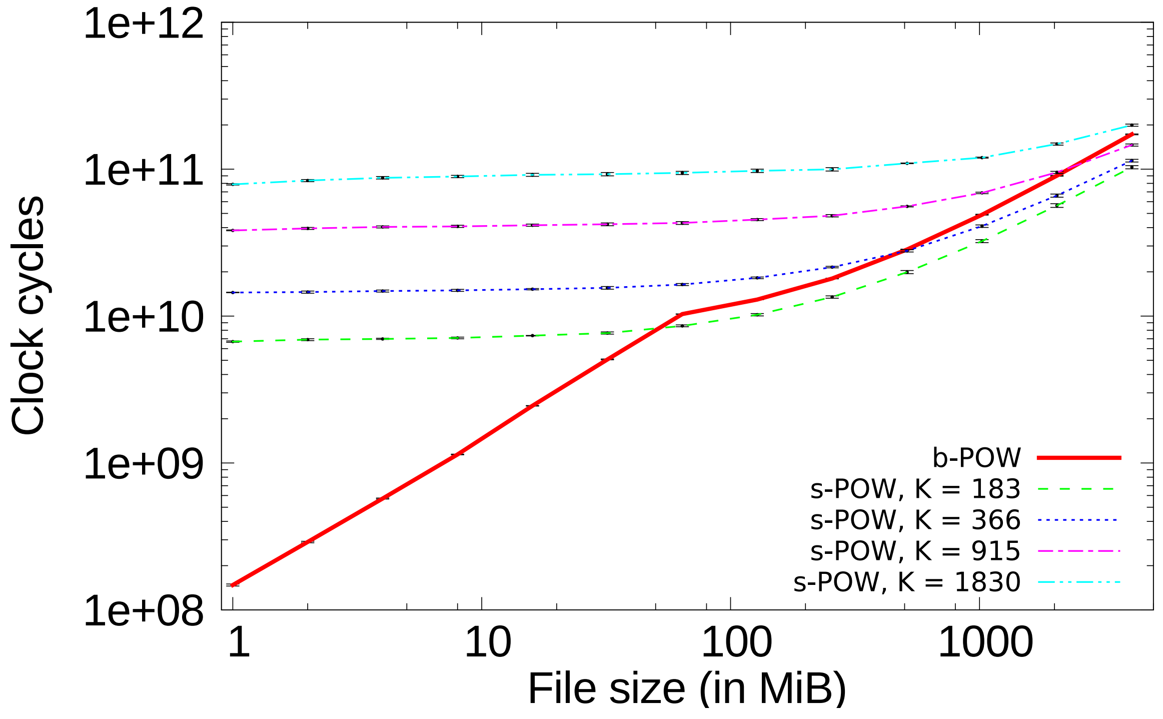

Figure 8: Comparison of the running time of the server initialization phase of b-POW with that of s-POW on Solid State Drives, for different values of K, as the input file size grows.

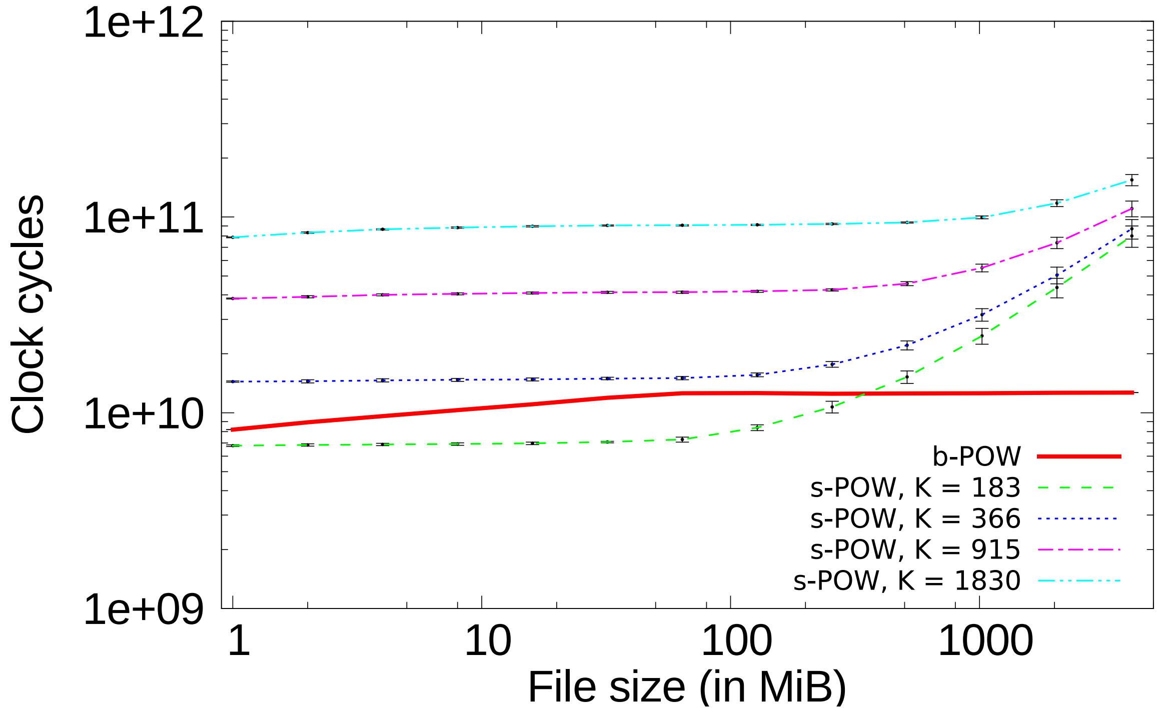

Figure 9: Comparison of the running time of the server regular execution phase of b-POW with that of s-POW on Solid State Drives, for different values of K, as the input file size grows.

6 Comparison and Discussion

In light of the analysis performed in the previous section, we now compare the state of the art solution and our proposals. Table 1 compares b-POW, s-POW and s-POW2 in terms of computational cost, I/O, storage and bandwidth requirements; we omitted s-POW3 from the comparison as it has the same asymptotic costs as s-POW2.

On the client-side s-POW and s-POW2 are far less demanding than b-POW from both the computational and the I/O perspective. This is a highly desirable characteristic, as the end user will receive better service. Such lower demands on the client-side are compensated by an increase of the footprint on the server-side. However, the design of the scheme allows the server to distribute computation and I/O over time, and to carry them out in moments of low system load.

From a computational perspective on the client-side, both b-POW and s-POW are dominated by the cost of calculating the hash of the file, whereas s-POW2 has a constant cost that only depends on the security parameter k, and is independent of the input file size. Similar considerations can be made when investigating the client-side I/O requirements.

On the server-side, we have made separate considerations for the initialization and for the regular execution phase. In the initialization phase, b-POW and s-POW are once more dominated by the computational and the I/O cost of the hash calculation, whereas s-POW2 only requires the precomputation of n challenges. The regular execution phase is particularly cheap for b-POW as no I/O and only constant computation are required. s-POW and s-POW2 regularly require replenishment of the stock of precomputed challenges. However, this operation can be performed offline in moments of low system load. Furthermore, files are often read at the server-side as part of standard management tasks anyway (e.g. periodic integrity checks, backup, replication); in this case, the I/O cost of the response pre-computation phase, which is by far the predominant cost, can be factored out.

As for server-side storage, b-POW requires only the root of the Merkle tree to be stored, whereas s-POW and s-POW2 require storing the pre-computed challenges. However, we emphasize that the number of responses to be pre-computed is a tunable parameter. Furthermore, the size of the responses is independent of the input file size – usually only a negligible fraction of the latter. For instance, storing 1000 responses, each 1830 bits long, would require less than 230 KiB.

Finally, the b-POW scheme requires the exchange of a super-logarithmic number of sibling paths of the Merkle tree between client and server, whereas s-POW and s-POW2 only require the exchange of a k-bit string.

6.1 Optimal Choices for Practical Deployments

Based on the analysis of each version of our scheme, we now suggest how a practical system could combine the different variants to obtain the best performance. Let us start with a few considerations: i) the cost of computing a standard cryptographic hash (e.g. SHA-1) for small files is negligible; ii) in view of the file size distribution in common datasets, and of the distribution of files with the same size, file size is a good indexing function for file sizes of 1 MiB and larger; iii) often the hash digest of a file is already available at the client-side; for instance, several peer-to-peer file sharing clients compute and/or store the hash of downloaded files; iv) as previously stated, in most modern filesystems, the I/O cost of reading a single bit at a random position is roughly equal to that of reading a larger chunk (either a disk sector or a filesystem block).

Table 1: Performance analysis of the discussed schemes; m represents the input file size, k is the security parameter. As explained in Section 4.1.1, the more precise formulation for $O(k)$ in our scheme is shown in Equation 2.

	b-POW	**s-POW**	**s-POW1**	**s-POW2**	**s-POW3**
Client-side computation	$O(m)$ hash	$O(m)$ hash	$O(m)$ hash	$O(k)$ PRNG	$O(k)$ PRNG
Client-side I/O	$O(m)$	$O(m)$	$O(m)$	$O(k)$	$O(k)$
Server-side computation (init.)	$O(m)$ hash	$O(m)$ hash	$O(m)$ hash	$O(nk)$ PRNG	$O(nk)$ PRNG
Server-side computation	$O(1)$	$O(nk)$ PRNG	$O(nk)$ PRNG	$O(nk)$ PRNG	$O(nk)$ PRNG
Server-side I/O (init.)	$O(m)$	$O(m)$	$O(m)$	$O(nk)$	$O(nk)$
Server-side I/O	0	$O(nk)$	$O(nk)$	$O(nk)$	$O(nk)$
Server-side storage	$O(1)$	$O(nk)$	$O(nk)$	$O(nk)$	$O(nk)$
Bandwidth	$O(k \log k)$	$O(k)$	$O(k)$	$O(k)$	$O(k)$

A practical system deployment would take into account these consideration as follows. First, all challenges would be performed at a block- or sector-level to obtain the improvements in terms of I/O highlighted in Section 4.2.1. Furthermore, the server would always accept a digest from the client if available, and use it as the lookup key (this requires the server to compute the digest of new files, but this is very likely performed for integrity protection anyway). Otherwise, two thresholds are set: for files smaller than t_1, the client has to compute the hash digest; for files in the t_1 to t_2 range, the approach of s-POW2 is used; for very large files (size greater than $t2$), the file size (and thus the approach of s-POW3) is used.

7 Conclusions

We have presented a suite of novel security protocols to implement proof of ownership in a deduplication scenario. Our core scheme is provably secure and achieves better performance than the state-of-the-art solution in the most sensitive areas of client-side I/O and computation. Furthermore, it is resilient to a malicious client leaking large portions of the input file to third parties, whereas other schemes described in the literature will be compromised in case of leaks that are larger than a pre-defined amount (64 MiB). On the downside, server-side I/O and computation are slightly higher than for state of the art solutions, but they can be conveniently mitigated by deferring them to moments of low system load. Note that the proposed solutions are fully customizable in the system parameters. Finally, extensive simulation results support the quality and viability of our proposal.

Acknowledgments

The authors would like to thank Geoff Kuenning and John Douceur for their help in accessing statistics on file sizes; Alexandra Shulman-Peleg and Christian Cachin for their insightful feedback; Charlotte Bolliger for her corrections.

Roberto Di Pietro has been partially supported by a Chair of Excellence granted by University Carlos III, Madrid.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST*, pages 31–45, 2007.

[2] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli*

Experimental Systems Conference, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009. ACM.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song. Remote data checking using provable data possession. *ACM Trans. Inf. Syst. Secur.*, 14(1):12:1–12:34, June 2011.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 598–609, New York, NY, USA, 2007. ACM.

[5] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, SecureComm '08, 2008.

[6] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. Cryptology ePrint Archive, Report 2012/631, 2012. `http://eprint.iacr.org/`.

[7] Bitcasa Infinite Storage. `https://www.bitcasa.com/`, 2013.

[8] Ciphertite High Security Online Backup. `https://www.cyphertite.com/`, 2013.

[9] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *ASIACCS*, pages 81–82, 2012.

[10] 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 *ICDCS*, pages 617–624, 2002.

[11] M. Dutch and L. Freeman. Understanding data de-duplication ratios. SNIA forum, 2008. `http://www.snia.org/sites/default/files/Understanding_Data_Deduplication_Ratios-20080718.pdf`.

[12] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, 2009.

[13] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *ACM Conference on Computer and Communications Security*, pages 491–500, 2011.

[14] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik. Estimation of deduplication ratios in large data sets. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1 –11, april 2012.

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

[16] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[17] R. Lewand. *Cryptological mathematics.* Classroom resource materials. Mathematical Association of America, 2000.

[18] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, pages 12–17, New York, NY, USA, 2008. ACM.

[19] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 1998.

[20] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[21] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[22] W. K. Ng, Y. Wen, and H. Zhu. Private data deduplication protocols in cloud storage. In *SAC*, pages 441–446, 2012.

[23] M. Saito and M. Matsumoto. Simd-oriented fast mersenne twister: A 128-bit pseudo-random number generator. In *In Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer-Verlag, 2007.

[24] M. W. Storer, K. M. Greenan, D. D. E. Long, and E. L. Miller. Secure data deduplication. In *StorageSS*, pages 1–10, 2008.

[25] W. van der Laan. dropship - dropbox api utilities. `https://github.com/driverdan/dropship`, 2011.

[26] J. Xu, E.-C. Chang, and 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*, ASIA CCS '13, pages 195–206, New York, NY, USA, 2013. ACM.

[27] Q. Zheng and 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, pages 1–12, New York, NY, USA, 2012. ACM.