A Tunable Proof of Ownership Scheme for Deduplication Using Bloom Filters

Jorge Blasco
Carlos III University of Madrid, Spain
jbalis@inf.uc3m.es

Agustin Orfila
Carlos III University of Madrid, Spain
adiaz@inf.uc3m.es

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

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

June 18, 2014

Abstract

Deduplication is a widely used technique in storage services, since it affords a very efficient usage of resources—being especially effective for consumer-grade storage services (e.g. Dropbox). Deduplication has been shown to suffer from several security weaknesses, the most severe ones enabling a malicious user to obtain possession of a file it is not entitled to. Standard solutions to this problem require users to prove possession of data prior to its upload. Unfortunately, the schemes proposed in the literature are very taxing on either the server or the client side. In this paper, we introduce a novel solution based on Bloom filters that provides a flexible, scalable, and provably secure solution to the weaknesses of deduplication, and that overcomes the deficiencies of existing approaches. We provide a formal description of the scheme, a thorough security analysis, and compare our solution against multiple existing ones, both analytically and by means of extensive benchmarking. Our results confirm the quality and viability of our approach.

1 Introduction

Cloud storage offers highly-available, virtually infinite and quick-to-scale storage; with its flexible pay-as-you go model, in recent years it has attracted new customers by the score. Coupled with dropping prices, the cloud paradigm has turned storage into a commodity [16]. The decreasing cost of storage media, the use of multi-tenancy, competition between cloud providers and the efficient use of the storage backend through compression and deduplication can be listed amongst the reasons for low-price high-quality cloud services, such as cloud storage services.

One of the techniques used to reduce the cost of cloud storage services is deduplication, which is currently implemented by providers such as Bitcasa [5] and Ciphertite [7]. In essence, deduplication avoids storing multiple copies of the same data. As an example, multiple copies of popular content (e.g. a song, a movie, a game) need to be stored only once upon the first upload; subsequent upload requests can be discarded and only require establishing a link from the uploading user to the original copy of the content. Deduplication can be performed very effectively at both file or block level: deduplication ratios vary from 2:1 to 50:1 for the same application by the same vendor depending on the setup and the input dataset [12].

Deduplication can take place at the client side (i.e. before the upload) or at the server side (i.e. after the upload). If deduplication is triggered at the client side it is more efficient, as it saves upload bandwidth. This is specially beneficial for service providers, due to the fact that network activity is the most energy-consuming task for cloud storage providers [3]. To avoid the transmission of the entire file content, but still allowing to check for its existence at the server side, clients are usually asked to generate a much shorter version of the file (a digest) and to use that digest to uniquely identify the file. The standard approach is to interpret the upload of a digest by a client as a proof that the client actually owns that file.

In their seminal work, Harnik *et al.* [13] first spotted the security weaknesses hidden behind this strategy. First, the privacy and confidentiality of users of a storage system can be compromised by an attacker that checks if another user has already uploaded a file by trying to upload it as well. If the upload does not take place, it means the server already stores it. This can be extremely dangerous if the file is very rare or private (e.g. a payslip). Second, deduplication can be abused to turn the service provider into a covert channel. Two colluding users with no direct connectivity can establish a protocol to exchange information stealthily. For instance, to exchange one bit of information, one of the users checks if a previously-agreed file has been uploaded or not during a certain time window. If the file was uploaded, the user can consider that a 1 has been transmitted (and 0 otherwise). Finally, a cloud storage service can be used as a content-distribution network (CDN). In such a case, a user can share large files with other users just by exchanging the corresponding digests. A real-world example of this attack was Dropship [8].

The root-cause of these threats lies in the fact that proof of ownership solely relies on the knowledge of a static, short piece of information (the digest). In order to mitigate this problem, the concept of "Proof of Ownership" was introduced by Halevi *et al.* [11]. A PoW scheme is a security protocol used by a server to verify that a client owns a file. A PoW scheme is considered secure if the probability of being able to fraudulently prove the ownership of a file is negligible in the security parameter, even if the attacker possesses a relevant part of the file [9]. Additionally, other important features of PoW schemes are their computational efficiency, in terms of bandwidth and I/O, for both legitimate clients and the server. Also, 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.

Contributions. In this paper we present a novel PoW scheme based on Bloom filters that competes with the state of the art proposals in terms of security guarantees and performance. A Bloom filter is a space-efficient randomized data structure used to check the membership of elements in a set. Bloom filters offer a trade-off between the size of the data-structure used to represent the set and the number of false positives that can occur when testing membership. Bloom filters have been successfully used in other domains of computer security [10]. However, to the best of our knowledge, they have never been applied to proof of ownership schemes. Our approach is more efficient at the client side than the solution proposed by Halevi *et al.* [11] and more efficient at the server side than the proposal by Di Pietro *et al.* [9]. The proposed solution is detailed and a thorough analysis of its security and efficiency is provided. Finally, extensive benchmarking supports the quality and viability of our proposal.

Roadmap. The rest of this document is structured as follows. Section 2 examines the related work. Section 3 introduces the requirements and model of an efficient scheme for proof of ownership, details our solution and formally analyses its security. Section 4 presents the results of our comprehensive benchmarks, where the performance of each different phase of our scheme is compared to that of the two competing solutions in the state of the art; finally, section 5 contains our concluding remarks.

2 Related Work

The seminal work of Harnik *et al.* [13] showed the shortcomings of client-side deduplication, and presented basic solutions to address privacy and confidentiality threats, such as triggering deduplication only when the same file has been uploaded a small, but random, number of times. Not knowing the exact threshold, the attacker cannot be certain whether the file had been previously uploaded. Later, Halevi *et al.* introduced the concept of Proof of Ownership [11] as a security control to effectively counter the shortcomings of deduplication. The authors presented three distinct PoW schemes based on Merkle trees [17] built on the content of the original file. All three schemes are based on the server challenging the client to provide valid sibling paths for a given random subset of tree leaves. Both server and client build the Merkle tree but the server only keeps the root node. Any time a client wants to prove the ownership of a file, the server asks for a super logarithmic random number of leaves of the Merkle tree. The server uses the received leaves and sibling paths to compute the root of the tree, and if it matches with the stored one, the proof of ownership is considered successful. Halevi *et al.* proposed three different ways to build the Merkle tree. The first one builds it from an erasure coding of the original file in order to "spread" unknown blocks of the file over a high number of blocks of the corresponding erasure coded version. Unfortunately, erasure coding is not I/O efficient and the input to the Merkle tree construction phase is a buffer whose size is greater than the file itself. The second scheme uses a hash function instead of an erasure coding, to the same end. They use a reduction buffer of 64 MiB to discourage sharing it among colluding users. Although this scheme is more efficient than the previous one it also incurs a high computational cost associated with hashing. In the third scheme, the server generates a sparse linear file when it receives a file for the first time by performing a set of reduction and mixing phases over the file contents. It is also assumed that the requirement of sharing 64MB among colluders would discourage them. For this third PoW they measured the client and server execution times, the network times and the time savings introduced by deduplication.

Di Pietro *et al.* [9] proposed a PoW scheme based on a challenge/response mechanism. Every time a file is uploaded to the server, the latter computes a set of challenges for that file and stores them. Each challenge is a PRF seed; the challenge seed is expanded to a number of bit positions in the file, and the response is the concatenation string of the bit values at the requested positions. Clients prove knowledge of a file by returning the correct string. This solution improves the one presented by Halevi *et al.* in terms of efficiency and bandwidth consumption at the client side, while requiring more computation at the server side (challenges have to be recomputed when they are exhausted). A similar proposal, but considering full file blocks, is proposed in a US patent [14]. Every time a user wants to prove ownership of a file, the server requests the content of a specific block of the file. In order to prove the ownership of the file, the client must send the whole block to the server. If they match, the server considers that the client owns the file. After each request by a client, the server refreshes the indexes to ask for. Other works present slight variations of the three schemes described so far [18, 22, 20, 21].

A related line of work takes into account the usage of end-to-end encryption to protect users' data. To allow deduplication under these restrictions, convergent encryption can be used [19]. The rationale behind this idea is to use information that can only be extracted from the plaintext file (available to all file owners) to encrypt the file prior to uploading it to the server. Thus, all file owners can derive the key to decrypt the file and the server only needs to store one encrypted version of the file. Although convergent encryption can be extremely useful to perform deduplication on popular files, it does not provide semantic security as it is vulnerable to content guessing attacks [4]. In this paper we focus on PoW and we do not consider the usage of any kind of encryption that eliminates the possibility of performing deduplication. Related interesting problems in cloud storage are Provable Data Possession (PDP) [1, 2] and Proof of Retrievability (PoR) [15] that deal with the dual problem of ensuring –at the client-side– that the server still stores the files.

3 Bloom filter Proof of Ownership Design

In this section we describe in detail our proposal. We start by describing the main building block, Bloom filters. We then formally introduce our scheme and constitutive algorithms.

3.1 Preliminaries

Bloom filters (BF) [6] are probabilistic data-structures used to represent sets of elements and to perform membership queries over them. The main advantage of Bloom filters is their memory and time efficiency, since they require less space than other data structures to store elements in the set, and less time to perform membership queries. The additional efficiency is traded for accuracy, since an element that is not in the set may be recognized as being part of it (false positives). False negatives, on the other hand, cannot occur by construction.

When a BF of size s is initialized, a vector of s bits is allocated, with its elements set to 0 ($InitBF(s) : BF \leftarrow \{b_0, b_1, \dots, b_{s-1}\}$). The BF requires k independent hash functions $\{h_0, h_1, \dots, h_{k-1}\}$ that are able to generate uniform random distributions of values between 0 and $s-1$. To insert an element e into the BF, the element is hashed with all k hash functions, and the bits at the positions corresponding to the output of the hash functions are set ($AddToBF(e, BF) : BF[h_i(e)] \leftarrow 1, i \in [0, k-1]$). Checking if the bloom filter contains a particular element requires it to be hashed with all k hash functions: if any of the bit positions corresponding to the output of the hash function is equal to 0 in the filter, the element is definitely not in the set. Conversely, either the element belongs to the set or a false positive occurred. That is, $InBF(e, BF) : \perp \Leftrightarrow \exists i \in [0, k-1] : BF[h_i(e)] = 0$.

The parameters of a Bloom filter are the total number N of elements to be inserted in the filter and the desired probability p_f that a false positive occurs; that is, the probability that $InBF(e, BF)$ returns $\top$ but no prior call to $AddToBF(e)$ was issued for the given element e. Once these parameters are set, the size s of the filter can be determined as

$$s = \left\lceil -\frac{N \ln p_f}{(\ln 2)^2} \right\rceil \tag{1}$$

The optimal number of hash functions can then be determined as $k = \left\lceil \frac{s}{N} \ln 2 \right\rceil$.

3.2 System and Objectives

Our scheme describes the interactions between clients and a server $\mathcal{S}$. Each client $\mathcal{C}$ has a unique identifier $id(\mathcal{C})$. Each client $\mathcal{C}$ uses $\mathcal{S}$ as a storage provider, uploading an arbitrary number of files. For any given file f, the scheme behaves differently for the *first* and for *subsequent* uploads. The server is able to distinguish first and subsequent uploads by means of a file digest h_f uploaded by the user: in particular, if the server receives a digest for the first time, the upload is treated as initial; if the server has already received the digest over a previous iteration from the same or a different client, the upload is treated as subsequent. For the first upload of file f, the server: 1. requires the client to upload its content; and, 2. initializes a set of data structures, including a Bloom filter, to be used during subsequent uploads of the same file. When a file is uploaded again, the server challenges the client to prove knowledge of the file, and verifies the responses provided by the client against the pre-computed information. We say that a client's PoW is successful if the client initiates a subsequent upload for a file f and responds correctly to the challenges sent by the server. After a successful PoW, the client legitimately owns the file, and can, later on, request its download.

The objective of a malicious client is to engage in a successful PoW run for a file he does not own. A malicious client may collude with the legitimate owner of the file, but we assume that the exchange of information does not take place interactively during the PoW challenge: that is, we do 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. The objectives of the scheme can be summarized as follows:

- security: a malicious client $\tilde{\mathcal{C}}$ that does not own a file f in its entirety engages in a successful PoW with negligible probability, given a security parameter κ;
- collusion resistance: a malicious client $\tilde{\mathcal{C}}$ who does not posses file f must exchange a minimum amount S_{min} of information with a legitimate owner of f in order to be able to engage in a successful PoW for f;
- bandwidth efficiency: the number of bytes exchanged between client and server upon a PoW run should be minimized;
- space efficiency: upon a PoW run, the server should only be required to load a small piece of information, whose size is independent of the input file size;

The first two objectives address the security requirements of the scheme: a malicious entity succeeds in a PoW for file f with negligible probability or by exchanging ahead of time at least S_{min} bytes with a colluding owner of f. The latter constraint is inspired by the work of Halevi *et al.* [11] and acts as a negative incentive for a legitimate owner $\mathcal{C}$ to collude with a malicious client $\tilde{\mathcal{C}}$, forcing the former to issue a large enough network transfer to the latter. The last two objectives set performance requirements for the scheme, attempting to minimize network bandwidth and memory consumption.

3.3 Our Scheme: bf-PoW

Every time a client $\mathcal{C}$ issues a request to $\mathcal{S}$ to store file f, the hash ($h_f \leftarrow \mathcal{H}_1(f)$) of that file is computed and sent to the server, where $\mathcal{H}_1 : \{0,1\}^* \rightarrow \{0,1\}^{n_1}$ is a cryptographic hash function and n_1 is a positive integer. The server keeps an associative array $\mathcal{A}$ that maps strings of finite size to 3-tuples; we use the dot notation to refer to components of tuples. The hash of a file h_f is the lookup key for $\mathcal{A}$: $\mathcal{A}[h_f].f$ contains the content of the file, $\mathcal{A}[h_f].BF$ contains the bits of a Bloom filter and $\mathcal{A}[h_f].AL$ contains a list of identifiers of clients that own f.

Informally, our scheme has two separate phases: in the initialization phase, the server receives a file for the first time, initializes a Bloom filter, splits the input file into *chunks* of equal size, creates *tokens* for each chunk and inserts a function of each token in the Bloom filter for that file. In the challenge phase, the server requests the client to upload a number of tokens to prove its knowledge of the file.

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, and play an important part in the security and performance of the scheme as we shall see later. The i-th chunk of file f is identified with $f[i]$. Let $PRF : \{0,1\}^l \times \{0,1\}^* \rightarrow \{0,1\}^{n_2}$ be a pseudorandom function,

Algorithm 1: Initialization phase.

```
Input: File $f$ uploaded by client $\mathcal{C}$
Output: The entry $\mathcal{A}[h_f]$
$BF \leftarrow InitBF(s)$;
for $i \leftarrow 0$ to $N-1$ do
    $t \leftarrow \mathcal{H}_2(f[i])$;
    $e \leftarrow PRF(t, i)$;
    $AddToBF(e, BF)$;
end
$\mathcal{A}[h_f].f \leftarrow f$;
$\mathcal{A}[h_f].BF \leftarrow BF$;
$\mathcal{A}[h_f].AL \leftarrow \{id(\mathcal{C})\}$;
return $\mathcal{A}[h_f]$;
```

for a positive integer n_2. Let us recall that the Bloom filter has two independent parameters: the probability p_f of false positives occurring and the number N of elements to be inserted: given the size F of a file f and the chunk size B, the number of elements in the filter can be determined as $N = \lceil \frac{F}{B} \rceil$. Finally, let κ be the security parameter and J be the number of tokens the client is required to produce in the challenge phase.

Algorithm 2: Challenge phase – client side.

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

The scheme operates as follows: the client computes and uploads the hash h_f for file f. If an entry for h_f is not found in $\mathcal{A}$, the server requests the client to upload the file and begins the bf-PoW *initialization phase* (Algorithm 1). Firstly, a Bloom filter is created and the file is split into chunks of equal size. Each chunk is used to generate the token t, which is in turn used to seed a PRF; the PRF is evaluated on the chunk index and the output is inserted into the Bloom filter. The Bloom filter, the file and the access list are inserted in the associative array.

If an entry for h_f is found in $\mathcal{A}$, the server generates an array *pos* of J randomly chosen chunk indexes and sends it to the client. The client then performs his side of the *challenge phase* (Algorithm 2): in particular, the client computes the token for each of the J chunk indexes and sends all tokens to the server. The server can then execute its side of the challenge phase (Algorithm 3): in particular, the server uses each token to seed the PRF, invokes the PRF on the corresponding chunk index and checks whether each output string belongs to the Bloom filter $\mathcal{A}[h_f].BF$ for f: if all do, the server considers the PoW run successful and assigns f to $\mathcal{C}$. If not, the client has failed the PoW. Note that a malicious client may exploit the occurrence of false positives in $InBF$ to engage in a successful PoW run without the exact knowledge of the content of the file; Section 3.5 will explain how the scheme handles such cases.

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

Algorithm 3: Challenge phase – server side.

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

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

Table 1: Complexity analysis of the bf-PoW against 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 PRF output size, p_f is the false positive rate of the BF.

$\mathcal{A}[h_f].f$ to $\mathcal{C}$ if the check is successful.

3.4 Complexity

In this section, we analyse bandwidth and space complexity of our scheme, along with its computational and I/O requirements. We compare our solution against the third and more optimized scheme presented by Halevi *et al.* [11] and the approach proposed by Di Pietro *et al.* [9]. We refer to b-PoW and s-PoW to refer to the two schemes, respectively. In all cases, we analyse the upper bounds only and consider all hash functions to have the same computational cost ($hash$). Results are summarized in Table 1. All PoW proposals require hashing the entire file f for identification purposes. Although Di Pietro *et al.* [9] reduce this complexity by using a function with similar properties but a smaller computational footprint, we choose to factor out this optimization as it can be applied to all three schemes alike.

In the case of bf-PoW, the client is required to execute J times $\mathcal{H}_2$ over chunks of size B ($O(B \cdot J) \cdot \mathcal{H}_2$). However, 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 overall computational cost is $O(F)$. At the server side, 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. Regarding memory consumption, initialization requires loading the entire file into memory. During regular execution, the server generates J pseudorandom bitstrings from the tokens (of length l) received by the client res. To check for membership in the Bloom filter, k hash functions are executed for each received token. The server is only required to load the Bloom filter into main memory, and not the entire file. In b-PoW, the server is only required to reconstruct the root of the Merkle-Tree with the received siblings. This requires transferring the root of the Merkle-tree into memory, which is a negligible cost. In s-PoW, the server is required to read the list of n precomputed challenges of size κ from disk.

In terms of bandwidth, bf-PoW requires J tokens to be sent to the server. This number increases roughly linearly as the security parameter κ increases, and decreases proportionally when the BF's false positive rate increases p_f. In the case of b-PoW, a super-logarithmic number of sibling paths of the Merkle tree is exchanged between the client and the server. Finally, s-PoW requires only K bits of the file chosen at random position to be sent to the server, where K is a superlinear function of the security parameter κ.

3.5 Security

In this section we will analyse the security of the proposed scheme. Firstly. let us model the adversary $\tilde{\mathcal{C}}$: given a file f, the objective of the adversary is to engage in a successful PoW, without actually possessing the file's content in its entirety. As stated above, the scheme does not protect against an adversary that uses a legitimate file owner as a real-time oracle supplying the correct responses to the PoW challenges. However, the adversary is free to interact with colluding clients prior to the PoW challenge. A simple way to model this interaction, taking into account that $\tilde{\mathcal{C}}$ may know parts of the file as well, is to bound the adversary's knowledge of the target file to a percentage p. This model also takes into account the knowledge of the file gained by $\tilde{\mathcal{C}}$ when possessing partial knowledge of the file content statistical distribution. That is, given a byte of the file at a randomly chosen position, the adversary knows it with probability p. We also assume that if the adversary does not posses a particular byte, it will be able to guess it with probability g. It is easy to prove that the best strategy for the adversary is to cluster its knowledge of the file into contiguous and aligned chunks of size B to obtain an optimal probability of success. We will then refer to p as the probability that the adversary knows in its entirety the content of a chunk of size B, whose position is chosen at random.

The PoW challenge requires the adversary to produce the tokens for J chunks at randomly chosen positions. Once received by the server, the token is processed with the PRF and the resulting bitstring is checked for membership in the Bloom filter. Let us focus on the generic i-th position out of the J random positions requested by the server and define the event in_i that $\tilde{\mathcal{C}}$ provides a token leading to a successful membership test in the Bloom filter. This happens in either of two cases: i) the adversary produces the token correctly (let us call this event tok_i); ii) a false positive occurs when checking membership in the Bloom filter. As discussed above, the latter happens with probability p_f. Then we can write

$$\begin{aligned} P(in_i) &= P(in_i \cap (tok_i \cup \overline{tok_i})) \\ &= P(in_i|tok_i)P(tok_i) + P(in_i|\overline{tok_i})P(\overline{tok_i}) \\ &= P(tok_i) + p_f P(\overline{tok_i}) \end{aligned} \tag{2}$$

Let us at first notice that the adversary cannot answer with the content of another chunk (unless the two are equal); this stems from the fact that the server will process the chunk index with a PRF, using the chunk's token as its seed. Let us now analyse the probability of the event tok_i

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, or does not; in the latter case, the adversary can either guess the B unknown bytes that compose the chunk (the probability of a correct guess being g^B 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 $g^B << 0.5^l$;[1] then we can write

$$\begin{aligned} P(tok_i) &= P(tok_i \cap (know_i \cup \overline{know_i})) \\ &= P(tok_i | know_i)P(know_i) + P(tok_i | \overline{know_i})P(\overline{know_i}) \\ &= p + (1-p)0.5^l \end{aligned} \tag{3}$$

Plugging Equation 3 into Equation 2, we derive

$$\begin{aligned} P(in_i) &= p + (1-p)0.5^l + p_f(1 - (p + (1-p)0.5^l)) \\ &= p + (1-p)0.5^l + p_f(1-p)(1-0.5^l) \\ &= p + (1-p)(0.5^l + p_f(1-0.5^l)) \end{aligned} \tag{4}$$

The adversary is challenged on J independent chunk positions. Therefore, we can compute the probability of success (let us call the event $succ$) of the adversary as

$$\begin{aligned} P(succ) &= P(in_i)^J \\ &= \Big(p + (1-p)(0.5^l + p_f(1-0.5^l))\Big)^J \end{aligned} \tag{5}$$

From Equation 5 we can derive a lowerbound for J that ensures $P(succ) \leq 2^{-\kappa}$, where κ is the security parameter, as

$$J \geq \frac{\kappa \ln 2}{(1-p)(1-(0.5^l + p_f(1-0.5^l)))} \tag{6}$$

Equation 6 ensures that the first security requirement highlighted in Section 3.2 is satisfied. 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 $\tilde{\mathcal{C}}$, in order for the latter to run a successful PoW for an unknown file. Given that tokens are shorter than the entire file chunk, the best strategy for the adversary is to request all tokens from the colluding client[2]. Given that there are $\frac{F}{B}$ tokens in a file f of size F, the token length l can be set as

$$l \geq S_{min}\frac{B}{F} \tag{7}$$

Note that Equation 7 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.

[1] A more complete security analysis, that we leave as an item of future work, would estimate g from the source entropy of the input file, for a large enough dataset, and would then be able to capture both strategies without assuming that one is always more advantageous than the other.

[2] 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.

4 Experimental Results

This section presents an experimental comparison of our scheme with other PoW schemes in the literature. We implement and benchmark our solution against the third and more optimized scheme presented by Halevi *et al.* [11] and the approach proposed by Di Pietro *et al.* [9]. We will refer to ours as bf-PoW, and will use b-PoW and s-PoW to refer to the other two schemes, respectively. The three schemes are implemented in C++; the OpenSSL crypto library is used as a provider for all crypto operations. In particular, $\mathcal{H}_1$ is implemented using SHA-1, whereas the variable-length hash function $\mathcal{H}_2$ its implemented by using SHA-1 to produce a key for RC4, and then encrypting as many zero-bytes as required to obtain l total bytes of ciphertext. Finally, PRF is implemented using a SHA-1-based HMAC. Network interactions between server and client are virtualised to eliminate benchmarking noise deriving from variations in network latency and bandwidth. The benchmarks are run on an Intel Xeon 2.27GHz CPU with 18 GiB of RAM running RHEL Server release Santiago (6.1). The input files contain random data, and their size ranges from 1 MiB to 4 GiB, with the size doubled at each step.

We choose the parameters for all schemes as follows: to be consistent with the choices in [11, 9], the security parameter κ is set to 66 and the threshold S_{min} is set to 64 MiB; finally, the probability p that the adversary knows a chunk of the file, is set to $\{0.5, 0.75, 0.9, 0.95\}$. Concerning the specific parameters of bf-PoW, the token size l is set to $\{16, 64, 256, 1024\}$ bytes, whereas the false positive rate p_f of the Bloom filter is set to 0.1 (and the Bloom filter is consequently sized). Note that Equation 6 fully characterizes the security of the scheme: the scheme's security is not affected by such a high choice of false positive rate, and such a choice affords very small Bloom filters. Given these values, J is set to $\{102, 204, 509, 1017\}$ according to Equation 6; the chunk size B is set to satisfy Equation 7, given the values of l, S_{min}, and the input file size.

4.1 Client Side

Here we compare the client-side performance of bf-PoW with that of b-PoW and s-PoW. In particular, we focus on the challenge phase of the three schemes; that is, we focus on the uploads of a given file f occurring after the first upload. The three schemes alike require the client to load the file from disk and compute its hash (SHA-1 in all three cases). Then: for b-PoW, the reduction phase, the mixing phase, and the calculation of a binary Merkle tree on the resulting reduction buffer are executed; for s-PoW, the required number of random positions (given the chosen parameters) is generated out of the received seed, and the bits of the file at the corresponding positions are collected to form the response; finally, for bf-PoW, J tokens are generated using Algorithm 2.

Let us at first analyse bf-PoW. Figure 1 focuses on Algorithm 2 of bf-PoW and plots its running time when l is fixed to 1024 bytes and p varies, and when p is fixed to 0.95 and l varies. From the first subplot we can see that as p grows, the running time of the algorithm increases by a constant factor; from the second subplot we can see that as l grows, the running time increase is more evident: this is due to the fact that, as the input file size grows, a larger token size implies a larger chunk, whose hashing time increases. In both plots, the flat zone before 64 MiB and the subsequent tilting is justified by the choice of S_{min}. Figure 2 shows the overall client-side performance of the three schemes, including the time required to load the file and compute the initial hash. We can see that file I/O and hash computation are by far the dominant factors in this phase and that, asymptotically, all three scheme behave similarly. b-PoW is the slowest of the three in this phase; the performance of bf-PoW is midway between

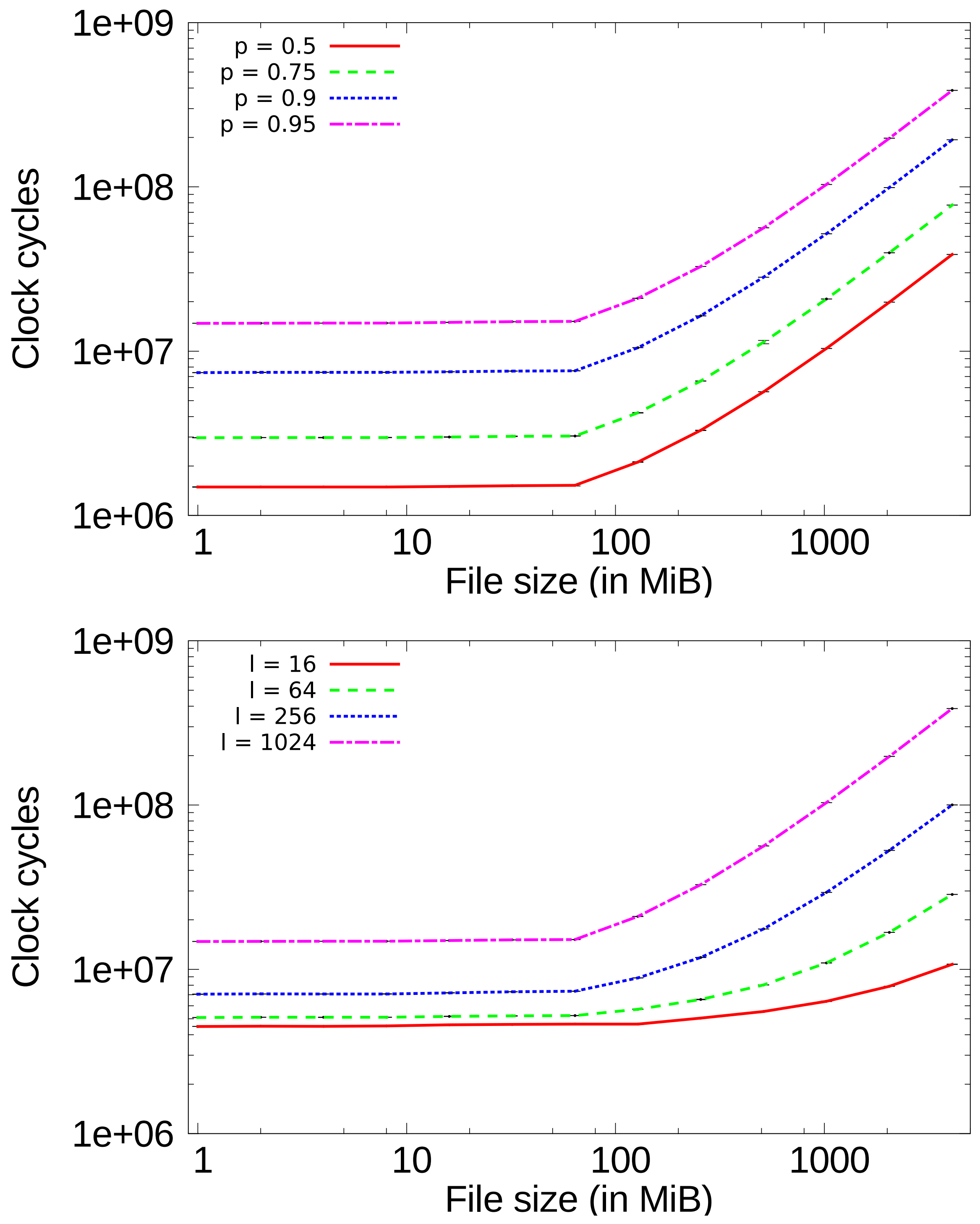


Figure 1: Execution time of Algorithm 2 as the input file size grows. The upper plot uses a constant $l = 1024$ bytes and different values of p; the lower plot uses constant $p = 0.95$ and different values of l.

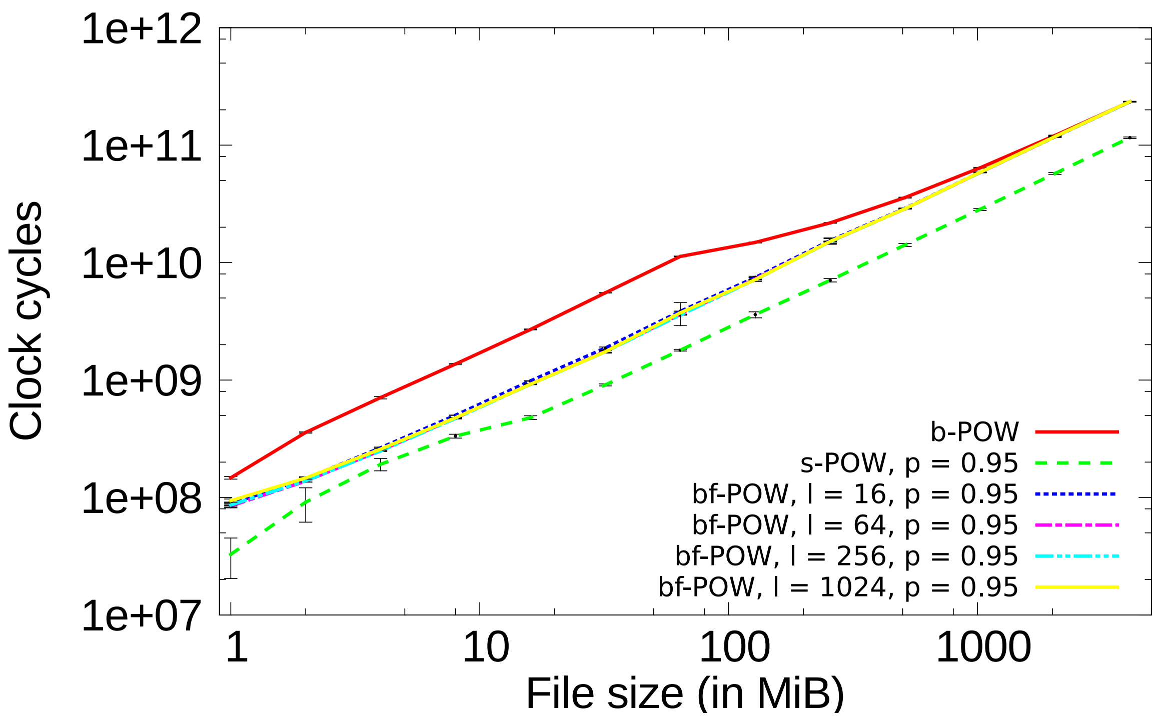


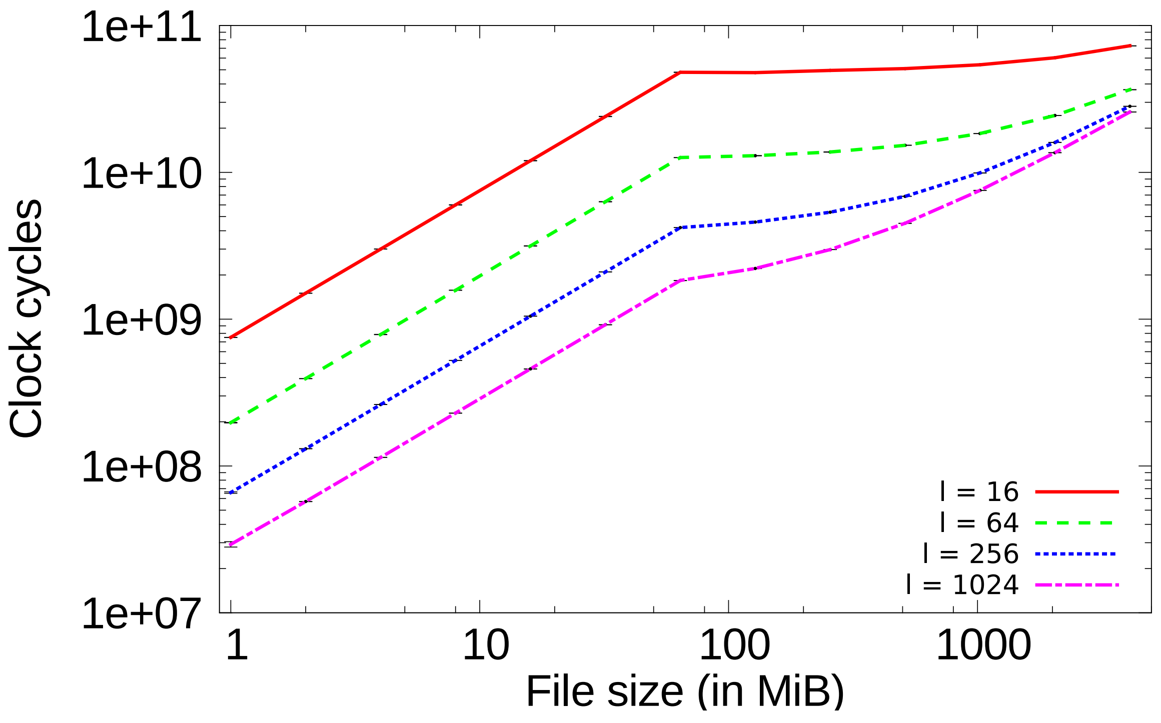
Figure 2: Execution time at the client side between different configurations of bf-PoW against b-PoW and s-PoW, as the input file size grows.

s-PoW and b-PoW, closer to the former for smaller files and to the latter for larger ones. We can also notice that different values of l (or of p) have little influence on the performance of the scheme.

4.2 Server Side

According to the benchmarking framework proposed by Di Pietro *et al.* [9], we split the server side into initialization and regular execution; the former covers the operations that take place upon the first upload of a file, whereas the latter includes the steps executed by the server for (w.l.o.g.) 10,000 subsequent uploads of the file. In all three schemes, the initialization phase starts with the computation of the digest of the file. Then, for bf-PoW, we add the time required to initialize the Bloom filter (Algorithm 1); for b-PoW, the same steps as in the client side are factored in; finally, for s-PoW, 10,000 PoW response strings are pre-computed. For the regular execution phase, each iteration of bf-PoW includes the time required by the server to verify J tokens; b-PoW requires verification of the correctness of the sibling path in the computed Merkle tree for a super-logarithmic number of leaves; and finally, much like the initialization phase, s-PoW requires the pre-computation of 10,000 PoW responses to as many challenges, and the equality test for the PoW response string (whose running time is negligible).

The first subplot of Figure 3 analyses the performance of Algorithm 1 for different token sizes (varying p has no impact on this algorithm): in this case, contrary to what we noticed for Algorithm 2, larger tokens imply a smaller total number of tokens and consequently, a faster execution time. The second subplot compares the execution time of all three schemes. s-PoW is the slowest; bf-PoW is faster than b-PoW for large tokens and slower otherwise, for the reasons explained above.



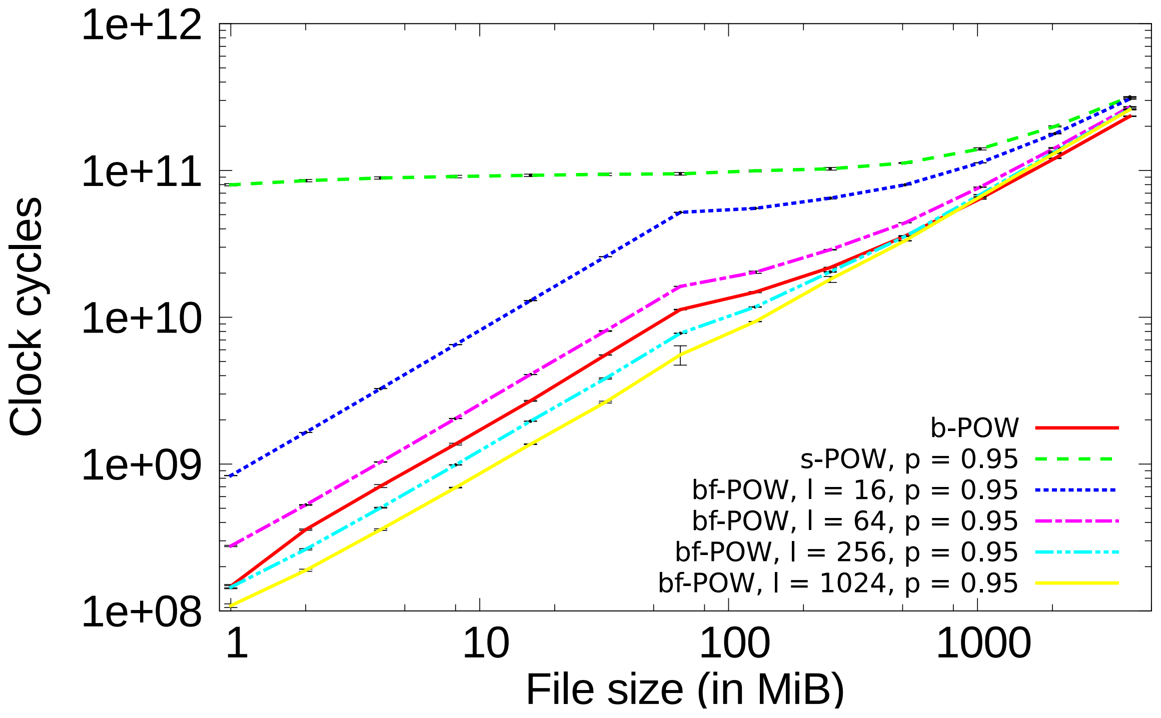


Figure 3: (Upper plot): execution time of Algorithm 1 for different values of l as the input file size grows. (Lower plot): comparison of the execution time of different configurations of bf-PoW during the server initialization phase with b-PoW and s-PoW, as the input file size grows.

Let us now turn to the regular execution phase (for 10,000 executions). The first subplot of Figure 4 analyses the running time required to verify J tokens (Algorithm 3); we can see that the execution time is constant for a fixed value of l and differences in p only affect the number J of tokens per challenge. Also, similarly to the client-side, small tokens offer better performance, as opposed to the initialization phase where the reverse was true. The second subplot compares the execution time of all three schemes. At first, we can notice that the execution time of both bf-PoW and b-PoW is constant as the input file size grows, whereas the performance of s-PoW degrades considerably for larger files. The running time of both bf-PoW and s-PoW depends on the chosen value of p: for very conservative values of p (e.g. 0.95), the probability of $\tilde{\mathcal{C}}$ knowing a random byte of the file, both schemes perform around 10 times slower than b-PoW; for smaller values of p, bf-PoW performs equally to (or even faster than) b-PoW.

4.3 Memory Requirements

From equations 1 and 7 we can see that the size (in bits) of the Bloom filter does not depend on the file size and can be determined as

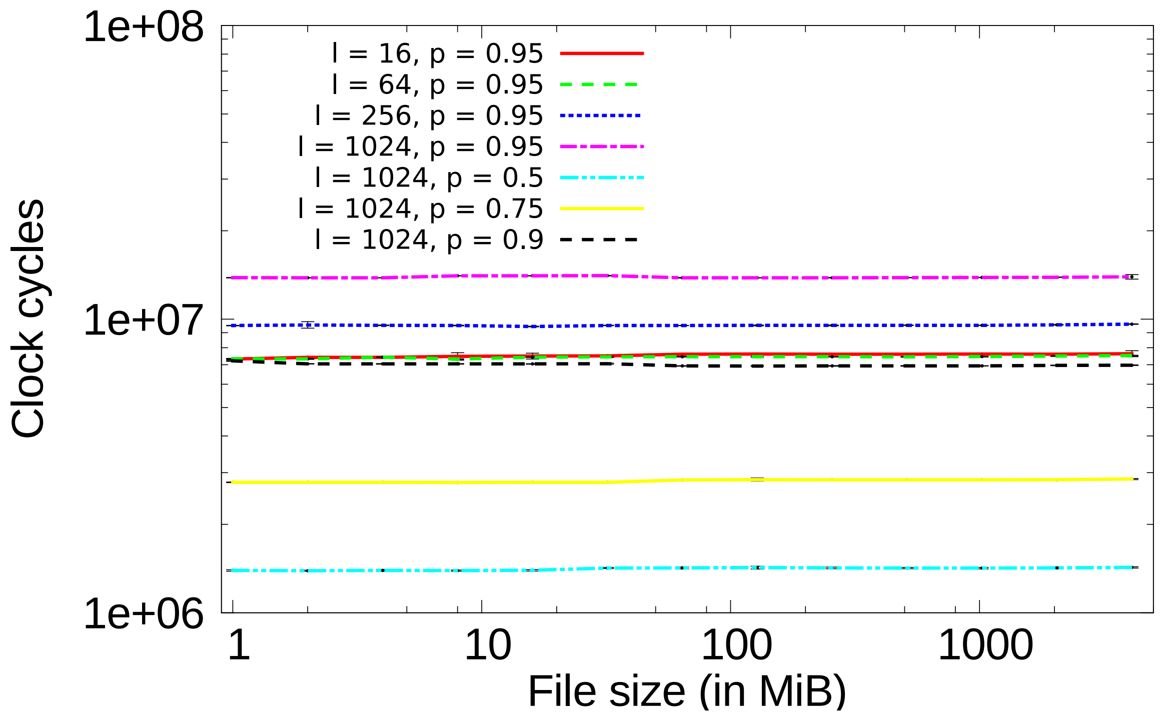
$$s = \left\lceil -\frac{S_{min} \ln p_f}{l\,(\ln 2)^2} \right\rceil \tag{8}$$

for given values of the threshold S_{min}, the token length l and the false positive rate p_f. From Equation 8 we can notice that the file size does not play a role in the length of the bloom filter, thus satisfying the last requirement listed in section 3.2. Figure 5 plots the size of the Bloom filter for a few different configurations of the token length l and of the false positive rate p_f. We can see how we can trade smaller filters for higher false positive rates (which our scheme can withstand as described in section 3.5), and how shorter tokens (which are favourable for clients since they afford both smaller response lengths and quicker execution times) unfortunately require larger filters. However, even in the worst case (for p_f equals to 0.1) we can see how the filter is never larger than 2 MiB (for $l = 128$ bits), and can be as small as 40 bytes for larger tokens.

The competing schemes behave as follows: b-PoW only requires the root of the Merkle tree to be stored (which is as large as our filter in the most favourable situation, as we can see in Figure 5); s-PoW cannot be compared fairly since the size of precomputed responses can be as short as a single one, but it then requires re-loading of the whole file when the need to recompute the next set of challenges arises.

4.4 Discussion

The results of our benchmarks show ample space for tradeoffs in the way bf-PoW can be configured. We have seen how the scheme sports better performance with small tokens in the client and server (regular execution) phases. Small tokens also entail smaller bandwidth requirements for the challenge phase: practical deployments are therefore expected to choose small tokens, which results in larger Bloom filters. As previously stated, in the worst configurations this would require up to 2MiB of additional storage per stored file. Due to storage savings thanks to deduplication, we consider this requirement acceptable. bf-PoW is the fastest scheme for certain choices of p at the server side, being always faster than s-PoW. At the client side, bf-PoW is faster than b-PoW and has performance comparable to that of s-PoW. Both bf-PoW and s-PoW have the advantage of a simple security analysis, based on standard assumption, whereas the security of b-PoW is (as admitted by the authors) based on assumptions that are hard to verify in practice. The latter should constitute an important additional point to consider.



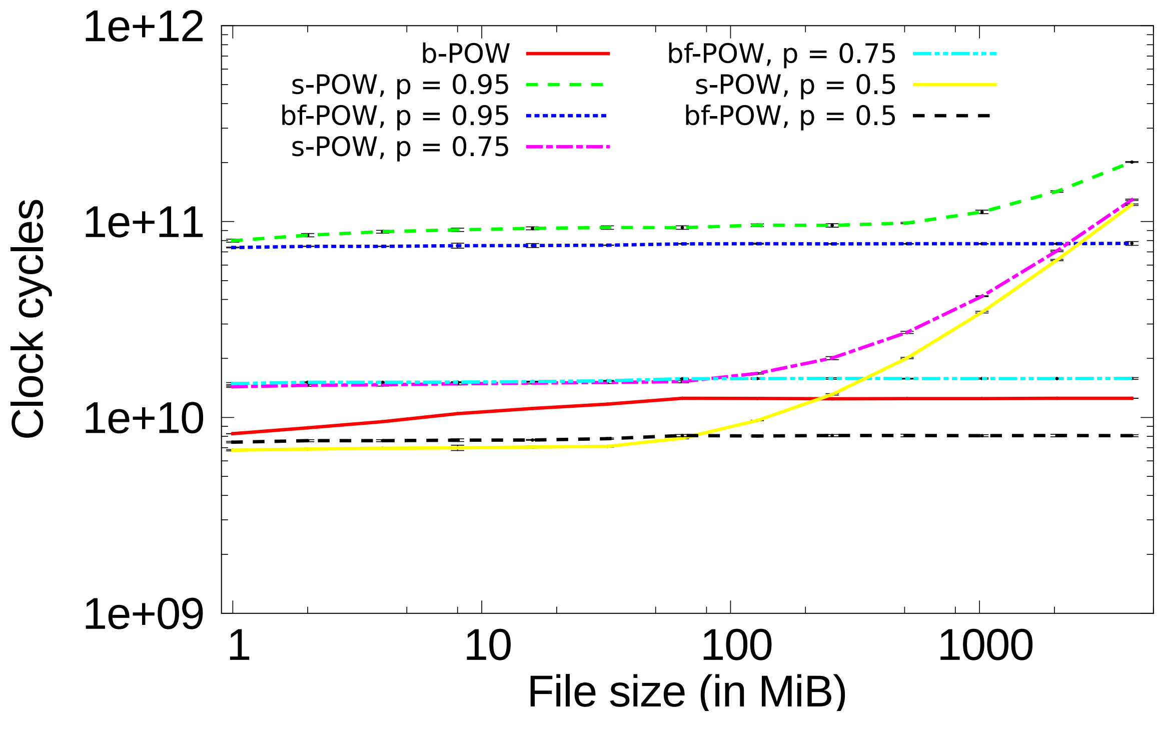


Figure 4: (Upper plot): execution time of Algorithm 3 for different values of l as the input file size grows. (Lower plot): comparison of the execution time of different configurations of bf-PoW during the server regular execution phase with b-PoW and s-PoW, as the input file size grows.

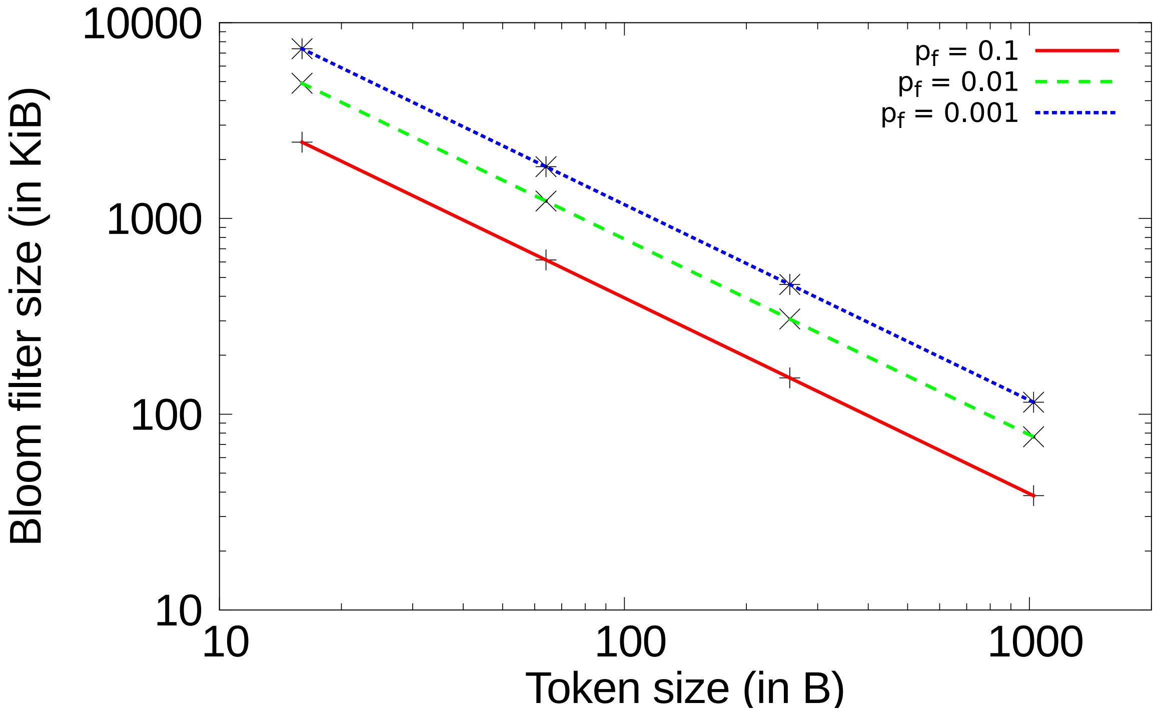


Figure 5: Size of the Bloom filter for different values of the token length l and the probability of a false positive p_f.

5 Conclusions

In this paper we have introduced a novel Proof of Ownership (PoW) scheme that thwarts known security threats to deduplication. In particular, we have provided a solution that, by leveraging Bloom filters, is an improvement over state of the art solutions. Our approach is more efficient at the client side than the solution proposed by Halevi *et al.* [11] and more efficient at the server side than the proposal by Di Pietro *et al.* [9]. Furthermore, our solution allows the systems and security parameters to be tailored to meet the expected security and efficiency requirements. Our proposal is an ideal alternative to state of the art solutions for a broad set of application scenarios. Finally, we have provided a detailed analysis of the security and efficiency of our proposal, and an extensive set of benchmarks that support the quality and viability of our findings.

References

[1] 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.

[2] 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, pages 9:1–9:10, New York, NY, USA, 2008. ACM.

[3] J. Baliga, R. W. Ayre, K. Hinton, and R. S. Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149–167, 2011.

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

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

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

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

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

[9] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 81–82. ACM, 2012.

[10] S. Geravand and M. Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, In Press(0):–, 2013.

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

[12] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik. Estimation of deduplication ratios in large data sets. In *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, pages 1–11, 2012.

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

[14] A. Juels. Method and system for preventing de-duplication side-channel attacks in cloud storage systems, Sept. 3 2013. US Patent 8,528,085.

[15] 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.

[16] X. Li, Y. Li, T. Liu, J. Qiu, and F. Wang. The method and tool of cost analysis for cloud computing. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, pages 93–100, 2009.

[17] R. C. Merkle. A certified digital signature. In *Advances in Cryptology—CRYPTO'89 Proceedings*, pages 218–238. Springer, 1990.

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

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

[20] 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.

[21] C. Yang, J. Ren, and J. Ma. Provable ownership of files in deduplication cloud storage. *Security and Communication Networks*, pages n/a–n/a, 2013.

[22] 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.