Accountable Storage

Giuseppe Ateniese[1], Michael T. Goodrich[2], Vassilios Lekakis[3], Charalampos Papamanthou[5], Evripidis Paraskevas[5(✉)], and Roberto Tamassia[4]

[1] Department of Computer Science, Stevens Institute of Technology, Hoboken, USA
gatenies@stevens.edu
[2] Department of Computer Science, University of California, Irvine, USA
goodrich@uci.edu
[3] Department of Computer Science, University of Maryland, College Park, USA
lex@cs.umd.edu
[4] Department of Computer Science, Brown University, Providence, USA
rt@cs.brown.edu
[5] Department of Electrical and Computer Engineering, University of Maryland, College Park, USA
cpap@umd.edu, evripar@terpmail.umd.edu

Abstract. We introduce *Accountable Storage* (AS), a framework enabling a client to outsource n file blocks to a server while being able (any time after outsourcing) to provably compute how many bits were discarded or corrupted by the server. Existing techniques (e.g., proofs of data possession or storage) can address the accountable storage problem, with linear server computation and bandwidth. Instead, our optimized protocols achieve $O(\delta \log n)$ complexity (where δ is the maximum number of corrupted blocks that can be tolerated) through the novel use of invertible Bloom filters and a new primitive called *proofs of partial storage*. With accountable storage, a client can be compensated with a dollar amount proportional to the number d of corrupted bits (that he can now provably compute). We integrate our protocol with Bitcoin, supporting automatic such compensations. Our implementation is open-source and shows our protocols perform well in practice.

1 Introduction

Cloud computing is revolutionizing our digital world, posing new security and privacy challenges. E.g., businesses and individuals are reluctant to outsource their databases for fear of having their data lost or damaged. Thus, they would benefit from technologies that would allow them to manage their risk of data loss, just like 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, and fortunately, *Provable Data Possession* (PDP) [3,11,13] and *Proofs of Retrievability* (POR) [10,19,26–28], have been conceived as a solution to the integrity problem of remote databases. PDP and

D. Gollmann et al. (Eds.): ACNS 2017, LNCS 10355, pp. 623–644, 2017.
DOI: 10.1007/978-3-319-61204-1_31

POR scheme can verify whether the server possesses the database originally uploaded by the client by having the server generate a proof in response to a challenge.

However, they leave unsettled several risk management issues. Arguably, an important question is

What happens if a PDP or POR scheme shows that a client's outsourced database has been damaged?

The objective of this work is to design new efficient protocols for *Accountable Storage* (AS) that enable the client to reliably and quickly assess the damage and at the same time automatically get compensated using the Bitcoin protocol.

To be precise, suppose 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 wishes to *provably* compute the value of a *damage metric*, such as

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

where b'_i is the file currently stored by Bob at the time of the challenge, $||.||$ denotes Hamming distance and w_i is a weight corresponding to file b_i. If $d = 0$, Alice is entitled to no dollar credit. Bob can easily prove to Alice that this is the case through existing protocols, as noted above. If $d > 0$, however, then Alice should receive a compensation proportional to the damage d, which should be provided automatically.[1]

Naive Approaches for AS. A PDP protocol [3,4,11,13,29] 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 that the server possesses. This could determine the damage, d (e.g., when all weights w_i are equal to 1). However this approach requires using of PDP at the bit level, and in particular computing one 2048-bit tag for each bit of our file collection which is very storage-inefficient.

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 (e.g., computed with an error-correcting code). 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

[1] We highlight that such fine-grained compensation models, which work at the bit level as opposed to at the file block level, 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 a usable movie whose resolution has deteriorated by just 5%.

the δ local check blocks (then the recovered blocks can be used to compute d). This procedure has $O(n)$ communication, since the $n - \delta$ blocks at the server must be sent to the client. IRIS [28] is a system along these lines, requiring the whole file system be streamed over to the client for recovery.

Finally we note here that while PDP techniques combined with redundant blocks stored at the client can be used to solve the accountable storage problem (even inefficiently, as shown above), POR techniques cannot. This is because POR techniques (e.g., [26]) cannot provide proofs of retrievability for a certain portion of the file (as is the case with PDP), but only for the whole file—this is partly due to the fact that error-correcting codes are used on top of all the file blocks.

Our AS Protocol. Our protocol for assessing damage d from Relation 1 is based on *recovering the actual blocks* $b_1, b_2, \ldots, b_\delta$ *and XORing them with the corrupted blocks* $b'_1, b'_2, \ldots, b'_\delta$ *returned by the server*. For recovery, we use the invertible Bloom filter (IBF) data structure [12,15]. An IBF is an array of t cells and can store $O(t)$ elements. Unlike a Bloom filter [7], its elements can be enumerated with high probability.

Let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of outsourced blocks and let δ be the maximum number of corrupted blocks that can be tolerated. In preprocessing, the client computes an IBF $\mathbf{T}_B$ with $O(\delta)$ cells, on the blocks $b_1, \ldots, b_n$. $\mathbf{T}_B$ is stored locally. Computing $\mathbf{T}_B$ is similar to computing a Bloom filter: every cell of $\mathbf{T}_B$ is mapped to a XOR over a set of at most n blocks, thus the local storage is $O(\delta)$. To outsource the blocks, the client computes homomorphic tags, $\mathtt{T}_i$ (as in [3]), for each block b_i. The client then stores $(b_i, \mathtt{T}_i)$ with the cloud and deletes $b_1, b_2, \ldots, b_n$ from local storage. In the challenge phase, the client asks the server to construct an IBF $\mathbf{T}_K$ of $O(\delta)$ cells on the set of blocks K the server currently has—this is the "proof" the server sends to the client. Then the client takes the "difference" $\mathbf{T}_L = \mathsf{subtract}(\mathbf{T}_B, \mathbf{T}_K)$ and recovers the elements of the difference $B - K$ (since $|B - K| \leq \delta$ and $\mathbf{T}_L$ has $O(\delta)$ cells). Recovering blocks in $B - K$ enables the client to compute d using Relation 1. Clearly, the bandwidth of this protocol is proportional to δ (due to the size of the IBFs), and not to the total number of outsourced blocks n. Our optimized construction in Sect. 5 achieves sublinear server and client complexities as well.

Fairness Through Integration with Bitcoin. The above protocol assures that Bob (the server) cannot succeed in persuading Alice that the damage of her file blocks is $d' < d$. After Alice is persuaded, compensation proportional to d must be sent to her. But Bob could try to cheat again. Specifically, Bob could try to give Alice a smaller compensation or even worse, disappear. To deal with this problem, we develop a modified version of the recently-introduced timed commitment in Bitcoin [2]. 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 Sect. 6 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.

Structure of the Paper. Section 2 presents background on IBFs and Bitcoin, Sect. 3 gives definitions, and Sects. 4 and 5 present our constructions. We present our Bitcoin protocol in Sect. 6, our evaluation in Sect. 7 and conclude in Sect. 8.

2 Preliminaries

Let τ denote the security parameter, δ denote an upper bound on the number of corrupted blocks that can be tolerated, n denote the number of file blocks, and $b_1, b_2, \ldots, b_n$ denote the file blocks. Each block b_i has λ bits. The first $\log n$ bits of each block b_i are used for storing the index i of the block, which can be retrieved through function $\mathsf{index}()$. Namely $i = \mathsf{index}(b_i)$. Let also $h_1, h_2, \ldots, h_k$ be k hash functions chosen at random from a universal family of functions $\mathcal{H}$ [9] such that $h_i : \{0,1\}^\lambda \to \{1, 2, \ldots, t\}$ for some parameter t.

Invertible Bloom Filters. An *Invertible Bloom Filter* (*IBF*) [12,15] can be used to compactly store a set of blocks $\{b_1, b_2, \ldots, b_n\}$: It uses a table (array) $\mathbf{T}$ of $t = (k+1)\delta$ cells. Each cell of the IBF's table $\mathbf{T}$ contains the following two fields[2]: (1) $\mathsf{dataSum}$: XOR of blocks b_i mapped to this cell; (2) $\mathsf{hashSum}$: XOR of cryptographic tags (to be defined later) $\mathtt{T}_i$ for all blocks b_i mapped to this cell. As in Bloom filters, we use functions $h_1, \ldots, h_k$ to decide which blocks map to which cells.

An IBF supports simple algorithms for insertion and deletion via algorithm update in Fig. 1. For $B \subseteq A$, one can also take the *difference* of IBFs $\mathbf{T}_A$ and $\mathbf{T}_B$, to produce an IBF $\mathbf{T}_D \leftarrow \mathsf{subtract}(\mathbf{T}_A, \mathbf{T}_B)$ representing the difference set $D = A - B$. Finally, given $\mathbf{T}_D$, we can enumerate its contents by using algorithm $\mathsf{listDiff}$ from [12]:

Lemma 1 (Adjusted from Eppstein et al. [12]). *Let $B \subseteq A$ be two sets having $\leq \delta$ blocks in their difference $A - B$, let $\mathbf{T}_A$ and $\mathbf{T}_B$ be their IBFs constructed using k hash functions and let $\mathbf{T}_D \leftarrow \mathsf{subtract}(\mathbf{T}_A, \mathbf{T}_B)$. All IBFs have $t = (k+1)\delta$ cells and their $\mathsf{hashSum}$ field is computed using a function mapping blocks to at least $k \log \delta$ bits. Then there is an algorithm $\mathsf{listDiff}(\mathbf{T}_D)$ that recovers $A - B$ with probability $1 - O(\delta^{-k})$.*

Bitcoin Basics. Bitcoin [23] is a decentralized digital currency system where 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 an ECDSA public key. Let A and B be two bitcoin addresses. A *standard* transaction contains a signature from A and mandates a certain amount of bitcoins 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.

[2] Note that we do not use the count field, as in [12,15].

Algorithm $\mathbf{T} \leftarrow \mathsf{update}(b_i, \mathbf{T})$

for each $j = 1, \ldots, k$ **do**
  Set $\mathbf{T}[h_j(b_i)].\mathsf{dataSum} \oplus = \ b_i$;
  Set $\mathbf{T}[h_j(b_i)].\mathsf{hashSum} \oplus = \ \mathtt{T}_i$;
return $\mathbf{T}$;

Algorithm $\mathbf{T}_D \leftarrow \mathsf{subtract}(\mathbf{T}_A, \mathbf{T}_B)$

for each $i = 1, \ldots, t$ **do**
  $\mathbf{T}_D[i].\mathsf{dataSum} = \mathbf{T}_A[i].\mathsf{dataSum} \oplus \mathbf{T}_B[i].\mathsf{dataSum}$;
  $\mathbf{T}_D[i].\mathsf{hashSum} = \mathbf{T}_A[i].\mathsf{hashSum} \oplus \mathbf{T}_B[i].\mathsf{hashSum}$;
return $\mathbf{T}_D$;

Fig. 1. Update and subtraction algorithms in IBFs.

Bitcoin allows for more complicated transactions (as we are using here), whose validation requires more than just a signature. In particular, each transaction can specify a *locktime* containing a timestamp t at which the transaction is locked (before time t, even if a valid signature is provided, the transaction is not final). Slightly changing the notation from [2], a Bitcoin transaction T_x can be represented as the table below, where Prev is the transaction (say T_y) that T_x is redeeming, InputsToPrev are inputs that T_x is sending to T_y so that T_y's redeeming can take place, 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 Locktime is the locktime. For standard transactions, InputsToPrev is a signature with the sender's secret key, and Conditions implements a signature verification with the recipient's public key. Also, standard transactions have locktime set to 0, meaning they are locked and final.

Prev :
InputsToPrev :
Conditions :
Amount :
Locktime :

3 Accountable Storage Definitions

We now define an AS scheme. An AS scheme does not allow the client to compute damage d directly. Instead, it allows the client to use the server's proof to retrieve the blocks $\mathcal{L}$ that are not stored by the server any more (or are stored corrupted). 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 1 (δ-AS scheme). *A δ-AS scheme $\mathcal{P}$ is the collection of four PPT algorithms:*

1. $\{\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathsf{T}_1, \ldots, \mathsf{T}_n\} \leftarrow \mathsf{Setup}(b_1, \ldots, b_n, \delta, 1^\tau)$ *takes as inputs file blocks $b_1, \ldots, b_n$, a parameter δ and the security parameter τ and returns a public key* pk*, a secret key* sk*, tags $\mathsf{T}_1, \ldots, \mathsf{T}_n$ and a client state* state*.*
2. $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$ *generates a challenge for the server;*
3. $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathsf{pk}, \beta_{i_1}, \ldots, \beta_{i_m}, \mathsf{T}_{i_1}, \ldots, \mathsf{T}_{i_m}, \mathsf{chal})$ *takes as inputs a public key* pk*, a collection of $m \leq n$ blocks and their corresponding tags. It returns a proof of accountability $\mathcal{V}$;*
4. $\{\mathsf{reject}, \mathcal{L}\} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$ *takes as inputs a public key* pk *and a proof of accountability $\mathcal{V}$. It returns a list of blocks $\mathcal{L}$ or* reject*.*

Relation to Proofs of Storage. A δ-AS scheme is a generalization of proof-of-storage (PoS) schemes, such as [3,19]. In particular, a 0-AS scheme (i.e., where we set $\delta = 0$) is equivalent to PoS protocols, where there is no tolerance for corrupted/lost blocks.

Definition 2 (δ-AS scheme correctness). *Let $\mathcal{P}$ be a δ-AS scheme. Let $\{\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathtt{T}_1, \ldots, \mathtt{T}_n\} \leftarrow \mathsf{Setup}(b_1, \ldots, b_n, \delta, 1^\tau)$ for some set of blocks $B = \{b_1, \ldots, b_n\}$. Let now $\mathcal{L} \subseteq B$ such that $|\mathcal{L}| \leq \delta$, $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$ and $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathsf{pk}, B - \mathcal{L}, \mathtt{T}(B - \mathcal{L}), \mathsf{chal})$, where $\mathtt{T}(B - \mathcal{L})$ denotes the tags corresponding to the blocks in $B - \mathcal{L}$. A δ-AS scheme is correct if the probability that $\mathcal{L} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$ is at least $1 - \mathsf{neg}(\tau)$.*[3]

To define the security of a δ-AS scheme, the adversary adaptively asks for tags on a set of blocks $B = \{b_1, b_2, \ldots, b_n\}$ that he chooses. 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.

Definition 3 (δ-AS security). *Let $\mathcal{P}$ be a δ-AS scheme as in Definition 1 and $\mathcal{A}$ be a PPT adversary. We define security using the following steps.*

1. ***Setup.*** *$\mathcal{A}$ chooses $\delta \in [0, n)$, blocks $B = \{b_1, b_2, \ldots, b_n\}$ and is given $\mathtt{T}_1, \ldots, \mathtt{T}_n$ and pk output by $\{\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathtt{T}_1, \ldots, \mathtt{T}_n\} \leftarrow \mathsf{Setup}(b_1, \ldots, b_n, \delta, 1^\tau)$.*[4]
2. ***Forge.*** *$\mathcal{A}$ is given $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$ and outputs a proof of accountability $\mathcal{V}$.*

Suppose $\mathcal{L} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$. We say that the δ-AS scheme $\mathcal{P}$ is secure if, with probability at least $1 - \mathsf{neg}(\tau)$: (i) $\mathcal{L} \subseteq B$; and (ii) there exists a PPT knowledge extractor $\mathcal{E}$ that can extract all the remaining file blocks *in $B - \mathcal{L}$.*

Note here that if the set $\mathcal{L}$ is empty, then the above definition is equivalent to the original PDP security definition [3]. Also note that the notion of a knowledge extractor is similar to the standard one, introduced in the context of proofs of knowledge [5]. If the adversary can output an accepting proof, then he can execute $\mathsf{GenProof}$ repeatedly until it extracts the selected blocks.

[3] Function $\lambda : \mathbb{N} \rightarrow \mathbb{R}$ is $\mathsf{neg}(\tau)$ iff $\forall$ nonzero polynomials $p(\tau)$ there exists N so that $\forall \tau > N$ it is $\lambda(\tau) < 1/p(\tau)$.

[4] $\mathcal{A}$ could also choose blocks adaptively, after seeing tags for already requested blocks. Our proof of security handles that.

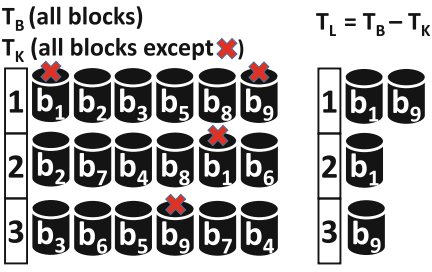
4 Our Basic Construction

We now give an overview of our basic construction: On input blocks $B = \{b_1, \ldots, b_n\}$ in local storage, the client decides on a parameter δ (meaning that he can tolerate up to δ corrupted files) and computes the local state, tags, public and secret key by running $\{\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathtt{T}_1, \ldots, \mathtt{T}_n\} \leftarrow \mathsf{Setup}(b_1, \ldots, b_n, \delta, 1^\tau)$. In our construction the tag $\mathtt{T}_i$ is set to $(h(i)g^{b_i})^d \mod N$, as in [3], where $h(.)$ is a collision-resistant hash function, N is an RSA modulus and (e, d) denote an RSA public/private key pair. The client then sends blocks $b_1, \ldots, b_n$ and tags $\mathtt{T}_1, \ldots, \mathtt{T}_n$ to the server and locally stores the state state, which is an IBF of the blocks $b_1, b_2, \ldots, b_n$.

At challenge phase, the client runs $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$ that picks a random challenge s and sends it to the server. To generate a proof of accountability (see Fig. 2-left) with $\mathsf{GenProof}$, the server computes an IBF $\mathbf{T}_K$ on the set of blocks that he (believes he) stores, along with a proof of data possession [3] on the same set of blocks. The indices of these blocks are stored in a set Kept. For the computation of the PDP proof, the server uses randomness derived from the challenge s.

To verify the proof, the client takes the difference $\mathbf{T}_L = \mathsf{subtract}(\mathbf{T}_B, \mathbf{T}_K)$ and executes algorithm $\mathsf{recover}$ from Fig. 2-right, which is a modified version of $\mathsf{listDiff}$ from [12]. Algorithm $\mathsf{recover}$ adds blocks whose tags verify to the set of lost blocks $\mathcal{L}$. Then it checks the PDP proof for those block indices corresponding to blocks that were not output by $\mathsf{recover}$. If this PDP proof does not reject, then the client is persuaded that the server stores everything except for blocks in $\mathcal{L}$. To make sure $\mathsf{recover}$ does not fail with a noticeable probability, our construction sets the parameters according to the following corollary. The detailed algorithms of our construction are in Fig. 3.

Corollary 1. *Let τ be the security parameter and B and K be two sets such that $K \subseteq B$ and $|B - K| \leq \delta$. Let $\mathbf{T}_B$ and $\mathbf{T}_K$ be IBFs constructed by algorithm*



Algorithm $\mathcal{L} \leftarrow \mathsf{recover}(\mathbf{T}_L)$
$\mathcal{L} = \emptyset$;
while there is a cell $i \leq t$ such that
$\mathbf{T}_L[i].\mathsf{hashSum}^e / h(ind) = g^{\mathbf{T}_L[i].\mathsf{dataSum}}$
(where $ind = \mathsf{index}(\mathbf{T}_L[i].\mathsf{dataSum})$
do
add $\mathbf{T}_L[i].\mathsf{dataSum}$ to $\mathcal{L}$;
call $\mathsf{update}(\mathbf{T}_L[i].\mathsf{dataSum}, \mathbf{T}_L)$;
return $\mathcal{L}$;

Fig. 2. (**Left**) On input $b_1, b_2, \ldots, b_9$, the client outputs an IBF $\mathbf{T}_B$ of three cells using two hash functions. The server loses blocks b_1 and b_9. $\mathbf{T}_K$ is computed on blocks $b_2, b_3, \ldots, b_8$ and $\mathbf{T}_L$ contains the lost blocks b_1 and b_9. (**Right**) The algorithm for recovering the lost blocks.

Alg. $\{\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathtt{T}_1, \ldots, \mathtt{T}_n\} \leftarrow \mathsf{Setup}(b_1, \ldots, b_n, \delta, 1^\tau)$. Let

- $N = pq$ be an RSA modulus of 2τ bits.
- Let e be a random prime and d be a number such that $ed = 1 \mod \phi(N)$.
- $\mathcal{H}$ be a set of k random hash functions $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$ such that $h_i : \{0,1\}^* \to [t]$, where $t = (k+1)\delta$ and $k = \tau / \log \delta$.
- g be a generator of $\mathbb{QR}_N$.
- $\theta_s : \{0,1\}^\tau \times \{0,1\}^* \to \{0,1\}^\tau$ be a pseudorandom function.
- $h : \{0,1\}^* \to \mathbb{QR}_N$ be a collision-resistant hash function (modeled as random oracle).

Set $\mathsf{pk} = (N, e, \mathcal{H}, g, \theta_s, h)$ and $\mathsf{sk} = d$. For block b_i compute tag

$$\mathtt{T}_i = \left(h(i) g^{b_i}\right)^d \mod N .$$

Set state to be the IBF $\mathbf{T}_B$ on $b_1, \ldots, b_n$ computed using Algorithm update from Figure 1.

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

Alg. $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathsf{pk}, b_{i_1}, \ldots, b_{i_m}, \mathtt{T}_{i_1}, \ldots, \mathtt{T}_{i_m}, \mathsf{chal})$. Let $\{b_{i_1}, \ldots, b_{i_m}\} \subseteq \{b_1, \ldots, b_n\}$ be the blocks that the server stores. Set $\mathsf{Kept} = \{i_1, i_2, \ldots, i_m\}$.

1. (PDP proof) Compute a combined tag $\mathtt{T}$ along with a combined sum S as

$$\mathtt{T} = \prod_{i \in \mathsf{Kept}} \mathtt{T}_i^{a_i} \text{ and } S = \sum_{i \in \mathsf{Kept}} a_i b_i, \text{ where } a_i = \theta_{\mathsf{chal}}(i). \tag{2}$$

2. (IBF proof) Compute IBF $\mathbf{T}_K$ on the set of blocks $\{b_i : i \in \mathsf{Kept}\}$.

The proof $\mathcal{V}$ is the tuple $\{\mathtt{T}, S, \mathbf{T}_K\}$.

Alg. $\{\mathsf{reject}, \mathcal{L}\} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$. Parse $\mathcal{V}$ as $\{\mathtt{T}, S, \mathbf{T}_K\}$, state as $\mathbf{T}_B$.

1. Set $\mathbf{T}_L \leftarrow \mathsf{subtract}(\mathbf{T}_B, \mathbf{T}_K)$
2. Set $\mathcal{L} \leftarrow \mathsf{recover}(\mathbf{T}_L)$ and $\mathsf{Kept} = [n]$.
3. For each $b \in \mathcal{L}$, set $\mathsf{Kept} = \mathsf{Kept} - \{\mathsf{index}(b)\}$.
4. Output reject if $\mathtt{T}^e / \prod_{i \in \mathsf{Kept}} h(i)^{a_i} \neq g^S$, where $a_i = \theta_{\mathsf{chal}}(i)$. Otherwise output $\mathcal{L}$.

Fig. 3. Our δ-AS scheme construction.

update *of Fig. 1 using* $\tau / \log \delta$ *hash functions. The IBFs* $\mathbf{T}_B$ *and* $\mathbf{T}_K$ *have* $t = (\tau / \log \delta + 1)\delta$ *cells and employ tags in the* hashSum *field that map blocks to* τ *bits. Then with probability at least* $1 - 2^{-\tau}$*, algorithm* recover(subtract($\mathbf{T}_B, \mathbf{T}_K$)) *will output* $\mathcal{L} = B - K$.

Our detailed proof of security is given in the Appendix. The local state that the client must keep is an IBF of $t = (k+1)\delta$ cells, therefore the asymptotic size of the state is $O(\delta)$. For the size of the proof $\mathcal{V}$, the tag $\mathtt{T}$ has size $O(1)$, the sum S has size $O(\log n + \lambda)$ and the IBF $\mathbf{T}_K$ has size $O(\delta)$. Overall, the size of $\mathcal{V}$ is $O(\delta + \log n)$. For the proof computation, note that algorithm GenProof must first access at least $n - \delta$ blocks in order to compute the PDP proof and then compute an IBF of δ cells over the same blocks, therefore the time is $O(n + \delta)$. 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 $O(n + \delta)$.

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

We now make two observations related to our construction. First, note that the server could potentially launch a DoS attack, by *pretending it does not store some of the blocks* so that the client is forced to spend cycles retrieving these blocks. This is not an issue, since as we will see later, the server will be penalized for that, so it is not in its best interest. Second, note that the tags that the client initially uploads are publicly verifiable so anyone can check their validity—therefore the client cannot upload bogus tags and blame the server later for that.

Streaming and Appending Blocks. Our construction assumes the client has all blocks available in the beginning. This is not necessary. Blocks b_i could come one at a time, and the client could easily update its local state with algorithm $\mathsf{update}(b_i, \mathbf{T}, 1)$, compute the new tag $\mathtt{T}_i$ and send 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]).

5 Sublinear Construction Using Proofs of Partial Storage

In the previous construction, the server and client run in $O(n + \delta)$ time. In this section we present optimizations that reduce the server and client performance to $O(\delta \log n)$. Recall that the proof generation in Fig. 3 has two distinct, linear-time parts: First, proving that a subset of blocks is kept intact (in particular the blocks with indices in Kept), and second, computing an IBF on this set of blocks. We show here how to execute both these tasks in sublinear time using (i) partial proofs of storage; (ii) a data structure based on segment trees that the client must prepare during preprocessing.

Proofs of Partial Storage. In our original construction, we prove that a subset of blocks is kept intact (in particular the blocks with indices in Kept) using a PDP-style proof, as originally introduced by Ateniese et al. [3]. In our new construction we will replace that part with a new primitive called *proofs of partial storage*. To motivate proofs of partial storage, let us recall how proofs of storage [26] work.

Proofs of storage provide the same guarantees with PDP-style proofs [3] but are much more practical in terms of proof construction time. In particular, one can construct a PoS proof in constant time as follows. Along with the original blocks $b_1, b_2, \ldots, b_n$ the client outsources an additional n redundant blocks $\beta_1, \beta_2, \ldots, \beta_n$ computed with an error-correcting code such as Reed-Solomon,

such that any n out of the $2n$ blocks $b_1, b_2, \ldots, b_n, \beta_1, \beta_2, \ldots, \beta_n$ can be used to retrieve the original blocks $b_1, b_2, \ldots, b_n$. Also, the client outsources tags $\mathtt{T}_i$ (as computed in Algorithm Setup in Fig. 3) for all $2n$ blocks. Now, during the challenge phase, the client picks a constant-sized subset of random blocks to challenge (out of the $2n$ blocks), say $\tau = 128$ blocks. Because the subset is chosen at random every time, the server, with probability at least $1 - 2^{-\tau}$, will pass the challenge (i.e., provide verifying tags for the challenged blocks) only if he stores at least half of the blocks $b_1, b_2, \ldots, b_n, \beta_1, \beta_2, \ldots, \beta_n$—which means that the original blocks $b_1, b_2, \ldots, b_n$ are recoverable.

Unfortunately, we cannot use proofs of storage as described above directly, since we want to prove that a *subset of the blocks* is stored intact, and the above construction applies to the whole set of blocks. In the following we describe how to fix this problem using a segment-tree-like data structure.

Our New Construction: Using a Segment Tree. A segment tree T is a binary search tree that stores the 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. Denote with $\mathsf{cover}(v)$ the set of blocks that are included in the leaves of the subtree rooted on node v. Let also $|v| = |\mathsf{cover}(v)|$. Every internal node v of T has a label $\mathsf{label}(v)$ that stores:

1. All blocks $b_1, b_2, \ldots, b_{|v|}$ contained in $\mathsf{cover}(v)$ along with respective tags $\mathtt{T}_i$. The tags are computed as in Algorithm Setup in Fig. 3;
2. Another $|v|$ redundant blocks $\beta_1, \beta_2, \ldots, \beta_{|v|}$ computed using Reed-Solomon codes such that *any* $|v|$ out of the $2|v|$ blocks $b_1, b_2, \ldots, b_{|v|}, \beta_1, \beta_2, \ldots, \beta_{|v|}$ are enough to retrieve the original blocks $b_1, b_2, \ldots, b_{|v|}$. Along with every redundant block β_i, we also store its tag $\mathtt{T}_i$.
3. An IBF T_v on the blocks contained in $\mathsf{cover}(v)$;

By using the segment tree, one can compute functions on *any subset* of $n - \delta$ blocks in $O(\delta \log n)$ time (instead of taking $O(n - \delta)$ time): For example, if $i_1, i_2, \ldots, i_\delta$ are the indices of the omitted δ blocks, the desired IBF T_K can be computed by combining (i.e., XORing the dataSum and hashSum fields and adding the count fields):

- The IBF T_1 corresponding to indices from 1 to $i_1 - 1$;
- The IBF T_2 corresponding to indices from $i_1 + 1$ to $i_2 - 1$;
- ...
- The IBF $T_{\delta+1}$ corresponding to indices from $i_\delta + 1$ to i_n.

Each one of the above IBFs can be computed in $O(\log n)$ time by combining a logarithmic number of IBFs stored at internal nodes of the segment tree and therefore the total complexity of computing the final IBF T_K is $O(\delta \log n)$. Similarly, a partial proof of storage for the lost blocks with indices $i_1, i_2, \ldots, i_\delta$ can be computed by returning.

– A proof of storage corresponding to indices from 1 to $i_1 - 1$;
– A proof of storage corresponding to indices from $i_1 + 1$ to $i_2 - 1$;
– ...
– A proof of storage corresponding to indices from $i_\delta + 1$ to i_n.

Again, each one of the above proofs of storage can be computed by returning $O(\log n)$ partial proofs of storage so in total, one needs to return $O(\delta \log n)$ proofs of storage. Note however that our segment tree increases our space to $O(n \log n)$ and also setting it up requires $O(n \log n)$ time. Therefore we have the following:

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

6 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.[5]

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:

1. 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.
2. 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.
3. If the client requests more than d bitcoins from the server by providing invalid evidence, the server receives all A deposited bitcoins back and the client receives nothing.

[5] Of course, this is just a simple setting, a proof of concept. Clearly other technical and financial instruments can be used to improve this approach if committing such a large amount of A bitcoins is too demanding.

Primitives Used in Our Protocol. Our protocol is using two primitives, which we describe informally in the following.

- A *trusted and tamper-resilient channel* between the client and the server, e.g., a bulletin board. This can be easily implemented by requesting all messages exchanged between the server and the client be posted on the blockchain (note that it is easy for a party P to post arbitrary data D on the blockchain by making a transaction to itself and by including D in the body of the transaction). From now on, we will assume that all messages are posted to the blockchain creating a history *hist*.
- A *trusted bitcoin arbitrator* BA. This is a trusted party that only intervenes in case of disputes. BA will always examine the history of transactions *hist* to assess the situation and determine whether to help the server. In all other cases it can remain offline.

Bitcoin Transactions. Our protocol uses three non-standard bitcoin transactions:

1. $\mathsf{safeGuard}(y)$: This transaction is posted by the server S and it effectively "freezes" A bitcoins to a hash output y. It can be redeemed by a transaction (called $\mathsf{retBtcs}$) posted by the server S which provides the preimage x of $y = \mathsf{H}(x)$ or by a transaction (called $\mathsf{fuse}(t)$) signed by both the client and the server. For the needs of our protocol, $\mathsf{fuse}(t)$ has a locktime t. The $\mathsf{safeGuard}$ transaction is the following:

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

We note here that transaction $\mathsf{safeGuard}(y)$ is based on the timed commitment over Bitcoin by Andrychowicz et al. [2], with an important difference: the committed value x (where $y = \mathsf{H}(x)$) is chosen by the verifier (client) and not by the committer (server). The server just uses y.

2. $\mathsf{retBtcs}$: Once the server gets a hold of value x, it can post the following transaction to redeem $\mathsf{safeGuard}$ and retrieve his A bitcoins.

Prev :	safeGuard
InputsToPrev :	$[\mathsf{retBtcs}], \mathsf{sig}_S([\mathsf{retBtcs}]), \bot, x$
Conditions :	$\underline{body, \sigma :}$ $\mathsf{ver}_S(body, \sigma)$
Amount :	A ฿
Locktime :	0

3. safeGuard can also be redeemed by $\mathsf{fuse}(t)$, as mentioned before:

Prev :	safeGuard
InputsToPrev :	$[\mathsf{fuse}], \mathsf{sig}_S([\mathsf{fuse}]), \mathsf{sig}_C([\mathsf{fuse}]), \bot$
Conditions :	$\underline{body, \sigma :}$ $\mathsf{ver}_C(body, \sigma)$
Amount :	A ฿
Locktime :	t

Protocol Details. We now describe our protocol in detail, as depicted in Fig. 4. Let S denote the server and C the client. For each step $i = 1, \ldots, 10$, there is a deadline, t_i, to complete the step, where timelock t of the $\mathsf{fuse}(t)$ transaction is $>> t_{10}$. We recall that, as mentioned in the beginning of this section, all messages exchanged between the client and the server are recorded on the blockchain, creating the history *hist*.

- **Step 1:** C picks a random secret x and sends the following items to S: (i) a hash $hash = \mathsf{H}(\mathbf{T}_B)$ of the IBF of the original blocks he stores; (ii) an encryption $\mathsf{Enc}_\mathsf{P}(x)$ of x under BA's public key, P; (iii) a cryptographic hash of x, $y = \mathsf{H}(x)$; and (iv) 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 or is not sent within time t_1, S aborts the protocol.
- **Step 2:** S posts bitcoin transaction $\mathsf{safeGuard}(y)$ for A bitcoins. It also sends to the client a signature of the $\mathsf{Fuse}(t)$ transaction, $\sigma_S = \mathsf{sig}_S([\mathsf{fuse}])$. If this transaction is not posted within time t_2 or the signature is not valid or is not sent within time t_2, C aborts the protocol.
- **Step 3:** S and C run the AS protocol from the previous section. S returns proof $\mathcal{V} = \{\mathtt{T}, S, \mathbf{T}_K\}$ and blocks $b'_1, b'_2, \ldots, b'_\delta$ for the current blocks he stores, in the position of the original blocks $b_1, b_2, \ldots, b_\delta$ that he lost. The client C computes now the damage d using CheckProof. If CheckProof 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 t_4 or ZKP_2 fails to verify, S jumps to Step 6. We note here that ZKP_2 is for the statement $(hash, \mathcal{V}, b'_1, b'_2, \ldots, b'_\delta, d, \mathsf{chal})$: $\exists$ *secret* $\mathbf{T}_B$ *such that*

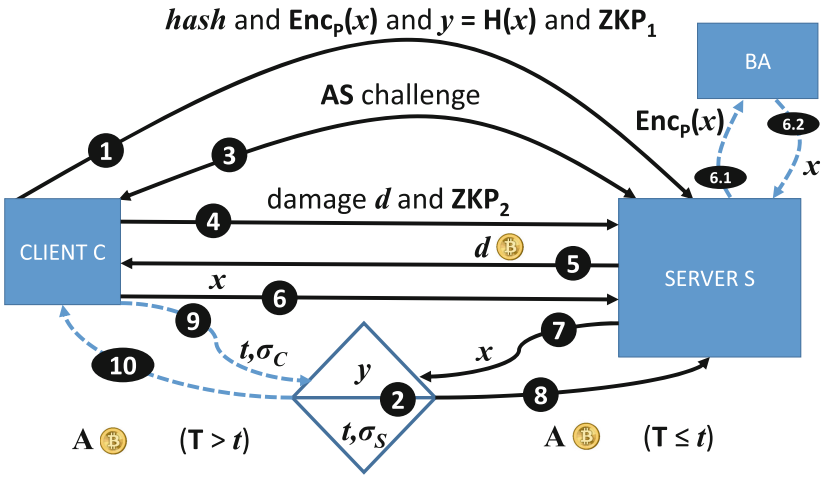


Fig. 4. Integration of the AS protocol with Bitcoin. The dotted lines indicate what happens when a party is trying to cheat. The rhomboid indicates the $\mathsf{safeGuard}(y)$ transaction (at the bottom of the rhomboid we show the server signature σ_S on the $\mathsf{fuse}(t)$ transaction) that is redeemed either by $\mathsf{retBtcs}$ (arrow 7) or by fuse (arrow 9), depending on the flow of the protocol.

$$hash = \mathsf{H}(\mathbf{T}_B) \,\&\, \{b_i\}_{i=1}^{\delta} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathbf{T}_B, \mathcal{V}, \mathsf{chal}) \,\&\, d = \sum_{i=1}^{\delta} ||b_i \oplus b_i'||.$$

The zero-knowledge proof ZKP_2 is needed here since the recovery algorithm takes as input the sensitive state of the client $\mathbf{T}_B$, which we want to hide from the server—otherwise the server can recover the original blocks himself and claim that there was no damage.

- **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 asks the BA to examine the history $hist$ up to that moment. BA checks $hist$ and if it is valid, BA sends x to S.
- **Step 7:** If S has secret x, S posts transaction $\mathsf{retBtcs}$.
- **Step 8:** If transaction $\mathsf{retBtcs}$ is valid, S receives A bitcoins before timelock t.
- **Step 9:** C waits until time t, computes $\sigma_C = \mathsf{sig}_S([\mathsf{fuse}])$ and posts transaction $\mathsf{fuse}(t)$ using σ_C and σ_S.
- **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 in the beginning of this section is achieved. We emphasize that BA can determine whether C properly followed the protocol by analyzing $hist$. If C has not done so and S reports it, BA will reveal x to S at any point in time. 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.

Global safeGuard. The protocol above protects the client at each AS challenge. But the cloud provider could stop interacting, simply disappear, and never be reachable by the client. Instead of the client aborting, we can use 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 the server cannot be reached at all or refuses to collaborate, but creates a scalability problem given that the server has to escrow a large amount of bitcoins for every client/customer. We do not address this problem technically but we expect it can be mitigated through financial mechanisms (securities, commodities, credit, etc.) typically deployed for traditional escrow accounts.

Removing the Bitcoin Arbitrator. Even though BA is only involved in case of disputes, it is preferable to remove it completely. Unfortunately, this seems impossible to achieve *efficiently* given the limitations of the Bitcoin scripting language. We sketch in this section two possible approaches to remove the BA. These will be further explored in future work.

The first approach relies on a secure two-party computation protocol. In a secure two-party computation 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 [31] 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 or more efficiently via the *cut-and-choose* method [20] or LEGO and MiniLEGO [14,24] (other efficient solutions were proposed in [18,30]).

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 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 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.

Another promising approach to remove the BA is to adopt smart contracts. Smart contracts are digital contracts that run through a blockchain. Ethereum [1] is a new cryptocurrency system that provides a Turing-complete language to write such contracts, which is expected to enable several decentralized applications without trusted entities. Smart contracts will enable our protocol to be fully automated without any arbitrators or trusted parties in between. To use a smart contract to run our protocol, the contract is expected to receive a

deposit from the server, inputs from both parties, and then it will decide the money flow accordingly based on the CheckProof result. The contract in that case will maintain some properties to ensure fair execution, for example both parties should be incentivized to follow the protocol, and if any party does not follow the protocol (by aborting for example), there should be a mechanism to end the protocol properly for the honest party. To make our protocol fit in the smart contract model, we will need to address the fact that the CheckProof computation would be too expensive to be performed by the contract, due to its overhead. In Ethereum for example, the participants must pay for the cost of running the contract, which is run by the miners.

In order to address the points above, zero knowledge SNARKs [6] could be employed to help reduce miners' overhead, and thus reduce the computational cost of the verification algorithm running on the network, while preserving the secrecy of the inputs.

7 Evaluation

We prototyped the proposed Accountable Storage (AS) scheme in Python 2.7.5. Our implementation is open-source[6] and consists of 4 K 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 perform 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 (including recover process). The second produces proof every time it receives a challenge. Both modules utilize the IBF 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.

Finally note that since it can be easily estimated, we have not evaluated the Bitcoin part of our protocol which is dominated by the time it takes for transactions to be part of the blockchain. Nowadays this latency is approximately 10 min.

Experimental Setup: Our experimental setup involves two nodes, one implementing the server and another implementing the client functionality. The two

[6] https://github.com/vlekakis/delta-AccountableStorage.

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 and the different sizes of blocks used are 1 KB, 2 KB, 4 KB and 8 KB. The total filesystem size varies from 100 KB to 4.1 GB. 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. For some of the large filesystems consisting of 100000 and 500000 number of blocks, we have estimated the results based on the experiments in lower filesystems due to computational power limitations.

In our experiments, we select the tolerance parameter δ, which indicates the maximum amount of data blocks that can be lost, to be equal to $\log_2(n)$. One other possible choice of δ is to set it equal to $\sqrt{n}$. We select the logarithm of the number of blocks as δ, because this provides a harder condition on how many blocks can be lost or corrupted from the cloud server.

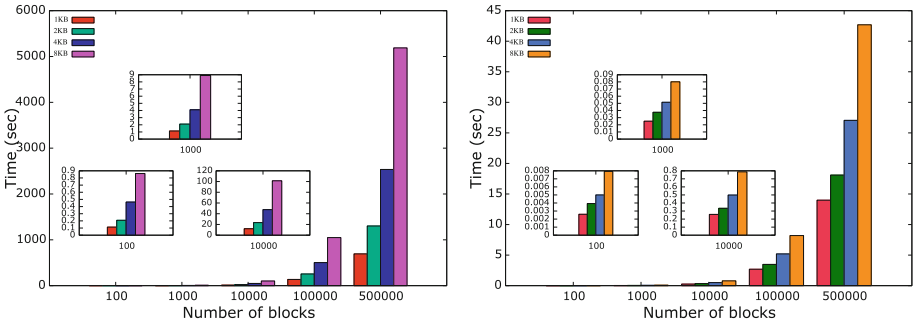
For the IBF construction, we have used the blocks and their generated tags. The selected number of hash functions used for the IBF construction is $k = 6$. This choice of hash functions leads to a very low probability of failure of the recovery algorithm, which depends on the values of k and δ.

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 estimated total size of the tags needed. The preprocessing memory overhead is proportional to the number of blocks in a filesystem.

Figure 5 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 IBF T_B. The tag generation time (Fig. 5a) increases linearly with both the available number of blocks and the size of each block. While this cost is significant for large file systems, it is an operation that client performs only once at the setup phase. On the other hand, the cost of construction of the IBF (Fig. 5b) is estimated to be negligible; the IBF construction of our biggest filesystem is estimated to take around 42 s.

Table 1. Memory footprint of the AS scheme (KB)

n	Tag size (KB)	Proof size (KB)			
		$1KB$	$2KB$	4 KB	8 KB
10^2	32	353	692	1369	2722
10^3	236	528	1035	2048	4073
10^4	2644	762	1493	2593	5875
10^5	24895	937	1898	3526	7226
$5 * 10^5$	118326	1077	2182	4054	8308



(a) Tag Generation (b) Client IBF Generation

Fig. 5. Preprocessing overheads

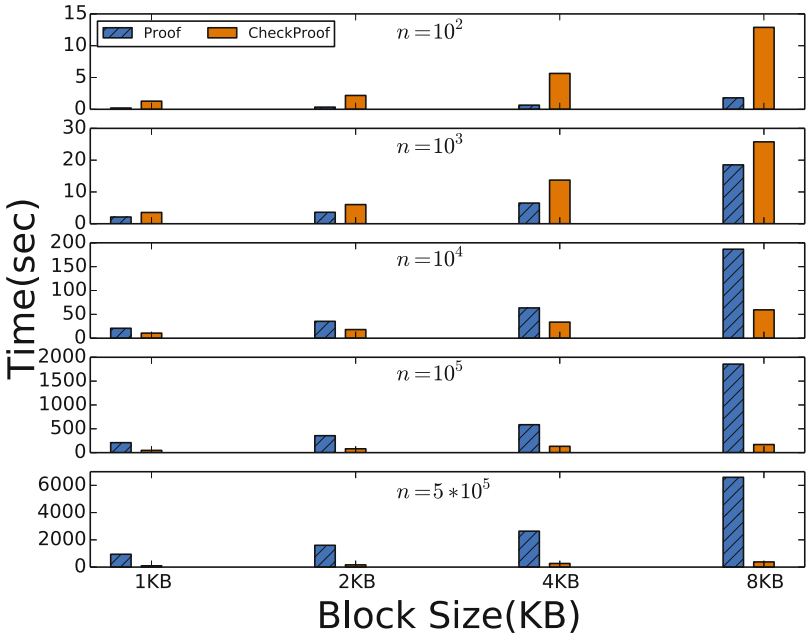


Fig. 6. Proof generation and proof check (including recover) time

Challenge-Proof Overheads: We now examine memory and CPU-related overheads for the challenge-proof exchange and the recovery phase. The last four columns of Table 1 show the proof sizes (in KB) for $\delta = \log_2(n)$, which increase proportionally to the block size.

Every subgraph of Fig. 6 shows how different block sizes affect the performance of the challenge-proof exchange for a given number of blocks. The left bar in the figure shows the proof generation time and the right bar the proof check

along that includes the time recover the lost blocks. We notice that larger block sizes are estimated to increase the time-overhead of challenge-proof exchange. We also notice that the proof check and in particular the recover process introduces higher time overhead in comparison to proof generation for small size filesystems (100 and 1000 number of blocks). This is expected, because the tag verification (publicly verifiable) in the recover process (in Fig. 2) introduces significant time overhead compared to proof generation (in Fig. 3) for a small number of blocks. However, in higher size filesystems, we observe from Fig. 6 that the time overhead of proof generation increases exponentially and overcomes the proof check time (including recover process) overhead. This is also expected, because the number of blocks (denoted by Kept set) used in the proof generation process is much higher than the number of blocks used in the recover process.

8 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.

Acknowledgments. Research supported in part by an NSF CAREER award CNS-1652259, NSF grants CNS-1525044, CNS-1526950, CNS-1228639 and CNS-1526631, a NIST award and by the Defense Advanced Research Projects Agency (DARPA) under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

Appendix

RSA Assumption

Definition 4. *Let $N = pq$ be an RSA modulus, where p and q are τ-bit primes. Given N, e and g, where g is randomly chosen from $\mathbb{Z}_N^*$ and e is a prime of $\Theta(\tau)$ bits, there is no PPT algorithm that can output $y^{1/e} \mod N$, except with probability $\mathsf{neg}(\tau)$.*

Proof of Security of Construction in Fig. 3.

Setup. $\mathcal{A}$ chooses parameter $\delta \in [0, n)$, blocks $B = \{b_1, b_2, \ldots, b_n\}$ and is given pk and $\mathsf{T}_1, \ldots, \mathsf{T}_n$ as output by $\{\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathsf{T}_1, \ldots, \mathsf{T}_n\} \leftarrow \mathsf{Setup}(b_1, \ldots, b_n, \delta, 1^\tau)$. The random oracle is programmed so that it returns $r_i^e g^{-b_i}$ on input i for some random r_i, i.e., $h(i) = r_i^e g^{-b_i} \mod N$.

Forge. $\mathcal{A}$ is given $\mathsf{chal} \leftarrow \mathsf{GenChal}(1^\tau)$, computes proof of accountability $\mathcal{V}$ and returns $\mathcal{V}$. Suppose $\mathcal{L} \leftarrow \mathsf{CheckProof}(\mathsf{pk}, \mathsf{sk}, \mathsf{state}, \mathcal{V}, \mathsf{chal})$. We must show that with probability $\geq 1 - \mathsf{neg}(\tau)$ it is (i) $\mathcal{L} \subseteq B$; and (ii) there exists a PPT knowledge extractor $\mathcal{E}$ that can extract *all the remaining file blocks* in $B - \mathcal{L}$.

1. **Showing** $\mathcal{L} \subseteq B$. Note that all blocks in $\mathcal{L}$ are output by Algorithm recover of Fig. 2. In this algorithm a block b'_i can enter $\mathcal{L}$ only if its tag verifies. Suppose now $b'_i \notin B$ (namely $b'_i \neq b_i$) and $\mathtt{tag}^e/h(i) = g^{b'_i}$ for some arbitrary $\mathtt{tag}$ computed by the adversary. But since $h(i) = r_i^e g^{-b_i}$, this can be written as $\mathtt{tag}^e/r_i^e g^{-b_i} = g^{b'_i}$ which gives $g^{b'_i - b_i} = (\mathtt{tag}/r_i)^e$. Since e is a prime and $b_i - b'_i \neq 0$, there exist α and β such that $(b'_i - b_i) \times \alpha + e \times \beta = 1$, giving $g^{1/e} = g^{-\beta}(tag/r_i)^{e \times \alpha}$, breaking the RSA assumption—see Definition 4.
2. **Showing there exists a PPT knowledge extractor $\mathcal{E}$ that can extract *all the remaining file blocks* in $B - \mathcal{L}$.** We now show how to build an extractor that, after $\ell = |B - \mathcal{L}|$ interactions with the adversary, he can extract the blocks $\{b_i : i \in B - \mathcal{L}\}$. The extractor will challenge the adversary exactly ℓ times, each time with different randomness. Let $S_1, S_2, \ldots, S_\ell, \mathtt{T}^{(1)}, \mathtt{T}^{(2)}, \ldots, \mathtt{T}^{(\ell)}$ be the sums and tags he receives by $\mathcal{A}$ during each challenge, as in Eq. (2). We have two cases:
   (a) $S_j = \sum_{i \in B - \mathcal{L}} a_{ij} b_i$, for $j \in B - \mathcal{L}$ (a_{ij} denotes the randomness of the j-th challenge corresponding to the i-th block). In this case, the extractor can solve a system of ℓ linear equations and retrieve the original blocks $\{b_i : i \in B - \mathcal{L}\}$.
   (b) Suppose there exists $j \in B - \mathcal{L}$ such that $S_j \neq \sum_{i \in B - \mathcal{L}} a_{ij} b_i = S$. For simplicity of notation, let's set $\mathtt{T}^{(j)} = \mathtt{T}$ and $S_j = \bar{S}$. Then by the CheckProof algorithm we have

$$\frac{\mathtt{T}^e}{\prod_{i \in B - \mathcal{L}} h(i)^{a_i}} = g^{\bar{S}}.$$

   But since $h(i) = r_i^e g^{-b_i}$ we have that

$$g^{\bar{S} - S} = \left(\frac{\mathtt{T}}{\prod_{i \in B - \mathcal{L}} r_i^{a_i}} \right)^e = \mathcal{Z}^e.$$

   Therefore we have $\mathcal{Z}^e = g^{\bar{S} - S}$. Again, since e is prime and $S \neq \bar{S}$ we can use the same trick as before, and break the RSA assumption. □

References

1. Ethereum: A platform for decentralized applications. www.ethereum.org/
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multi-party computations on bitcoin. In: IEEE SSP (2014)
3. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: ACM CCS (2007)
4. Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: SecureComm (2008)

5. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993). doi:10.1007/3-540-48071-4_28
6. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40084-1_6
7. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Comm. ACM **13**, 422–426 (1970)
8. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 236–254. Springer, Heidelberg (2000). doi:10.1007/3-540-44598-6_15
9. Carter, I.L., Wegman, M.N.: Universal classes of hash functions. In: ACM STOC (1977)
10. Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious RAM. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 279–295. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38348-9_17
11. Curtmola, R., Khan, O., Burns, R.C., Ateniese, G.: MR-PDP: multiple-replica provable data possession. In: ICDCS (2008)
12. Eppstein, D., Goodrich, M.T., Uyeda, F., Varghese, G.: What's the difference? Efficient set reconciliation without prior context. In: SIGCOMM (2011)
13. Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: ACM CCS (2009)
14. Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B., Nordholt, P.S., Orlandi, C.: MiniLEGO: efficient secure two-party computation from general assumptions. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 537–556. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38348-9_32
15. Goodrich, M.T., Mitzenmacher, M.: Invertible Bloom Lookup Tables. ArXiv e-prints, January 2011
16. Google. Google protocol buffers. www.developers.google.com/protocol-buffers/
17. Van Horsen, C.: Gmpy2: Mupltiple-precision arithmetic for python. www.gmpy2.readthedocs.org/en/latest/intro.html/
18. Jarecki, S., Shmatikov, V.: Efficient two-party secure computation on committed inputs. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 97–114. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72540-4_6
19. Juels, A., Kaliski Jr. B.S.: PORs: proofs of retrievability for large files. In: ACM CCS (2007)
20. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72540-4_4
21. Litzenberger, D.C.: Pycrypto - the python cryptography toolkit. www.dlitz.net/software/pycrypto/
22. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Fair two-party computations via the bitcoin deposits. In: FC (2014)
23. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. www.bitcoin.org/bitcoin.pdf
24. Nielsen, J.B., Orlandi, C.: LEGO for two-party secure computation. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 368–386. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00457-5_22

25. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: IEEE SSP (2013)
26. Shacham, H., Waters, B.: Compact proofs of retrievability. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 90–107. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89255-7_7
27. Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: ACM CCS (2013)
28. Stefanov, E., van Dijk, M., Oprea, A., Juels, A.: Iris: a scalable cloud file system with efficient integrity checks. In: ACSAC (2012)
29. Wang, Q., Wang, C., Li, J., Ren, K., Lou, W.: Enabling public verifiability and data dynamics for storage security in cloud computing. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 355–370. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04444-1_22
30. Woodruff, D.P.: Revisiting the efficiency of malicious two-party computation. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 79–96. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72540-4_5
31. Yao, A.C.-C.: How to generate and exchange secrets. In: SFCS (1986)