Accountable Storage

Giuseppe Ateniese[*] Michael T. Goodrich[†] Vassilios Lekakis[‡]
Charalampos Papamanthou[§] Evripidis Paraskevas[§] Roberto Tamassia[¶]

Abstract

We introduce *Accountable Storage* (AS), a framework allowing a client with small local space to outsource n file blocks to an untrusted server and be able (at any point in time after outsourcing) to provably compute how many bits have been discarded by the server. Such protocols offer "provable storage insurance" to a client: In case of a data loss, the client can be compensated with a dollar amount proportional to the damage that has occurred, forcing the server to be more "accountable" for his behavior. The insurance can be captured in the SLA between the client and the server.

Although applying existing techniques (e.g., proof-of-storage protocols) could address the AS problem, the related costs of such approaches are prohibitive. Instead, our protocols can provably compute the damage that has occurred through an efficient recovery process of the lost or corrupted file blocks, which requires only sublinear $O(\delta \log n)$ communication, computation and local space, where δ is the maximum number of corrupted file blocks that can be tolerated. Our technique is based on an extension of invertible Bloom filters, a data structure used to quickly compute the distance between two sets.

Finally, we show how our AS protocol can be integrated with Bitcoin, to support automatic compensations proportional to the number of corrupted bits at the server. We also build and evaluate our protocols showing that they perform well in practice.

1 Introduction

Cloud computing is revolutionizing the entire field of information technology, but it is also posing new security and privacy challenges. On the one hand, clients expect to gain reliability and availability from having their data stored remotely, but, on the other hand, remote storage forces their data to live outside their control and protection. Organizations are therefore reluctant to outsource their databases for fear of having their data lost or damaged, but they also do not want the burden of fully managing all their data themselves. Thus, they would benefit from technologies that would allow them to manage their risk of data loss, much in the same way that insurance allows them to manage their risk of physical or financial losses, e.g., from fire or liability.

As a first step, a client needs a mechanism for verifying that a cloud provider is storing her entire database intact, even the portions rarely accessed, and, fortunately, a series of *proofs-of-storage* protocols have been proposed to solve this problem. These techniques include *Provable Data Possession* (PDP) (e.g., [4, 11, 13]) and *Proofs of Retrievability* (POR) [10, 19, 28, 29, 30], both conceived as a solution to the integrity problem of remote databases. A PDP/POR scheme can verify whether an untrusted server possesses the entire database originally uploaded by the client. This is achieved by having the server generate a proof in response to a challenge. In addition, a client is typically required to store a constant amount of metadata to verify the server's proof.

Proof-of-storage schemes have witnessed remarkable improvements over the years and are now considered to be well established. Still, they leave unsettled several risk management issues. Arguably, the most important question of all is

What happens if a proof-of-storage scheme shows that a client's outsourced database has been damaged?

Clearly, the cloud is liable for this damage, and the client should be compensated for her loss. But to assess this damage and receive compensation, two significant hurdles must be overcome:

- The client may not be able to reliably and quickly assess the damage, because she no longer stores her database locally.
- The client has no technological way to automatically receive compensation and must instead follow cumbersome legal procedures, specified in a service level agreement (SLA), in order to be compensated for her loss.

[*]Department of Computer Science, John Hopkins University. Email: `ateniese@cs.jhu.edu`
[†]Department of Computer Science, University of California, Irvine. Email: `goodrich@uci.edu`
[‡]Department of Computer Science, University of Maryland College Park. Email: `lex@cs.umd.edu`
[§]Department of Electrical and Computer Eng., University of Maryland College Park. Email: `cpap,evripar@umd.edu`; Corresponding Author
[¶]Department of Computer Science, Brown University. Email: `rt@cs.brown.edu`

The objective of this work is to design new efficient protocols that address the above challenges. Our framework is intentionally modular in that we address both challenges independently and we merge the solutions only afterwards. In this way, it is possible to improve a solution to one challenge without significantly affecting the other.

1.1 Quantifying Data Loss

To be more precise, suppose a client, Alice, outsources her file blocks, $b_1, b_2, \ldots, b_n$, to a potentially malicious cloud storage provider, Bob. Since Alice does not trust Bob, she wishes, at any point in time, to be able to compute the amount of damage, if any, that her file blocks have undergone, by engaging in a simple challenge-response protocol with Bob.

For instance, she may wish to quickly and verifiably compute the value of a *damage metric*, such as

$$d = \sum_{i=1}^{n} ||b_i \oplus b'_i|| \,, \tag{1}$$

where b'_i is the file currently stored by Bob at the time of the challenge and $||.||$ denotes Hamming distance. If $d = 0$, then Alice is entitled to no dollar credit, as Bob stores Alice's data intact. Bob can easily prove to Alice that this is the case through existing proof-of-storage protocols, as noted above. If $d > 0$, however, then Alice should receive a compensation proportional to the damage d, which should be provided automatically (proof-of-storage protocols cannot help here).

Such a quantitative protocol (one that Alice could use to compute the damage d) would force the cloud provider, Bob, to be more "accountable", since it places a monetary liability on loss based on damage size. Moreover, such fine-grained compensation models, which work at the bit level as opposed to at the file block level, beneficially allow Alice to better manage her risk for damage even within the same file. For example, compensation for an unusable movie stored by Bob could be larger than that for an usable movie whose resolution has deteriorated by just 5%.

Note that it is imperative that a protocol for the computation of the damage d be *efficient*. Alice should not need to download the entire file collection and compare it to some local or remote backup to calculate the value of d. Moreover, computation of the damage d should also be *verifiable*, so that it is computationally infeasible for Bob to persuade Alice (or a third party) that the current damage is less than the actual damage.

1.2 Our Contributions

The contributions of this work are as follows:

(1) We introduce and formally define *Accountable Storage* (AS), a new framework where a client, Alice, outsources n file blocks to an untrusted server, Bob, so that later Alice can provably recover up to δ blocks that cannot be extracted by accessing Bob's storage (e.g., due to random failures or malicious behavior)—see Section 3. *Recovering the lost blocks directly enables the client to compute the damage d as defined in Relation 1.* Such a framework enforces accountability for storage servers, since the client can demand a compensation proportional to the damage d.

(2) We construct an efficient AS protocol—see Section 4. Our protocol runs in one round, requires Alice to store just $O(\delta \log n)$ local state, and has $O(\delta \log n)$ bandwidth, where δ is the maximum number of corrupted file blocks that can be tolerated.

(3) We provide an open-source prototype implementation of our AS construction and perform a thorough experimentation at Section 6. Our final system is practical: E.g., for a 4GB file system, we can process a small proof to provably locate and recover 6MB of lost data (that can be scattered in any location within the file) in about 100 seconds.

(4) We show how to integrate our AS protocol with Bitcoin [1] to enable an automatic compensation to Alice proportional to the damage, d. See Section 5.

1.3 Related Work and Potential Approaches

We are not aware of any existing work on Accountable Storage.[1] The work of Yumerefendi and Chase [34] provides "accountability" for network storage, but only in the sense of detection, not for assessing damage. There are also some ways to use the techniques described below to achieve properties similar to those of Accountable Storage, but these result in inefficient protocols.

Using PDP. A provable data possession (PDP) protocol [4, 5, 11, 13, 31] enables a server to prove to a client that all of the client's data is stored intact. One could design an AS protocol by using a PDP protocol only for the portion of storage

[1]We note here that our notion of accountability focuses on quantifying the amount of damage d that has occurred at a remote repository. However, in general, providing accountability can help a cloud provider identify specific reasons that caused the damage such as hardware failures or insider attacks. These are outside the scope of this work.

that the server possesses. This could determine the damage, d. The problem with such an approach, however, is that it requires use of PDP at the bit level, computing one tag for each bit of our file collection. PDP tags are usually 2048 bits [4]; therefore, it is impractical to keep that many extra bits for every bit of initial storage.

Using error-correcting codes. To overcome the above problem, one could use PDP at the block level, but at the same time keep some redundancy locally. Specifically, before outsourcing the n blocks at the server, the client could store δ extra check blocks locally (the check blocks could be computed with a simple error correcting code such as Reed-Solo-mon). The client could then verify through PDP that a set of at most δ blocks have gone missing and retrieve the lost blocks by executing the decoding algorithm on the remote intact $n-\delta$ data blocks and the δ local check blocks (then the recovered blocks can be used to compute d). However, this procedure would require $O(n)$ communication, since the $n-\delta$ intact blocks at the server must be sent to the client. A system that could operate along these lines, for example, is IRIS [30]. IRIS employs $O(\sqrt{n}\log n)$ client side redundancy, with which $\delta = \sqrt{n}$ file blocks can be recovered. Although IRIS is fully-dynamic (our construction supports only append-only updates), the *bandwidth* and *computation* for retrieving $\sqrt{n}$ corrupted blocks is *linear* in the number of the file blocks, as the whole file system needs to be streamed from the server to the client. To reduce communication to $O(\delta)$ the client could send his δ check blocks to the server. However, this does not work either, since the server would recover the file blocks and pretend that no damage ever occured.

1.4 Overview of Our Techniques

AS construction. Our protocol for assessing the damage d from Relation 1 is based on *recovering the actual blocks* $b_1, b_2, \ldots, b_\delta$ *and subsequently "XORing" them with the corrupted blocks* $b'_1, b'_2, \ldots, b'_\delta$ *returned by the server*. For recovery (described in Section 4), we use the invertible Bloom filter (IBF) data structure, introduced by Eppstein et al. [12] and Goodrich and Mitzenmacher [15]. An IBF is an array of t cells and can store a set of $O(t)$ elements. Unlike traditional Bloom filters [6], the elements of an IBF can be enumerated with high probability.

Let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of blocks outsourced by the client and let δ be the maximum number of corrupted blocks that the client can tolerate. In the preprocessing phase, the client computes an IBF $\mathbf{T}_B$ that has $O(\delta)$ cells, on the blocks $b_1, \ldots, b_n$. The client stores $\mathbf{T}_B$ locally. Computing $\mathbf{T}_B$ is similar to computing a plain Bloom filter: every cell of $\mathbf{T}_B$ is mapped to a sum over a set of at most n blocks, thus the local storage is $O(\delta \log n)$.

To outsource the blocks, we modify the IBF specification so that the client computes homomorphic tags, $\mathtt{T}_i$, for each file block, b_i. Incidentally, we use the same tags as in the original PDP scheme by Ateniese et al. [4]. The client then stores $(b_i, \mathtt{T}_i)$ with the cloud and deletes the blocks $b_1, b_2, \ldots, b_n$ from local storage. In the challenge phase (i.e., when the client wants to provably compute the damage d), the client asks the server to construct a *randomized* IBF $\mathbf{T}_K$ (again, of $O(\delta)$ cells) on the set of blocks K that the server currently possesses. The IBF $\mathbf{T}_K$ comprises the "proof" that the server sends to the client.

Note that since $\delta < n$, both $\mathbf{T}_B$ and $\mathbf{T}_K$ "lose" information. Therefore sets B and K cannot be enumerated only by processing $\mathbf{T}_B$ and $\mathbf{T}_K$. However, one property of the IBFs is the fact that if the size of the *difference* $B - K$ is at most δ, then the client can take the algebraic difference $\mathbf{T}_L = \mathbf{T}_B - \mathbf{T}_K$ and, with high probability, recover the elements of the difference $B - K$ by only working with $\mathbf{T}_L$ (note that $\mathbf{T}_L$ has $O(\delta)$ cells). Recovering the blocks in $B - K$ enables the client to compute damage d by using Relation 1.

Bitcoin protocol. We note here that in the above protocol, Bob (the server) is *malicious* and is trying to persuade Alice that the weighted damage of her file blocks is $d' < d$. However, the cryptographic tags that we use ensure that Bob cannot succeed in that (except with negligible probability) and therefore Alice will eventually find out the exact damage, d. After that stage, compensation proportional to d must be sent to Alice. But Bob could try to cheat again.

Specifically, Bob could try to give Alice a compensation less than required or, even worse, disappear. To deal with this problem, we develop a modified version of the recently-introduced timed commitment in Bitcoin [3]. At the beginning of the AS protocol, Bob deposits a large amount, A, of bitcoins, where A is contractually agreed on and is typically higher than the maximum possible damage to Alice's file blocks. The Bitcoin-integrated AS protocol of Section 5 ensures that unless Bob fully and timely compensates Alice for damage d, then A bitcoins are automatically and irrevocably transferred to Alice. At the same time, if Alice tries to cheat (e.g., by asking for compensation higher than the contracted amount), our protocol ensures that she gets no compensation at all while Bob gets back all A of his bitcoins.

2 Preliminaries

We let τ denote the security parameter, δ denote an upper bound on the number of corrupted blocks that can be tolerated by the client, n denote the number of file blocks, and $b_1, b_2, \ldots, b_n$ denote the file blocks. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$ and PPT stands for "probabilistic polynomial time".

```
Algorithm $\mathbf{T} \leftarrow \mathsf{update}(b_i, \mathbf{T}, z)$
for each $j = 1, \ldots, k$ do
  Set $ind = h_j(i)$;
  Set $\mathbf{T}[ind].\mathsf{count} = \mathbf{T}[ind].\mathsf{count} + z$;
  Set $\mathbf{T}[ind].\mathsf{dataSum} = \mathbf{T}[ind].\mathsf{dataSum} + zb_i$;
  Set $\mathbf{T}[ind].\mathsf{hashProd} = \mathbf{T}[ind].\mathsf{hashProd} \times f(b_i)^z$;
return $\mathbf{T}$;
```

Figure 1: Update in an IBF. The input $z \in \{-1, 1\}$ determines whether we insert b_i into or delete b_i from the IBF.

Blocks $b_1, b_2, \ldots, b_n$ do not contain only *data*. They also encode the respective index of the block as the first $\log n$ most significant bits, i.e., $b_i = i||\beta_i$ where i is the index and β_i is an m-bit data block. Such representations $i||\beta_i$ can be viewed as *integers* belonging to a universe $\mathcal{U}$. When we refer to a block b_i, we imply that one can retrieve its index by "stripping out" its $\log n$ most significant bits, which we denote as $i = \mathsf{index}(b_i)$. Therefore, the block requires $\log n + m$ bits.

2.1 Invertible Bloom filters (IBFs)

An Invertible Bloom Filter (IBF) can be used to compactly store a set of blocks $\{b_1, b_2, \ldots, b_n\}$. Like the counting Bloom filter [6, 8], an IBF allows both insertions and deletions, and it allows the number of inserted elements to far exceed the capacity of the data structure as long as most of the inserted elements are deleted later. But, unlike the counting Bloom filter, the IBF allows the elements of the set to be enumerated. These properties make it possible to represent two large sets as small IBFs, and to quickly determine the elements in the symmetric difference of the sets. In the following, we describe the IBF data structure of Eppstein et al. [12], slightly modified to fit our setting.

IBF setup. An IBF is essentially a table (array) $\mathbf{T}$ of t cells, for a given parameter $t = (k+1)\delta$—we refer to table $\mathbf{T}$ in detail in the next section. To set up and maintain an IBF, we need the following functions: **(1)** A set of k hash functions, $h_1, h_2, \ldots, h_k$ chosen at random from a universal family of functions $\mathcal{H}$ [9], which map any integer in $[n]$ (corresponding to the index of block b_i) to k distinct cells in $\mathbf{T}$, i.e., $h_i : [n] \rightarrow \{1, 2, \ldots, t\}$. Functions h_i are used to insert a block b_i indexed i to the IBF, by computing the positions $h_j(i)$ $(j = 1, \ldots, k)$ in $\mathbf{T}$ and subsequently updating the respective fields (see Figure 1); **(2)** A function $f : \mathcal{U} \rightarrow \{0,1\}^{\tau \log \delta}$, which maps any block $b_i \in \mathcal{U}$ to a random value in the range $\{0,1\}^{\tau \log \delta}$ (recall τ is the security parameter). In our work we carefully choose this function as

$$f(b_i) = g^{a_i b_i} \mod N \,, \tag{2}$$

where N is an RSA modulus of $\tau \log \delta$ bits and $a_i = \theta_s(i)$, where θ_s is a pseudorandom function that takes input a secret s. Note that, for a specific input b_i, $f(b_i)$ depends on randomness $a_i = \theta_s(i)$.

The table T of the IBF. As we mentioned earlier, an IBF consists of table $\mathbf{T}$ of t cells. Each cell of the IBF's table $\mathbf{T}$ contains the following fields:

1. count: an integer count of the number of items mapped to this cell through functions h_i;
2. dataSum: a sum of all the blocks b mapped to this cell through functions h_i;
3. hashProd: a product of $f(b)$ values for all blocks b mapped to this cell through functions h_i.

In the dataSum field, we employ arithmetic over the integers. In the hashProd field, we employ arithmetic modulo N.

The hashProd field is used to distinguish between two IBFs having a cell i with identical dataSum value which however corresponds to different underlying blocks.

IBF algorithms. An IBF supports several simple algorithms for insertion, deletion, and enumeration of its contents.

To build an IBF $\mathbf{T}_B$ for a set of blocks B we insert each block $b_i \in B$ in the IBF via algorithm update in Figure 1, i.e., by executing $\mathsf{update}(b_i, \mathbf{T}_B, 1)$ for all $b_i \in B$. Similarly, if we want to delete a block b_i, we can execute the same algorithm with argument $z = -1$, which employs subtractions instead of additions (for the field dataSum) and divisions instead of multiplications (for the field hashProd). Note that the above algorithm is similar to the counting Bloom filters algorithms, with the difference that we also perform operations on the fields dataSum and hashProd.

The most interesting IBF operation (one we exploit in our setting) is the *algebraic difference* of two IBFs. Namely, we can take the algebraic difference of one IBF $\mathbf{T}_A$ representing a set of blocks A and IBF $\mathbf{T}_B$ representing a set of blocks B, to produce an IBF $\mathbf{T}_D$ representing the symmetric difference $D = A - B \cup B - A$. In the new IBF $\mathbf{T}_D$, produced by $\mathsf{subtract}(\mathbf{T}_A, \mathbf{T}_B)$ in Figure 2, items in $A - B$ have positive signs for their cell fields in $\mathbf{T}_D$ and items in $B - A$ have negative signs for their cell fields in $\mathbf{T}_D$.

```
Algorithm $\mathbf{T}_D \leftarrow \mathsf{subtract}(\mathbf{T}_A, \mathbf{T}_B)$
for each $i = 1, \dots, t$ do
  $\mathbf{T}_D[i].\mathsf{count} = \mathbf{T}_A[i].\mathsf{count} - \mathbf{T}_B[i].\mathsf{count}$;
  $\mathbf{T}_D[i].\mathsf{dataSum} = \mathbf{T}_A[i].\mathsf{dataSum} - \mathbf{T}_B[i].\mathsf{dataSum}$;
  $\mathbf{T}_D[i].\mathsf{hashProd} = \mathbf{T}_A[i].\mathsf{hashProd} / \mathbf{T}_B[i].\mathsf{hashProd}$;
return $\mathbf{T}_D$;
```

Figure 2: Subtracting $\mathbf{T}_A$ and $\mathbf{T}_B$. The resulting IBF $\mathbf{T}_D = \mathbf{T}_A - \mathbf{T}_B$ encodes the symmetric difference $A - B \cup B - A$.

Finally, given the IBF $\mathbf{T}_D$, produced by a subtract operation, we can enumerate its contents by using algorithm listDiff in Figure 9 in the Appendix, which was presented in previous work [12]. Algorithm listDiff repeatedly looks for cells $\mathbf{T}_D[i]$ with count fields of $+1$ or -1 and deletes these items for those cells if they pass a test for consistency. This test of consistency, described in [12], identifies if a cell positively or negatively *pure*. The definition of such a cell is the following:

Definition 1 (Pure cells) *Let $\mathbf{T}_D$ be an IBF of t cells. A cell $\mathbf{T}_D[i] \in \{1, 2, \dots, t\}$ is pure iff $\mathbf{T}_D[i].\mathsf{count} = \pm 1$ and*

$$\mathbf{T}_D[i].\mathsf{hashProd} = f(\mathbf{T}_D[i].\mathsf{dataSum})\,.$$

Depending on whether $\mathbf{T}_D[i].\mathsf{count}$ is $+1$ or -1, $\mathbf{T}_D[i]$ is called positively or negatively pure cell respectively.

As we will see, for algorithm listDiff to succeed with high probability, we require the size of all the IBFs involved to be $O(\delta)$, where δ is an upper bound on the size of the symmetric difference of sets A and B, which does not depend on the size of the sets A and B.

In the following we give an important result from [12]:

Lemma 1 (Eppstein et al. [12]) *Let A and B be two sets having at most δ blocks in their symmetric difference, and let $\mathbf{T}_A$ and $\mathbf{T}_B$ be IBFs, which are constructed by algorithm $\mathsf{update}(b_i, \mathbf{T}_A, 1)$ for all $b_i \in A$ and $\mathsf{update}(b_j, \mathbf{T}_B, 1)$ for all $b_j \in B$. The IBFs $\mathbf{T}_A$ and $\mathbf{T}_B$ have $t = (k+1)\delta$ cells and employ the function $f : \mathcal{U} \rightarrow \{0,1\}^{\Omega(\tau)}$ where $\tau \geq k \log \delta$ is the security parameter. Then with probability $O(\delta^{-k})$, algorithm $\mathsf{listDiff}(\mathbf{T}_A, \mathbf{T}_B)$ will output* reject, *failing to recover the symmetric difference $A - B \cup B - A$.*

2.2 Bitcoin

As we mentioned in the introduction, we enable automatic compensation to the client, after she figures out the damage, d. It turns out that an ideal way to do that is through Bitcoin [23].[2]

Bitcoin is a decentralized digital currency system that does not rely on financial entities. Transactions are recorded on a public ledger (the blockchain) and are verified through the collective effort of *miners*. A bitcoin address is the hash of a ECDSA public key and transactions are signed and verified by anyone in the system. Transactions are all linked to each other using hash chains. Let A and B two bitcoin addresses. A *standard* transaction contains a signature from A and mandates that a certain amount of bitcoins is to be transferred from A to B. If A's signature is valid, the transaction is inserted into a block which is then stored in the blockchain.

But Bitcoin allows for more expressive conditions than just a signature. In particular, A can specify a series of conditions that have to be satisfied before transferring bitcoins to B. To list such conditions, A uses the bitcoin scripting language which is a purposely-simple stack-based language that provides basic cryptographic functions, such as hashes and digital signatures, conditional statements, and operations on numbers and strings. In addition, each transaction specifies a *locktime* which may contain a block number or a timestamp at which the transaction is locked (before this timestamp, even if the conditions are satisfied, the transaction is not finalized). Slightly changing the notation from [3], a transaction T_x is represented as a table:

Prev :	
InputsToPrev :	
Conditions :	
Amount :	
Locktime :	

,

[2]It is worth noting that although Bitcoin's price is volatile, recent work (e.g., LOCKS [2]) aims to eliminate this problem.

where Prev represents the transaction (say T_y) that T_x is redeeming, $\mathsf{InputsToPrev}$ are inputs that T_x is sending to T_y so that T_y's redeeming can take place, $\mathsf{Conditions}$ is a program written in the Bitcoin scripting language (outputting a boolean) controlling whether T_x can be redeemed or not (given inputs from another transaction), Amount is the value in bitcoins, and $\mathsf{Locktime}$ is the locktime. For standard transactions, $\mathsf{InputsToPrev}$ is just a signature with the sender's secret key, and $\mathsf{Conditions}$ implements a signature verification with the recipient's public key. Moreover, standard transactions have locktime set to 0, meaning they are locked and final.

Andrychowicz *et al.* [3] describe an ingenious bitcoin contract mechanism to realize timed commitments [7] through the blockchain. With timed commitments in place, they show that it is possible to run fairly any secure multi-party protocols with the stipulation that parties pay a fine in bitcoins if they cheat, that is, if all parties follow the protocol specification then a function on private inputs is computed otherwise any party that deviates from (e.g., interrupts) the protocol will inescapably pay a sum of bitcoins to the others.

3 Accountable Storage model

In this section, we introduce the AS model. As we will see below, an AS sheme does not allow the client to compute damage d directly. Instead, it allows the client to use the server's proof to compute the blocks $\mathcal{L}$ that cannot be extracted by accessing the server's storage (i.e., the ones that got lost). By having the server send the current blocks he stores in the position of blocks in $\mathcal{L}$ (in addition to the proof), computing the damage d is straightforward.

Definition 2 (δ-AS scheme) *A δ-AS scheme $\mathcal{P}$ is the collection of six polynomial-time algorithms:*

1. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\tau)$ *is a key generation algorithm that is run by the client to setup the scheme. It takes a security parameter τ as input and returns a pair of matching public and secret keys* $(\mathsf{pk}, \mathsf{sk})$*;*
2. $\mathtt{T}_i \leftarrow \mathsf{TagBlock}(\mathsf{pk}, \mathsf{sk}, b_i)$ *is an algorithm run by the client to generate verification metadata for a block b_i. It takes as inputs a public key* pk*, a secret key* sk*, and a file block b_i and returns the tag $\mathtt{T}_i$;*
3. $\mathsf{state} \leftarrow \mathsf{GenState}(\mathsf{pk}, B, \delta)$ *is an algorithm run by the client to generate local state* state*. Its inputs are a public key* pk*, a set of file blocks $B = \{b_1, b_2, \ldots, b_n\}$ and the parameter δ. It returns the local state;*
4. $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$ *is an algorithm run by the client to generate a challenge for the server;*
5. $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathsf{pk}, B, \Sigma, \mathsf{chal})$ *is run by the server in order to generate a proof of accountability. It takes as inputs a public key* pk*, an ordered collection B of blocks and an ordered collection Σ which is the verification metadata corresponding to the blocks in B. It returns a proof of accountability $\mathcal{V}$ for the blocks in B;*
6. $\{\mathsf{reject}, \mathcal{L}\} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$ *is run by the client. It takes as inputs a public key* pk*, a secret key* sk *and a proof of accountability $\mathcal{V}$. It returns a list of blocks $\mathcal{L}$ or* reject.

An AS protocol between a client and a server can be constructed from a δ-AS scheme as follows. First, the client has blocks $B = \{b_1, b_2, \ldots, b_n\}$ in local storage, runs $\{\mathsf{pk}, \mathsf{sk}\} \leftarrow \mathsf{KeyGen}(1^\tau)$ and stores the pair $(\mathsf{sk}, \mathsf{pk})$. Then he computes:

1. The tags $\mathtt{T}_i \leftarrow \mathsf{TagBlock}(\mathsf{pk}, \mathsf{sk}, b_i) \forall i = 1, \ldots, n$;
2. The state $\mathsf{state} \leftarrow \mathsf{GenState}(\mathsf{pk}, B, \delta)$.

The client sends $\mathsf{pk}, B, \Sigma = (\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n)$ to the server and may delete the blocks B and the verification metadata Σ—but he locally stores the state state.

During the challenge phase, the client generates $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$ and sends it to the server which runs $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathsf{pk}, B, \Sigma, \mathsf{chal})$ and returns the proof of accountability $\mathcal{V}$. The client can check the proof $\mathcal{V}$ by executing, $\{\mathsf{reject}, \mathcal{L}\} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$ the client checks the proof $\mathcal{V}$. If the algorithm $\mathsf{CheckProof}$ does not reject, then $\mathcal{L}$ is the set of blocks that cannot be extracted by accessing the server's storage, which can be used to compute d and define the compensation.

AS correctness and security. Let B be the set of n initial blocks and $\mathcal{L} \subset B$ be another set of at most $\delta < n$ blocks. A δ-AS scheme is correct if algorithm $\mathsf{CheckProof}$, on input a proof $\mathcal{V}$ output by algorithm $\mathsf{GenProof}$ that executes correctly on the set of blocks $B - \mathcal{L}$ outputs the blocks $\mathcal{L}$. We now formalize this:

Definition 3 (δ-AS scheme correctness) *Let $\mathcal{P}$ be a δ-AS sche-me consisting of the algorithms of Definition 2. $\mathcal{P}$ is* correct *if for all $\tau \in \mathbb{N}$, for all $\{\mathsf{pk}, \mathsf{sk}\}$ output by* $\mathsf{KeyGen}(1^\tau)$*, for all sets of blocks B, for all sets of tags Σ output by* $\mathsf{TagBlock}$ *where $\Sigma = \{\mathtt{T}_j \leftarrow \mathsf{TagBlock}(\mathsf{pk}, \mathsf{sk}, b_j) : b_j \in B\}$, for all states* state *output by* $\mathsf{GenState}(\mathsf{pk}, B, \delta)$*, for all set of blocks $\mathcal{L} \subset B$ such that $|\mathcal{L}| \leq \delta$, for all challenges* chal *output by* $\mathsf{GenChal}(1^\tau)$*, for all proofs $\mathcal{V}$ output by* $\mathsf{GenProof}(\mathsf{pk}, B - \mathcal{L}, \Sigma, \mathsf{chal})$*, the probability that* $\mathcal{L} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$ *is* $1 - 1/\mathsf{poly}(\delta)$.

As we are going to see the correctness of our scheme is based on Lemma 1 that we presented before.

To define the security of a δ-AS scheme, we introduce a game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. The game is set up in such a way so that the adversary asks the challenger to compute tags on a set of blocks B of his liking. After the adversary gets access to the tags, his goal is to output a proof $\mathcal{V}$, so that if $\mathcal{L}$ is output by algorithm $\mathsf{CheckProof}$, where $|\mathcal{L}| \leq \delta$, then (a) either $\mathcal{L}$ is *not* a subset of the original set of blocks B; (b) or the adversary does *not* store all remaining blocks in $B - \mathcal{L}$ intact.

Such a proof is invalid since it would allow the verifier to either recover the *wrong* set of blocks (e.g., a set of blocks whose Hamming distance from the corrupted blocks is a lot smaller) or to accept a corruption of more than δ file blocks (i.e., the file blocks in $B - \mathcal{L}$).

Definition 4 (δ-AS scheme security) *Let $\mathcal{P}$ be a correct δ-AS scheme consisting of the algorithms of Definition 2, $\mathcal{A}$ be a PPT adversary and $\mathcal{C}$ be a challenger. Consider the following δ-AS game between $\mathcal{A}$ and $\mathcal{C}$:*

1. *Initialization. $\mathcal{C}$ runs $\{\mathsf{pk}, \mathsf{sk}\} \leftarrow \mathsf{KeyGen}(1^k)$, sends pk to $\mathcal{A}$ and keeps sk secret;*
2. *Query. For $j = 1, \ldots, n$, $\mathcal{A}$ chooses a block b_j and sends it to $\mathcal{C}$. $\mathcal{C}$ computes the tag $\mathtt{T}_j$ output by $\mathsf{TagBlock}(\mathsf{pk}, \mathsf{sk}, b_j)$ and sends $\mathtt{T}_j$ to $\mathcal{A}$. $\mathcal{A}$ stores $(b_j, \mathtt{T}_j)$ and $\mathcal{C}$ stores the block b_j;*
3. *Computing local state. Let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of blocks that were chosen by the adversary above. Then $\mathcal{C}$ computes the*
   local state $\mathsf{state} \leftarrow \mathsf{GenState}(\mathsf{pk}, B, \delta)$;
4. *Challenge. $\mathcal{C}$ outputs $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^k)$ and requests $\mathcal{A}$ to provide a proof of accountability for challenge chal;*
5. *Forge. $\mathcal{A}$ computes a proof of accountability $\mathcal{V}$ and returns $\mathcal{V}$.*

The adversary $\mathcal{A}$ wins the δ-AS game if

$$\mathcal{L} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})\,.$$

We say that the δ-AS scheme $\mathcal{P}$ is secure *if for any PPT adversary $\mathcal{A}$ the probability that $\mathcal{A}$ wins the δ-AS game is negligibly close to the probability that (i) $\mathcal{L} \subset B$* **and** *(ii) the challenger can extract* all the remaining file blocks *in $B - \mathcal{L}$ by means of a PPT knowledge extractor $\mathcal{E}$, where where $|\mathcal{L}| \leq \delta$.*

Note here that if the set $\mathcal{L}$ is empty, then the above definition is equivalent with the original PDP security definition [4].

4 Our construction

We first give an infomal overview of our construction: Suppose the client wishes to outsource to the server blocks b_1, $b_2, \ldots, b_n$. At setup phase the client computes the tag $\mathtt{T}_i$ for each block b_i. Then he produces the local state as a *partial* IBF of these blocks. We call this IBF partial because it does not store the $\mathsf{hashProd}$ field. The $\mathsf{hashProd}$ field is changing every time the client challenges the server and is computed by using the proof returned by the server.

After the end of the setup phase, the server stores the blocks and the tags. At the challenge phase, the client picks a random k-bit integer s and sends it to the server. To generate a proof of accountability (see Figure 4), the server partitions the set of indices $[n] = \{1, 2, \ldots, n\}$ into two sets, the set Kept of blocks that the server stores and the set Lost of blocks that the server has lost. The server then computes a partial IBF $\mathbf{T}_K$ on the set of blocks $\{b_i : i \in \mathsf{Kept}\}$, along with a proof of data possession [4] on the same set of blocks. For the computation of the PDP proof, the server uses randomness derived from the challenge s.[3]

The goal of the client is to output the blocks $\{b_i : i \in \mathsf{Lost}\}$, after he verifies possession of blocks $\{b_i : i \in \mathsf{Kept}\}$. To achieve that, the client takes the difference $\mathbf{T}_L = \mathbf{T}_B - \mathbf{T}_K$ and executes algorithm $\mathsf{recover}$ from Figure 5, which is a modified version of $\mathsf{listDiff}$ from Figure 9 in the Appendix.

However, the IBF $\mathbf{T}_L = \mathbf{T}_B - \mathbf{T}_K$ is still partial, being the difference of two partial IBFs $\mathbf{T}_L$ and $\mathbf{T}_K$. Recall now that in order to enumerate the elements of $\mathbf{T}_L$, we require access to the $\mathsf{hashProd}$ field, which is now absent. To confront this issue, the proof generated by the server also contains the necessary information (e.g., a few combined tags) that will allow the client to recover the field $\mathsf{hashProd}$ of the IBF $\mathbf{T}_L$, without having to explicitly divide the $\mathsf{hashProd}$ fields of the individual IBFs $\mathbf{T}_B$ and $\mathbf{T}_K$—it only requires access to the tags of the *lost* blocks, and not the blocks themselves. The detailed algorithms of our construction are in Figure 3.

Finally, we note here that as we mentioned before, algorithm $\mathsf{recover}$ (Figure 5) is a slight modification of algorithm $\mathsf{listDiff}$ from Figure 9. The main difference is that algorithm $\mathsf{recover}$ does not check for negatively pure cells. This is because we know ahead of time that the set $K - B$ (that is created through negatively pure cells in Figure 9) is always empty since, before executing algorithm $\mathsf{recover}$, we have already verified that $K \subseteq B$, through a PDP-style verification (see Relation 5). Therefore, Lemma 1 holds for algorithm $\mathsf{recover}$ as well.

[3]Note that if $\mathsf{Kept} = [n]$, then the server has not lost anything and our protocol becomes a plain PDP protocol.

Alg. $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\tau)$. Let $N = pq$ be an RSA modulus, e, d be primes such that $ed = 1 \mod \phi(N)$, g be a generator of $\mathbb{QR}_N$ and v be a τ-bit string chosen randomly. Also, a set $\mathcal{H}$ of k random hash functions $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$ such that $h_i : \mathbb{N} \to [t]$ $t = (k+1)\delta]$, a PRF $\theta_s : \{0,1\}^\tau \times \{0,1\}^{\log n} \to \{0,1\}^k$ and a random oracle $h : \{0,1\}^* \to \mathbb{QR}_N$, are initialized. Output $\mathsf{pk} = (N, g, \mathcal{H}, h, \theta_s)$ and $\mathsf{sk} = (e, d, v)$.

Alg. $\mathtt{T}_i \leftarrow \mathsf{TagBlock}(\mathsf{pk},\mathsf{sk}, b_i)$. Parse pk as $(N, g, \mathcal{H}, h, \theta_s)$, sk as (e, d, v). For block b_i set $\mathtt{W}_i = v||i$ and compute tag $\mathtt{T}_i = \left(h(\mathtt{W}_i)g^{b_i}\right)^d \mod N$. Note that we use the same tags as in the original work of Ateniese et al. [4].

Alg. $\mathsf{state} \leftarrow \mathsf{GenState}(\mathsf{pk}, B, \delta)$. Parse pk as $(N, g, \mathcal{H}, h, \theta_s)$ and B as $\{b_1, b_2, \ldots, b_n\}$. Compute the partial IBF $\mathbf{T}_B$ on the set of blocks B by executing $\mathsf{update}(b_i, \mathbf{T}_B, 1)$ for all b_i (see Figure 1) and by setting $\mathbf{T}_B[j].\mathsf{hashProd} = 1$ for all $j = 1, \ldots, t$. Output $\mathsf{state} = \mathbf{T}_B$.

Alg. $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^k)$. Pick a random $s \in \{0,1\}^k$ and output $\mathsf{chal} = s$.

Alg. $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathsf{pk}, B, \Sigma, \mathsf{chal})$. Parse pk as $(N, g, \mathcal{H}, \theta_s)$, B as $\{b_1, b_2, \ldots, b_n\}$, Σ as $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$ and $\mathsf{chal} = s \in \{0,1\}^k$. Let Kept be a set of at least $n - \delta$ indices corresponding to blocks in B (it is to the best interest of the server to make Kept as large as possible—ideally, Kept contains all indices). Set $\mathsf{Lost} = [n] - \mathsf{Kept}$. Compute a combined tag $\mathtt{T}$ along with a combined sum S as

$$\mathtt{T} = \prod_{i \in \mathsf{Kept}} \mathtt{T}_i^{a_i} \text{ and } S = \sum_{i \in \mathsf{Kept}} a_i b_i, \text{ where } a_i = \theta_s(i) \text{ is the randomness corresponding to block } i. \tag{3}$$

Following, compute the partial IBF $\mathbf{T}_K$ on the set of blocks $\{b_i : i \in \mathsf{Kept}\}$ by executing $\mathsf{update}(b_i, \mathbf{T}_K, 1)$ for all $i \in \mathsf{Kept}$ (see Figure 1) and by setting $\mathbf{T}_K[i].\mathsf{hashProd} = 1$ for $i = 1, \ldots, t$, i.e., the output IBF uses only fields count and $\mathsf{dataSum}$.
Finally, consider the t sets of indices Q_r, for $r = 1, \ldots, t$, where

$$\mathsf{Q}_r = \{i \in \mathsf{Lost} : \exists j \in [k] : h_j(i) = r\}\,. \tag{4}$$

These sets represent the positions of the lost blocks in the IBF (for an illustrative example, see Figure 4). Compute t combined tags $\mathtt{L}_1, \mathtt{L}_2, \ldots, \mathtt{L}_t$ (t is the size of the IBF) such that $\mathtt{L}_r = \prod_{i \in \mathsf{Q}_r} \mathtt{T}_i^{a_i}$ for $r = 1, \ldots, t$. The proof $\mathcal{V}$ is the tuple $\{\mathsf{Lost}, \mathtt{T}, S, \mathbf{T}_K, \mathtt{L}_1, \ldots, \mathtt{L}_t\}$.

Alg. $\{\mathsf{reject}, \mathcal{L}\} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$. Parse pk as $(N, g, \mathcal{H}, \theta_s)$, sk as (e, d, v), state as $\mathbf{T}_B$ and the proof $\mathcal{V}$ as $\mathcal{V} = \{\mathsf{Lost}, \mathtt{T}, S, \mathbf{T}_K, \mathtt{L}_1, \ldots, \mathtt{L}_t\}$. Accept (and output the set of blocks $\mathcal{L}$) if all of the following tests are true:

- Lost is a subset of $[n]$ of size at most δ;
- It is
$$\frac{\mathtt{T}^e}{\prod_{i \in \mathsf{Kept}} h(\mathtt{W}_i)^{a_i}} = g^S, \text{ where } a_i = \theta_s(i). \tag{5}$$
- (recovering the lost blocks) For the final test, we have: For $r = 1, \ldots, t$ compute
$$\mathsf{lostSum}_r = \frac{\mathtt{L}_r^e}{\prod_{i \in \mathsf{Q}_r} h(\mathtt{W}_i)^{a_i}}\,. \tag{6}$$
Note that the sets Q_r can be easily computed given the set of indices Lost (see Equation 4). Then compute $\mathbf{T}_L = \mathbf{T}_B - \mathbf{T}_K$ and set $\mathbf{T}_L[r].\mathsf{hashProd} = \mathsf{lostSum}_r$ for all $r = 1, \ldots, t$. Then execute algorithm $\mathsf{recover}(\mathbf{T}_L, \mathsf{Lost}, \mathcal{H})$ from Figure 5. If it does not reject, output the set of blocks $\mathcal{L}$ and accept.

Figure 3: Our δ-AS scheme construction.

Correctness, security and efficiency. Our proof of security is in the random oracle and is based on the RSA assumption (see Definition 6 in the Appendix). Due to space limitations, detailed proofs of correctness and security are given in the Appendix.

The local state that the client must keep is an IBF of $t = (k+1)\delta$ cells. Each cell stores a counter (which can be at most n) and the sum of at most n m-bit blocks. Therefore the asymptotic size of the state is $O((k+1)\delta(\log n + m)) = O(\delta \log n)$. For the size of the proof $\mathcal{V}$, the tag $\mathtt{T}$ has size $O(1)$, the sum S has size $O(\log n + m)$, the partial IBF $\mathbf{T}_K$ has size $O(\delta \log n)$ and the tags $\mathtt{L}_r$ ($r = 1, \ldots, t$) have size $O(t) = O(\delta k)$ since each tag $\mathtt{L}_r$ is of constant size. Overall, the size of $\mathcal{V}$ is $O(\delta \log n)$.

For the proof computation, note that algorithm $\mathsf{GenProof}$ must first access $n - \delta$ blocks in order to compute the PDP proof and then compute an IBF over the remaining blocks, therefore the time is $O(n + \delta \log n)$. Likewise, the verification algorithm needs to verify a PDP proof for a linear number of blocks and to process a proof of size $O(\delta \log n)$, thus its computation time is again $O(n + \delta \log n)$ (note that algorithm $\mathsf{recover}$ takes time $O(\delta)$, however computing the difference $\mathbf{T}_B - \mathbf{T}_k$ takes time $O(\delta \log n)$).

Theorem 1 (δ-AS scheme) *Let n be the number of blocks. For all $\delta \leq n$, there exists a δ-AS scheme such that: (1) It is*

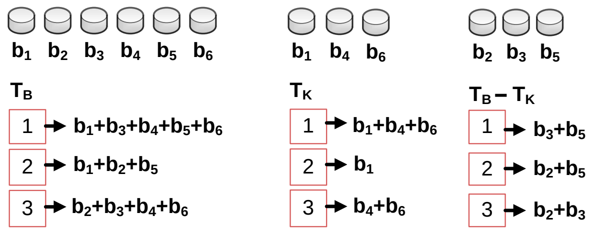


Figure 4: (**Left**) On input six blocks $b_1, b_2, \ldots, b_6$, the client outputs an IBF $\mathbf{T}_B$ of three cells, i.e., $t = 3$. For this example, we use two hash functions h_1 and h_2. (**Middle**) The server loses blocks b_2, b_3 and b_5, i.e., Lost $= \{2, 3, 5\}$. To provide a proof of accountability, the server computes $\mathbf{T}_K$ on blocks b_1, b_4, b_6. (**Right**) The client receives $\mathbf{T}_K$ and computes $\mathbf{T}_B - \mathbf{T}_K$ which gives an IBF that enables him to recover the lost blocks b_2, b_3, b_5.

```
Algorithm $\{\mathcal{L}, \text{reject}\} \leftarrow \text{recover}(\mathbf{T}_L, \text{Lost}, \mathcal{H})$
while there is a positively pure cell $\mathbf{T}_L[i]$ $(i \leq t)$ do
  $val = \mathbf{T}_L[i]$.dataSum;
  $ind = \text{index}(\mathbf{T}_L[i].\text{dataSum})$;
  if $ind \notin$ Lost then return reject;
  add $val$ to $\mathcal{L}$;
  call delete$(val, \mathbf{T}_L)$;
  Lost := Lost $- \{ind\}$;
for $i = 1, \ldots, t$ do
  if $\mathbf{T}_L[i]$.count $\neq 0$ or
    $\mathbf{T}_L[i]$.dataSum $\neq 0$ or
    $\mathbf{T}_L[i]$.hashProd $\neq 1$ or
  then return reject;
if Lost is not empty then return reject;
return $\mathcal{L}$;
```

Figure 5: Recovering the lost blocks.

correct according to Definition 3; (2) It is secure in the random oracle model based on the RSA assumption and according to Definition 4; (3) The proof has size $O(\delta \log n)$ and its computation at the server takes $O(n + \delta \log n)$ time; (4) Verification at the client takes $O(n + \delta \log n)$ time and requires local state of size $O(\delta \log n)$; (5) The space at the server is $O(n)$.

Streaming and appending blocks. We note here that our construction assumes the client has all the blocks available in the beginning, computes the local state and then outsources the blocks to the server. This is not necessary. Our construction applies to a streaming setting as well (e.g., [27]), where the blocks b_i are coming one at a time, the client easily updates its local state with algorithm update$(b_i, \mathbf{T}, 1)$ from Figure 1, computes the new tag $\mathtt{T}_i$ and sends the pair $(b_i, \mathtt{T}_i)$ to the server for storage. This also means that our construction is partially-dynamic, supporting append-only updates. Modifying a block is not so straightforward due to replay attacks. However techniques from various fully-dynamic PDP schemes could be potentially used for this problem (e.g., [13]).

4.1 An optimized construction

In the previous construction, the server and client run in $O(n + \delta \log n)$ time. In this section we present two algorithmic optimizations that can be used to eventually reduce the server and client performance to $O(\delta \log n)$.

Randomized PDP checking. Recall that the main overhead of algorithms CheckProof and GenProof is due to a verification of a PDP proof that has been computed over $n - \delta$ blocks. To reduce this overhead, we can use randomized block checking as proposed by Ateniese et al. [4] in their original work. To incorporate this optimization, our protocol must be modified in the following ways:

1. Set $\lambda = 2 \cdot \max\{\delta, \tau\}$, where τ is the security parameter;
2. The algorithm GenChal, apart from the challenge $s \in \{0, 1\}^k$ (that is used to produce randomness a_i) outputs another challenge $s' \in \{0, 1\}^k$ to be used as an input to a new PRF that is going to output a *random* subset $\mathcal{R} \subseteq [n]$ of λ

indices;

3. The server runs algorithm $\mathsf{GenProof}$ as before. However, instead of computing the PDP proof over all the blocks in Kept (see Relation 3), the server computes the PDP proof on the blocks in a smaller set, i.e., the set $\mathcal{Y} = \mathcal{R} \cap \mathsf{Kept}$, where $\mathcal{R}$ was produced by the challenge s' picked at random by the client. Note that, by the way that λ is computed in Item 1, it is always $\tau \leq |\mathcal{Y}| \leq 2 \cdot \max\{\delta, \tau\}$.
4. Finally, the client runs the verification algorithm as is, with the difference that he checks the PDP proof only for the blocks in $\mathcal{Y} = \mathcal{R} \cap \mathsf{Kept}$. Namely, instead of computing the product over all the blocks in Kept $\prod_{i \in \mathsf{Kept}} h(\mathtt{W}_i)^{a_i}$, he now needs to compute the product $\prod_{i \in \mathcal{Y}} h(\mathtt{W}_i)^{a_i}$ which takes time $O(\tau + \delta)$ (instead of $O(n)$). Given that τ is a constant (i.e., not dependent on n), the verification now takes time $O(\delta \log n + \tau + \delta) = O(\delta \log n)$.

However, the theoretical guarantees of such an approach are not that good. This is because, it could be the case, that the index of some block that has been tampered with *and* belongs to the set Kept is *not* included in the set of random indices $\mathcal{Y}$. If the set Kept contains f such blocks, the probability p_f that all indices in $\mathcal{Y}$ "land" in locations different than the locations of the f tampered file blocks (which will cause the PDP verification to accept) is at most

$$p_f = \left(\frac{n - \delta - f}{n - \delta}\right)^{|\mathcal{Y}|} \leq \left(\frac{n - \delta - f}{n - \delta}\right)^{\tau},$$

since $|\mathcal{Y}| = \Omega(\tau)$ and $(n - \delta - f)/(n - \delta) < 1$, for any $f > 0$. As such, the probability p_f is a lot larger than $\mathsf{neg}(\tau)$. However, for all practical purposes, Ateniese et al. [4] showed that the guarantees of such an approach can be acceptable. For example, one can always adjust this probability if one increases the number of λ, even in the case of small f—and of course the best guarantees are achieved when f is a constant fraction of n.

Segment tree. The technique we described in the previous section allows the server to compute the PDP proof (which is part of the accountability proof) a lot faster. However, apart from the PDP proof, an accountability proof contains the IBF $\mathbf{T}_K$ over the blocks with indices in Kept. Unfortunately, computing the IBF $\mathbf{T}_K$ requires accessing $n - \delta$ blocks and therefore the server time remains still linear. In this section, we show how to compute the IBF $\mathbf{T}_K$ in $O(\delta \log n)$ time, by using a data structure called segment tree [26].

A segment tree T_B is a binary search tree that stores a set B of n key-value pairs (i, b_i) at the leaves of the tree, ordered by the key. Let v be an internal node of the tree T_B. Denote with $\mathsf{cover}(v)$ the set of keys that are included in the leaves of the subtree rooted on node v. Every internal node v of the tree has a label $\mathsf{label}(v)$ that stores the *sum of the values* corresponding to the keys in $\mathsf{cover}(v)$. Namely $\mathsf{label}(v) = \sum_{i \in \mathsf{cover}(v)} b_i$. By using the segment tree, one can compute the sum S of *any subset* of $n - \alpha$ values (blocks) in $O(\alpha \log n)$ time (instead of taking $O(n - \alpha)$ time): To see that, if $i_1, i_2, \ldots, i_\alpha$ are the indices of the omitted α blocks, the desired sum can be written as

$$\sum_{i \neq \{i_1, i_2, \ldots, i_\alpha\}} b_i = \sum_{i=1}^{i_1 - 1} b_i + \sum_{i=i_1+1}^{i_2 - 1} b_i + \ldots + \sum_{i=i_\alpha+1}^{n} b_i$$

and each one of the above partial sums can be computed in $O(\log n)$ time by accessing a logarithmic number of internal nodes of the segment tree.

We modify our protocol in order to use the above data structure as follows. First, the server maintains *one segment tree* T_l per IBF cell $l = 1, \ldots, t$. Namely, whenever the client uploads a block b_j, the server inserts b_j to the k segment trees $T_{h_i(j)}$ for $i = 1, \ldots, k$. Note that such an operation (i.e., updating the segment tree) takes $O(\log n)$ time.

The server can now use the segment trees to compute the IBF $\mathbf{T}_K$, i.e., for each cell i of $\mathbf{T}_K$ the server uses the segment tree T_i to compute the sum over a set of $n - \alpha_i$ blocks (specifically this is the subset of the blocks in Kept that map to cell i), where $i = 1, \ldots, t$ and $\sum_{i=1}^{t} \alpha_i = \delta k$. This task takes time

$$\sum_{i=1}^{t} \alpha_i \log n = \log n \sum_{i=1}^{t} \alpha_i = O(\delta \log n).$$

This optimization, combined with the previous one, brings the computation of the server down to $O(\delta \log n)$.

5 Bitcoin Integration

After the client computes the damage d using the AS protocol described in the previous section, we would like to enable automatic compensation by the server to the client in the amount of d bitcoins. The server initially makes a "security deposit" of A bitcoins by means of a special bitcoin transaction that automatically transfers A bitcoins to the client unless the server transfers d bitcoins to the client before a given deadline. Here, the amount A is a parameter that is contractually established by the client and server and is meant to be larger than the maximum damage that can be incurred by the server. Specifically, we have designed a variation of the AS protocol integrated with bitcoin that, upon termination, achieves one of the following outcomes within an established deadline:

- If both the server and the client follow the protocol, the client gets exactly d bitcoins from the server and the server gets back his A bitcoins.
- If the server does not follow the protocol (e.g., he tries to give fewer than d bitcoins to the client, fails to respond in a timely manner, or tries to forge an AS proof), the client gets A bitcoins from the server automatically.
- If the client does not follow the protocol (e.g., she requests more than d bitcoins from the server), the server receives all A deposited bitcoins back and the client receives nothing.

To guarantee the above outcomes, we implement the security deposit of A bitcoins by the server via a special Bitcoin transaction $\mathsf{safeGuard}(x, t)$, and the related transactions $\mathsf{retBtcs}$ and fuse, described in Section 5.3 and depicted with a diamond in Figure 6. This transaction is based on the timed commitment over Bitcoin recently introduced by Andrychowicz et al. [3], where x is the committed value and t is the bitcoin locktime.

The functionality associated with $\mathsf{safeGuard}(x, t)$ guarantees the following: (1) if x is known by the server then $\mathsf{safeGuard}(x, t)$ can be opened and the server (and only the server) can get his A bitcoins back; (2) after time t all A bitcoins will go to the client. Namely, until either x is revealed or t has passed, the bitcoins of the cloud in the transaction $\mathsf{safeGuard}(x, t)$ are effectively frozen. We emphasize here that the $\mathsf{safeGuard}$ transaction we are implementing has an important difference from the one introduced in [3]: the committed value x is chosen by the verifier (client) and not by the committer (server).

We now describe our protocol in detail, as depicted in Figure 6. Let S denote the server and C the client. Our protocol involves a trusted "Bitcoin Arbitrator" (BA). However, we note that the BA is only contacted by S and never by the client C (more on this later). For each step $i = 1, \ldots, 7$, there is a deadline, t_i, to complete the step, where timelock $t >> t_7$. We also assume neither the client nor the sever can forge the timestamped transcript of the protocol, which can be verified by BA. This can be accomplished via standard techniques: all messages are inside a single authenticated session where messages are signed by both parties, and contain nonces and a summary (typically a hash) of all previous messages to avoid tampering.

- **Step 1:** C picks a random secret x and sends the following items to S: (*i*) an encryption $\mathsf{Enc}_\mathsf{P}(x)$ of x under BA's public key, P; (*ii*) a cryptographic hash of x, $\mathsf{H}(x)$; and (*iii*) a zero-knowledge proof, ZKP_1, that $\mathsf{H}(x)$ and $\mathsf{Enc}_\mathsf{P}(x)$ encode the same secret x. If ZKP_1 does not verify, S aborts the protocol.
- **Step 2:** S posts bitcoin transaction $\mathsf{safeGuard}(x, t)$ for A bitcoins with timelock t, as done in [3]. If this transaction is not posted within time t_2 (the server is not following the protocol), C aborts the protocol.
- **Step 3:** S and C run the AS protocol from the previous section, which results in C computing the damage, d. If the AS protocol rejects or S delays it past time t_3, C jumps to Step 9.
- **Step 4:** C notifies S that the damage is d and sends a zero-knowledge proof, ZKP_2, to S for that. If C fails to do so by time t_4 or ZKP_2 does not verify, S jumps to Step 6.
- **Step 5:** S sends d bitcoins to C. If S has not done so by time t_5, C jumps to Step 9.
- **Step 6:** C sends secret x to S. If S has not received x by t_6, S contacts BA and sends the timestamped transcript of the protocol up to that moment. BA checks the transcript and if it is valid, BA sends x to S. Note that the transcript must be tamper-proof and should contain the encryption of x *and* all messages exchanged up to that moment.
- **Step 7:** If S has secret x, S posts transaction $\mathsf{retBtcs}$ (i.e., S opens the timed commitment using x).
- **Step 8:** If transaction $\mathsf{retBtcs}$ is valid, S receives A bitcoins before timelock t.
- **Step 9:** C waits until time t and posts transaction fuse.
- **Step 10:** If transaction fuse is valid, C receives A bitcoins.

It is easy to see that when the above protocol terminates, one of the three outcomes described at the beginning of this section is achieved. Also, we note here that for the zero-knowledge proofs ZKP_1 and ZKP_2, we can use a SNARK with zero-knowledge [25], that was recently implemented and shown to be practical.

5.1 Global safeGuard

The protocol above protects the client at each AS challenge. But the cloud provider could simply disappear and never be reachable by the client. Our accountable framework thus establishes that there must be a *global* safeGuard transaction at the time the client and the server initiate their business relationship (i.e., when the client uploads the original file blocks and they both sign the SLA). This global transaction is meant to protect the client if: (1) the server cannot be reached at all or (2) refuses to post the safeGuard transaction during any AS challenge, or (3) posts the safeGuard transaction but asks BA to recover the bitcoins in it without participating in the AS challenge. The global safeGuard transaction has the same format

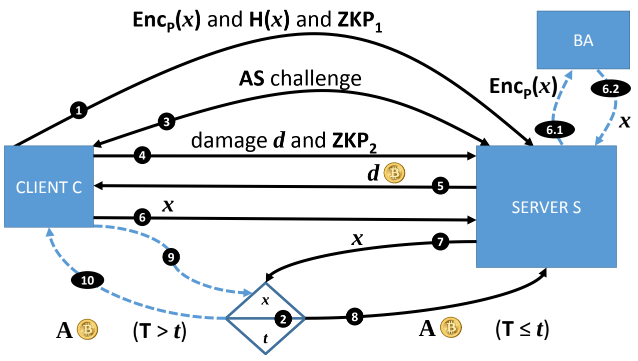


Figure 6: Integration of the AS protocol with Bitcoin to enable an automatic compensation of the client in the case of data loss.

as the per-challenge safeGuard transaction and can be handled by the same BA. The only significant difference is that now the BA must mediate and interact with both the client and the server before returning any bitcoins to the server and, in case, reinitiate a new global safeGuard transaction between the client and the server.

5.2 Removing the Bitcoin Arbitrator

Even though the client never communicates with the BA as per our goal, it is preferable to remove it completely. Unfortunately, this seems impossible to achieve *efficiently* given the limitations of the Bitcoin scripting language. However, it is possible to remove the BA, at least in theory, via a secure two-party protocol. In a secure two-party protocol (2PC), party A inputs x and party B inputs y and they want to compute $f_A(x, y)$ and $f_B(x, y)$ respectively, without learning each other's input other than what can be inferred from the output of the two functions. Yao's seminal result [33] showed that oblivious transfer implies 2PC secure against honest-but-curious adversaries. This result can be extended to generically deal with malicious adversaries through zero-knowledge proofs at each stage of the 2PC or more efficiently via the *cut-and-choose* method of Lindell and Pinkas [20] or LEGO and MiniLEGO by Nielsen and Orlandi [24] and by Frederiksen et al. [14] (other efficient solutions have been proposed by Woodruff [32] and Jarecki and Shmatikov [18]).

To remove the BA, it is enough to create a symmetric version of our original scheme where both parties create a safeGuard transaction and then exchange the secrets of both commitments through a fair exchange protocol embedded into a 2PC. The secrets must be verifiable in the sense that the fair exchange must ensure that the secrets open the initial commitments or fail (as in "committed 2PC" by Jarecki and Shmatikov [18]). Unfortunately, generic techniques for 2PC results in quite impractical schemes and this is the reason why we prefer a practical solution with an arbiter. An efficient 2PC protocol with Bitcoin is proposed in [22] but it does not provide fairness as it is claimed since the 2PC protocol can be interrupted at any time by one of the parties. In the end, since this generic approach is too expensive in practice, we will not elaborate on it any further in this paper.

5.3 The safeGuard transaction

In this section, we describe the safeGuard(x, t) is detail. The scheme is set up so that safeGuard(x, t) is created by S but where x is known only to C—S only knows $h = \mathsf{H}(x)$. This is feasible to achieve through the first step on the protocol described in Figure 6. We now describe the bitcoin transactions using the notation in Section 2.2. Since S knows h, he makes the following Bitcoin transaction, which we call safeGuard:

Prev :	aTransaction
InputsToPrev :	$\mathsf{sig}_S([\mathsf{safeGuard}])$
Conditions :	$\underline{body, \sigma_1, \sigma_2, x :}$ $\mathsf{H}(x) = h \wedge \mathsf{ver}_S(body, \sigma_1)$ $\vee$ $\mathsf{ver}_S(body, \sigma_1) \wedge \mathsf{ver}_C(body, \sigma_2)$
Amount :	A ฿
Locktime :	0

.

The above transaction redeems a transaction called aTransaction that has at least A Ƀ as value and can be redeemed by transactions whose InputsToPrev are of the type $body, \sigma_1, \sigma_2, x$ (i.e., two signatures on the same transaction and an integer x) and satisfy the conditions specified.

Once the client reveals x to the server, the server posts the following transaction retBtcs to recover his A bitcoins.

Prev :	safeGuard
InputsToPrev :	$[\text{retBtcs}], \text{sig}_S([\text{retBtcs}]), \perp, x$
Conditions :	$\frac{body, \sigma :}{\text{ver}_S(body, \sigma)}$
Amount :	A Ƀ
Locktime :	0

.

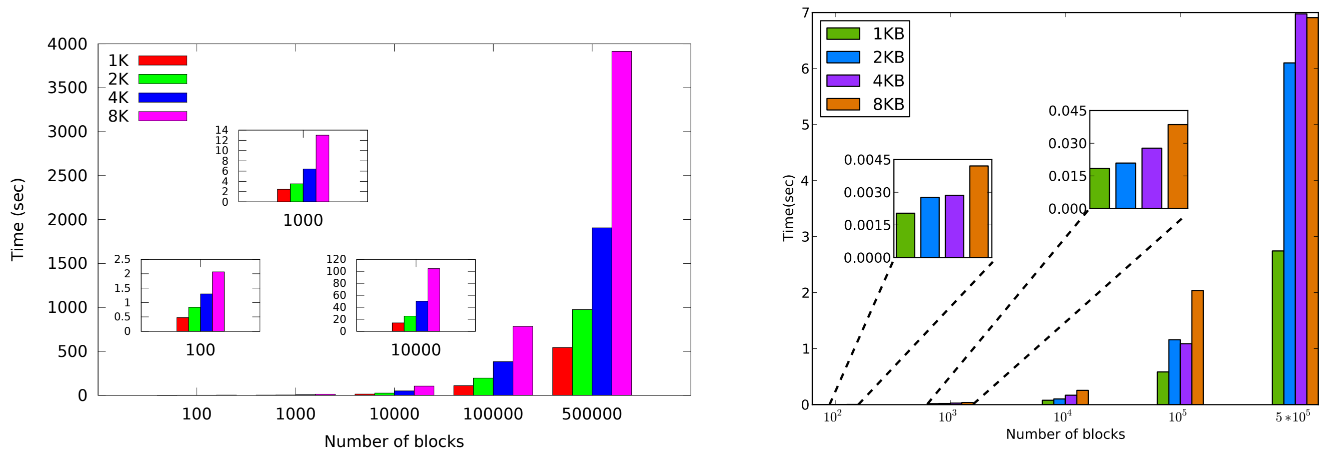
If the server does not cooperate (e.g., see Step 5 in the protocol description), the client publishes the transaction fuse after time t and receives a compensation of A bitcoins from the server.

Prev :	safeGuard
InputsToPrev :	$[\text{fuse}], \text{sig}_S([\text{fuse}]), \text{sig}_C([\text{fuse}]), \perp$
Conditions :	$\frac{body, \sigma :}{\text{ver}_C(body, \sigma)}$
Amount :	A Ƀ
Locktime :	t

.

It is very important to notice here that the locktime of the above transaction is t, meaning that if the transaction is posted earlier than t, it is not going to be accepted. This is what enables A Ƀ to be transferred to C if the server does not know x (which C controls!). Finally we note that the transaction Fuse is possible only because client C has already the server's signature $\text{sig}_S([\text{fuse}]$ on the body [fuse], which includes the locktime t. We can assume that this signature is obtained in the beginning of the protocol.

6 Evaluation



(a) Tag Generation. (b) Local IBF Generation.

Figure 7: Preprocessing overheads.

We prototyped the proposed δ-AS scheme in Python 2.7.5. Our implementation consists of 4K lines of source code. We use the pycrypto library 2.6.1 [21] and an RSA modulus N of size 1024 bits. We serialize the protocol messages using Google Protocol Buffers [16] and performed all the modulo exponentation operations using GMPY2 [17], which is a C-coded Python extension module that supports fast multiple precision arithmetic (the use of GMPY2 gave us 60% speedup in exponentiations in comparison with the regular python arithmetic library).

We divide the prototype in two major components. The first is responsible for data pre-processing, issuing proof challenges and verifying proofs. The second produces proof every time it receives a challenge. Both modules utilize the IBF

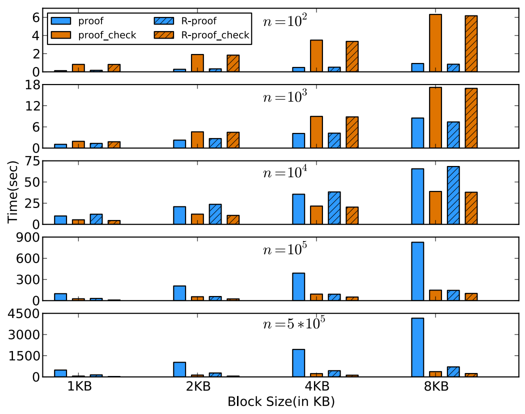


Figure 8: Proof Generation and Recover time.

Table 1: Memory Footprint of the δ-AS Scheme (KB)

n	Tag Size (KB)	Proof Size (KB)			
		1KB	2KB	4KB	8KB
10^2	49	578	1140	2254	4515
10^3	409	1787	3526	7003	13955
10^4	4375	4111	8114	16118	32125
10^5	42206	12983	25629	50913	101482
$5 * 10^5$	204910	29047	57328	113904	227029

data structure to produce and verify proofs. Our prototype uses parallel computing via the Python multiprocessing module to carry out many of the heavy, but independent, cryptographic operations simultaneously. We used a *single-producer, many-consumers* approach to divide the available tasks in a pool of [8-12] processes-workers. The workers use message passing to coordinate and update the results of their computations. This approach significantly enhanced the performance of preprocessing as well as the proof generation and checking phase of the protocol. Our parallel implementation provides an approximate 5x speedup over a sequential implementation.

Experimental Setup: Our experimental setup involves two nodes, one implementing a server and another implementing a client functionality. The two nodes communicate through a Local Area Network (LAN). The two machines are equipped with an Intel 2.3 Ghz Core i7 processor and have 16 GB of RAM.

Our data are randomly generated filesystems. Every file-system includes different number of equally-sized blocks. The number of blocks ranges from 100 to 500000. The total filesystem size varies from 100
KByte to 4.1 Gbyte. Our experiments consist of 10 trials of challenge/proof exchanges between the client and the server for different filesystems. Throughout the evaluation we report the average values over these 10 trials.

Preprocessing Overheads: We first examine the memory overhead of the preprocessing phase, which is shown in Table 1. The first column describes the available number of blocks in a filesystem and the second represents the total size of the tags needed. The preprocessing memory overhead is proportional to the number of blocks in a filesystem.

Figure 7 shows the CPU-time-related overheads of the preprocessing of the protocol. These overheads are divided to tag generation and the creation of the client state represented by the partial IBF T_B. The tag generation time (Figure 7a) increases linearly with both the available number of blocks. While this cost is significant for large file systems, it is an operation that client performs only once. On the other hand, the cost of construction of the partial IBF (Figure 7b) is negligible; the IBF construction of our biggest filesystem only takes 7 seconds. The construction cost does not significantly increase with the number of blocks because it does not involve any cryptographic operations.

Challenge-proof Overheads: We now examine memory and CPU related overheads for the challenge-proof exchange and the recovery phase. The last four columns show the proof sizes (in KB) for $\delta = \sqrt{n}$, which increase proportionally to the block size (unlike the tags size).

Every subgraph of Figure 8 shows how different block sizes affect the performance of the challenge-proof exchange for a given number of blocks. The solid bars represent the baseline construction (see Section 4), with the left bar showing the proof generation time and the right bar the proof check along with the time to recover the lost blocks. Longer block sizes increase the time-overhead of challenge-proof exchange due to the expensive modulo exponentiations that are dominant in this stage of the protocol.

Figure 8 demonstrates an important trade-off between efficiency and performance. The baseline version of the protocol produces and verifies a proof by scanning all the available blocks in a filesystem. This provides low probability of PDP proof failure; this approach, however, is quite expensive.

One way to mitigate the cost of the baseline scheme is to reduce the amount of data scanned during the proof generation/check steps of the protocol, as we explain in Section 4.1. The randomized PDP checking (Section 4.1) serves this purpose. The hatched parts in Figure 8 show the performance of this randomized approach, which consistently outperforms the baseline version, especially for larger filesystems. We noticed a speedup of $9\times$ in the proof generation for the case of $5 * 10^5$ blocks with 8KB of block sizes. It achieves that by examining a random subset λ of the available blocks. Figure 8 shows the performance of the randomized approached when $\lambda = 15\%$ of the available blocks n. Despite its efficiency, this approach comes with a higher probability of PDP proof failure. For example, for $\delta = 13$, $n = 10,000$ and $f = 10$ then the probability p_f for PDP proof failure is 0.12.

The good performance properties of the randomized sche-me makes the scheme appealing for real-life applications. For example, in a cloud based scenario, a client may use it to verify, in real time, if the cloud provider meets the SLA guarantees.

δ-tolerance: To examine the actual effect of the parameter δ on our implementation, we experimented with a larger value of δ on both the baseline and the randomized version of the protocol. Table 2 reports the average values over 10 trials of challenge/proof exchange for three important steps of the protocol. The parameter δ affects only the recovery process of the δ-AS scheme. This is expected, since the other two parts (proof check and proof generation) operate over the total number of available blocks in the filesystem. The same principles hold for the memory related overheads.

Table 2: Effect of δ for $n = 500,000$ and Block size 8KB for baseline (B) and randomized (R) versions of δ-AS scheme.

	$\delta = 18$		$\delta = 707$	
	Proof Size (KB)	Proof Recovery (sec)	Proof Size (KB)	Proof Recovery (sec)
B	5799.0	2.6	227029.0 Kb	107.2
R	5591.0	2.2	209011.0 Kb	100.13

7 Conclusions

In this paper we put forth the notion of accountability in cloud storage. Unlike existing work such as proof-of-storage schemes and verifiable computation, we design protocols that respond to a verification failure, enabling the client to assess the damage that has occurred in a storage repository. We also present a protocol that enables automatic compensation of the client, based on the amount of damage, and is implemented over Bitcoin. Our implementation shows that our system can be used in practice.

References

[1] https://bitcoin.org/bitcoin.pdf.

[2] Locks. http://www.coindesk.com/coinapult-launches-locks-tool-eliminate-bitcoin-price-volatility/ .

[3] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *IEEE SSP*, 2014.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007.

[5] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, pages 9:1–9:10, 2008.

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

[7] D. Boneh and M. Naor. Timed commitments. In *CRYPTO*, pages 236–254, 2000.

[8] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *ESA*, volume 4168, pages 684–695, 2006.

[9] I. L. Carter and M. N. Wegman. Universal classes of hash functions. In *ACM STOC*, pages 106–112, 1977.

[10] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, pages 279–295, 2013.

[11] R. Curtmola, O. Khan, R. C. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *ICDCS*, pages 411–420, 2008.

[12] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese. What's the difference?: efficient set reconciliation without prior context. In *SIGCOMM*, pages 218–229, 2011.

[13] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, pages 213–222, 2009.

[14] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *EUROCRYPT*, pages 537–556. 2013.

[15] M. T. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. *ArXiv e-prints*, January 2011.

[16] Google. Google protocol buffers. `https://developers.google.com/protocol-buffers/`.

[17] C. V. Horsen. Gmpy2: Mupltiple-precision arithmetic for python. `https://gmpy2.readthedocs.org/en/latest/intro.html/`.

[18] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114. 2007.

[19] A. Juels and J. Burton S. Kaliski. PORs: proofs of retrievability for large files. In *ACM CCS*, pages 584–597, 2007.

[20] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78. 2007.

[21] D. C. Litzenberger. Pycrypto - the python cryptography toolkit. `https://www.dlitz.net/software/pycrypto/`.

[22] D. M. Marcin Andrychowicz, Stefan Dziembowski and L. Mazurek. Fair two-party computations via the bitcoin deposits. In *FC*. 2014.

[23] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[24] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *Theory of Cryptography*, pages 368–386. 2009.

[25] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[26] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, Oct. 1990.

[27] D. Schröder and H. Schröder. Verifiable data streaming. In *ACM CCS*, pages 953–964, 2012.

[28] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, pages 90–107, 2008.

[29] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336, 2013.

[30] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A scalable cloud file system with efficient integrity checks. In *ACSAC*, 2012.

[31] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*, pages 355–370, 2009.

[32] D. P. Woodruff. Revisiting the efficiency of malicious two-party computation. In *EUROCRYPT*, pages 79–96. 2007.

[33] A. C.-C. Yao. How to generate and exchange secrets. In *SFCS*, pages 162–167, 1986.

[34] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *TOS*, 3(3), 2007.

Appendix

Definition 5 (Negligible function) *Function $\lambda : \mathbb{N} \rightarrow \mathbb{R}$ is $\mathsf{neg}(\tau)$ iff for any nonzero polynomial $p(\tau)$ there is N such that for all $\tau > N$ it is $\lambda(\tau) < 1/p(\tau)$.*

The security of our construction is based on the RSA assumption and holds in the random oracle model. The RSA assumption is stated as follows:

Definition 6 (RSA assumption) *Let $N = pq$ be an RSA modulus, where p and q are two τ-bit primes. Given N, e and y, where y is randomly chosen from $\mathbb{Z}_N^*$ and e is a a prime of $\Theta(\tau)$ bits, there is no PPT algorithm that can output $y^{1/e} \bmod N$, except with negligible probability* $\mathsf{neg}(\tau)$.

```
**Algorithm** ${(D^+, D^-), \mathsf{reject}} \leftarrow \mathsf{listDiff}(\mathbf{T}_A, \mathbf{T}_B)$
$\mathbf{T}_D = \mathsf{subtract}(\mathbf{T}_A, \mathbf{T}_B)$;
 while there is a pure cell $\mathbf{T}_D[j]$ $(j = 1, \ldots, t)$ do
  Set $b_i = \mathbf{T}_D[j]$.dataSum; ($i$ is the block index)
  if $\mathbf{T}_D[j]$ is positively pure then
   add $b_i$ to $D^+$;
   call $\mathsf{delete}(b_i, \mathbf{T}_D)$;
  else
   add $-b_i$ to $D^-$;
   call $\mathsf{insert}(-b_i, \mathbf{T}_D)$;
for $j = 1, \ldots, t$ do
  if $\mathbf{T}_D[j]$.count $\neq 0$ or
    $\mathbf{T}_D[j]$.dataSum $\neq 0$ or
    $\mathbf{T}_D[j]$.hashProd $\neq 1$
  then return reject;
return $(D^+, D^-)$;
```

Figure 9: Listing the data blocks that are contained in the symmetric difference $A - B \cup B - A$. For example, if $A = \{b_1, b_2, b_3\}$ and $B = \{b_2, b_5\}$, it is $D^+ = A - B = \{b_1, b_3\}$ and $D^- = B - A = b_5$.

Proof of correctness

We prove correctness by contradiction. Suppose there exist a $k \in \mathbb{N}$, a pair $\{\mathsf{pk}, \mathsf{sk}\}$ output by $\mathsf{KeyGen}(1^k)$, a set of blocks B, a set of tags Σ output by algorithm TagBlock, i.e., $\Sigma = \{\mathtt{T}_j \leftarrow \mathsf{TagBlock}(\mathsf{pk}, \mathsf{sk}, b_j) : b_j \in B\}$, a state state output by algorithm $\mathsf{GenState}(\mathsf{pk}, B)$, a set $\mathcal{L} \subseteq B$ that contain at most δ blocks, a challenge chal output by algorithm $\mathsf{GenChal}(1^k)$, a proof $\mathcal{V}$ output by algorithm $\mathsf{GenProof}(\mathsf{pk}, B - \mathcal{L}, \Sigma, \mathsf{chal})$, such that the probability that
$\mathcal{L}' \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal}) \wedge \mathcal{L} \nsubseteq \mathcal{L}'$ is greater than $\mathsf{neg}(k)$. Note that algorithm GenProof executes of the set of blocks $B - \mathcal{L}$, therefore, by insepcting the code of GenProof, set Lost always contains a superset of the indices of the blocks in $\mathcal{L}$. Let now $\mathcal{V} = \{\mathsf{Lost}, \mathtt{T}, S, \mathbf{T}_K, \mathtt{L}_1, \ldots, \mathtt{L}_t\}$ be the proof that algorithm GenProof outputs, where

1. $\mathtt{T} = \prod_{i \in \mathsf{Kept}} \mathtt{T}_i^{a_i}$;
2. $S = \sum_{i \in \mathsf{Kept}} a_i b_i$;
3. $\mathbf{T}_K$ is the partial IBF on the set of blocks $\{b_i : i \in \mathsf{Kept}\}$ which is computed by executing $\mathsf{insert}(b_i, \mathbf{T}_K)$ for all $i \in \mathsf{Kept}$ (see Figure 1). Set $\mathbf{T}_K$.hashProd $= 1$, i.e., the output IBF uses only fields count and dataSum;
4. $\mathtt{L}_r = \prod_{i \in \mathsf{Q}_r} \mathtt{T}_i^{a_i}$ for $r = 1, \ldots, t$, where Q_r is defined in Relation 4.

Note now that if one takes the difference $\mathbf{T}_L = \mathsf{state} - \mathbf{T}_K$ and then set

$$\mathbf{T}_L[r].\mathsf{hashProd} = \frac{\mathtt{L}_r^e}{\prod_{i \in \mathsf{Q}_r} h(\mathtt{W}_i)^{a_i}},$$

it is easy to see that $\mathbf{T}_L$ corresponds to the IBF representing blocks in Lost, namely a set of blocks $\mathcal{L}' \supseteq \mathcal{L}$. Executing algorithm $\mathsf{recover}(\mathbf{T}_L, \mathsf{Lost}, \mathcal{H})$ will produce that set of blocks $\mathcal{L}'$, with probability at least $1 - \mathsf{neg}(k)$, by Lemma 1. This is a contradiction and concludes the proof. □

Proof of security

To prove security, we play the game given in Definition 4 between a challenger $\mathcal{C}$ and the adversary $\mathcal{A}$. Eventually, we show how the challenger will break the RSA assumption with the help of the adversary. The challenger $\mathcal{C}$ simulates a PDR environment for $\mathcal{A}$ as follows:

Initialization. $\mathcal{C}$ computes $g = y^2 \mod N \in \mathbb{QR}_N$, picks a set of k random hash functions $\mathcal{H}$ and sets the public key $\mathsf{pk} = (N, g, \mathcal{H}, h, \theta_s)$. Note that $h(.)$ is modeled here as a random oracle, which is programmed by the challenger. Finally $\mathcal{C}$ generates a random secret value $v \in \{0,1\}^k$ and a large secret prime e.

Query. $\mathcal{C}$ answers $\mathcal{A}$'s tagging queries using a random oracle as follows. When $\mathcal{C}$ receives a tagging query for a block b indexed $1 \leq i \leq n$, in the case that a previous tagging query for that block has occurred before, $\mathcal{C}$ retrieves the stored tuple $(b, i, r_i, \mathtt{W}_i)$ and returns $\mathtt{T}_i = r_i$. Otherwise, the challenger $\mathcal{C}$ picks a random $r_i \in \mathbb{QR}_N$, computes $\mathtt{W}_i = v||i$, stores the tuple $(b, i, r_i, \mathtt{W}_i)$ and returns $\mathtt{T}_i = r_i$. Also, when hash queries for input x are made for the first time by $\mathcal{A}$, $\mathcal{C}$ picks a random value $\omega_x \in \mathbb{QR}_N$ and returns $h(x) = \omega_x$. He also stores the tuple (x, ω_x) so that he can answer consistently the next time he is asked to return $h(x)$. Finally, $\mathcal{C}$ programs the random oracle $h()$ such that, on input $\mathtt{W}_i$ it gives

$$h(\mathtt{W}_i) = r_i^e g^{-b_i} \mod N\,,$$

where b_i is the block indexed i chosen by the adversary.

Computing local state. After all the tagging queries for blocks $B = \{b_1, b_2, \ldots, b_n\}$ have been made, $\mathcal{C}$ stores all the blocks $B = \{b_1, b_2, \ldots, b_n\}$. Finally, $\mathcal{C}$ computes the state $\mathsf{state} = \mathbf{T}_B$ as in algorithm $\mathsf{GenState}$. $\mathcal{C}$ keeps the state state locally.

Challenge. $\mathcal{C}$ requests $\mathcal{A}$ to provide a proof of data recovery for blocks $b_1, b_2, \ldots, b_n$ and sends a challenge chal, which is a random $s \in \{0,1\}^k$.

Forge. $\mathcal{A}$ computes a proof of data recovery $\mathcal{V}$ and returns $\mathcal{V}$. Parse $\mathcal{V}$ as $\{\mathsf{Lost}, \mathtt{T}, S, \mathbf{T}_K, \mathtt{L}_1, \ldots, \mathtt{L}_t\}$. Suppose algorithm $\mathsf{CheckProof}$ accepts. This means that algorithm $\mathsf{recover}(\mathbf{T}_L, \mathsf{Lost}, \mathcal{H})$ also accepts, outputting a set of blocks $\mathcal{L}$. Suppose for contradiction that $\mathcal{L} \nsubseteq B$. This means there exists at least one block $b_j \in \mathcal{L}$ such that $b_j \notin B$. Since the blocks in $\mathcal{L}$ have indices in $[n]$ (this is due to the fact that algorithm $\mathsf{recover}$ only adds to $\mathcal{L}$ blocks with indices in Lost and Lost is always a subset of $[n]$), this means that there exists a block $b_j' \in B$ such that $b_j' \neq b_j$. Namely set B contains a block with index j (i.e., the correct index) but different value b_j'.

Let $z \in \{1, \ldots, t\}$ be an index of the IBF such that j maps to z, i.e., there is u for some $u = 1, \ldots, k$ such that $h_(j) = z$. Consider the remaining block indices $i_1, i_2, \ldots i_w$ mapping that index z and define Q_z as

$$\mathsf{Q}_z = \{i_1, i_2, \ldots i_w, j\}\,.$$

By the verification Equation 6, it needs to hold

$$\mathsf{lostSum}_z = \frac{\mathtt{L}_z^e}{\prod_{i \in \mathsf{Q}_z} h(\mathtt{W}_i)^{a_i}} = \frac{\mathtt{L}_z^e}{\prod_{i \in \mathsf{Q}_z} r_i^e g^{-a_i b_i}}\,. \tag{7}$$

However, for $\mathsf{recover}$ to accept, $\mathsf{lostSum}_z$ should be equal to 1 at the end of the algorithm. Since $\mathsf{lostSum}_z$ is updated through delete, the initial value of $\mathsf{lostSum}_z$ is

$$\mathsf{lostSum}_z = g^{\bar{S}},\ \text{where } \bar{S} = \sum_{u=1}^{w} a_{i_u} b_{i_u}' + a_j b_j',$$

where $\{b_{i_1}', b_{i_2}', \ldots, b_{i_w}', b_j'\} \subseteq \mathcal{L}$. Therefore Equation 7 can be written as

$$g^{\bar{S}} = \frac{\mathtt{L}_z^e}{\prod_{i \in \mathsf{Q}_z} r_i^e g^{-a_i b_i}} \Leftrightarrow g^{\bar{S}-S} = \frac{\mathtt{L}_z^e}{\prod_{i \in \mathsf{Q}_z} r_i^e}\,, \tag{8}$$

where $S = \sum_{u=1}^{w} a_{i_u} b_{i_u} + a_j b_j$ and $\{b_{i_1}, \ldots, b_{i_w}, b_j\}$ are the *original blocks*. But we know that $b_j \neq b_j'$ therefore the probability of $\bar{S} = S$ is negligible (since the a_i's are picked at random). Therefore it is $\bar{S} \neq S$ and Equation 8 can be written as

$$g^{\bar{S}-S} = \frac{\mathtt{L}_z^e}{\prod_{i \in \mathsf{Q}_z} r_i^e} = \left(\frac{\mathtt{L}_z}{\prod_{i \in \mathsf{Q}_z} r_i}\right)^e = \mathcal{Z}^e\,.$$

Therefore we have $\mathcal{Z}^e = g^{\bar{S}-S} = y^{2(\bar{S}-S)}$. Now, note that it is $\mathsf{gcd}(e, 2(\bar{S}-S)) = 1$ with overwhelming probability (since e is a large prime unknown to $\mathcal{A}$). Therefore we can apply "Shamir's trick", and use the extended Euclidean algorithm to find integers α and β such that $\alpha \times e + \beta \times 2(\bar{S}-S) = 1$. This gives $y^{1/e} = y^{\alpha}\mathcal{Z}^{\beta}$, breaking the RSA assumption. □