Proofs of Ownership in Remote Storage Systems *

Shai Halevi[1], Danny Harnik[2], Benny Pinkas[3], and Alexandra Shulman-Peleg[2]

[1]IBM T. J. Watson Research Center, [2]IBM Haifa Research Lab, [3]Bar Ilan University

Abstract

Cloud storage systems are becoming increasingly popular. A promising technology that keeps their cost down is *deduplication*, which stores only a single copy of repeating data. *Client-side deduplication* attempts to identify deduplication opportunities already at the client and save the bandwidth of uploading copies of existing files to the server.

In this work we identify attacks that exploit client-side deduplication, allowing an attacker to gain access to arbitrary-size files of other users based on a very small hash signatures of these files. More specifically, an attacker who knows the hash signature of a file can convince the storage service that it owns that file, hence the server lets the attacker download the entire file. (In parallel to our work, a subset of these attacks were recently introduced in the wild with respect to the Dropbox file synchronization service.)

To overcome such attacks, we introduce the notion of proofs-of-ownership (PoWs), which lets a client efficiently prove to a server that that the client holds a file, rather than just some short information about it. We formalize the concept of proof-of-ownership, under rigorous security definitions, and rigorous efficiency requirements of Petabyte scale storage systems. We then present solutions based on Merkle trees and specific encodings, and analyze their security. We implemented one variant of the scheme. Our performance measurements indicate that the scheme incurs only a small overhead compared to naive client-side deduplication.

Keywords: Cloud storage, Deduplication, Merkle trees, Proofs of ownership.

1. Introduction

Cloud computing provides a low-cost, scalable, location-independent infrastructure for data management and storage. The rapid adoption of Cloud services is accompanied by increasing volumes of data stored at remote servers, so techniques for saving disk space and network bandwidth are needed. A central up and coming concept in this context is *deduplication*, where the server stores only a single copy of each file, regardless of how many clients asked to store that file. All clients that store the file merely use links to the single copy of the file stored at the server. Moreover, if the server already has a copy of the file, then clients do not even need to upload it again to the server, thus saving bandwidth as well as storage (this is termed client-side deduplication). Reportedly, business applications can achieve deduplication ratios from 1:10 to as much as 1:500, resulting in disk and bandwidth savings of more 90% [8]. Deduplication can be applied at the file level or at the block level, and this paper focuses on file-level deduplication. (Block-level deduplication is discussed briefly in Section 6.)

In a typical storage system with deduplication, a client first sends to the server only a hash of the file and the server checks if that hash value already exists in its database. If the hash is not in the database then the server asks for the entire file. Otherwise, since the file already exists at the server (potentially uploaded by someone else), it tells the client that there is no need to send the file itself. Either way the server marks the client as an owner of that file, and from that point on there is no difference between the client and the original party who has uploaded the file. The client can therefore ask to restore the file, regardless of whether he was asked to upload the file or not.

Harnik et al. observed recently that client-side deduplication introduces new security problems [13]. For example, a server telling a client that it need not send the file reveals that some other client has the exact same file, which could be a sensitive information. The findings in [13] apply to popular file storage services such as MozyHome and Dropbox, among others.

A New Attack. The starting point of this work is a much more direct threat to storage systems that use client-side deduplication across multiple users. Specifically, by accepting the hash value as a "proxy" for the entire file, the server allows anyone who gets the hash value to get the entire file. Consider for example the following case: Bob is a researcher who routinely publishes hashes of his lab reports, as a time-stamp for the results of his research. Bob is also careful to daily backup his hard disk to an online file storage service. Alice is interested in learning the results of Bob's research. She signs up to the same storage service, asks to backup a file, and when asked to present its hash value she presents the value that was published by Bob as a time-stamp for his file. The service forgoes the upload, and just makes note that Alice and Bob uploaded the same file. Later on, Alice asks to recover that file and the service complies, sending her a copy of Bob's lab report.

The problem illustrated above stems from the fact that by learning just a small piece of information about the file, namely its hash value, an attacker is able to get the entire file from the server. This extra piece of information is not really meant to be secret, and in fact it must be computable from the file via a deterministic public algorithm which is shared by all clients.

A simplistic fix for this attack is to hash some context information together with the file, e.g., ``deduplication at cloud service XYZ'' and a system-wide random salt value (i.e., a single random value used for all files). This makes it highly unlikely for different applications to use the same hashing mechanism, thus preventing the specific attack from above. We note, however, that

* The work of Benny Pinkas was supported by the SFEROT project funded by the European Research Council (ERC). The work of Danny Harnik and Alexandra Shulman-Peleg has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n 257019.

this solution is still unsatisfactory, in that it does not address the root cause of the attack: There is still a very short piece of information that represents the file, and an attacker that learns that piece of information can get access to the entire file. Although we plugged one avenue by which the attacker was able to get this piece of information, there could be others.

Potential Attacks

Below we list some plausible attack scenarios, supporting our main point that representing a large file by a small piece of "not really secret" information is a bad idea. Some of the attacks were demonstrated against popular storage services.

Usage of a common hash function. This is essentially the attack described in the previous example, where the deduplication system uses a standard hash function in a straightforward manner. (It was recently reported that Dropbox uses SHA256 for this purpose [1].) In that case it is plausible that an attacker can get hold of the hash values of files owned by other users (e.g., if these hash values are used elsewhere or the user publishes a signature on the file).

Using the storage service as an unintended CDN. In this scenario, Alice the attacker uploads a copy of a (potentially huge, and possibly copyright infringing) file to the storage server and publishes the hash of that file (e.g., on her web page). Now anyone wishing to obtain the file can attempt to upload it to the storage service, present the hash value to the service and be identified as owning that file, and then ask to restore the file from the storage service. Hence Alice essentially uses the storage service as a *content distribution network* (CDN). This behavior may run afoul with the business model of that storage server, which is likely to be designed to support many uploads but very few restore operations. This behavior might also support piracy and copyright infringing behavior.

Server break-in. Consider an attacker that is able to temporarily compromise a server machine, getting access to its internal cache, which includes the hash values for all the recently accessed files. Having obtained this (relatively small) piece of information, the attacker is now able to download all these files, which may include confidential files of others. The attacker can even publish these hash values, thus enabling anyone in the world to get these files.

Note that it is unlikely that the attacker can use a short compromise event to actually download all the large files themselves, but the server cache may be many orders of magnitude smaller. Further, note that even if the server learns about the attack early on, there is not much that it can do about it. The only effective remedy that it has is to turn off client side deduplication for the affected files, essentially "forever."

Malicious client software. A similar attack to the above attack is also possible on the client side. For example, Alice the attacker can install malicious software on the machine of Bob the victim. The malicious software can use a very low-bandwidth covert channel to send to Alice the hash of interesting files, thus enabling Alice to get the file itself from the server. (In fact, in [13] it was shown how to use the deduplication service itself as a low bandwidth covert channel. The attack described here can amplify this covert channel and enable the leakage of much larger amounts of data.)

As opposed to the malicious software stealing Bob's access password to the storage server, the hash-based attack works no matter how well Bob protects his credentials (e.g., even if Bob uses an external device or some TPM-based scheme to authenticate to the server). Further, the usage of a stolen password can be identified by Bob checking his account login audit files. A hash based attack cannot be easily tracked since the attacker pose as legitimate users who happen to upload the same files as Bob.

In one variant of this attack, the software temporarily copies a file on Bob's machine to a directory whose contents are routinely backed up online, wait for a backup operation, then delete the file from that directory. Online storage services typically keeps deleted files for a at least 30 days, giving Alice ample opportunity to recover the file with very little chance for Bob to trace how the leak happened.

The attack can be amplified even further in the following way. The examination of Dorredorf and Pinkas [7] revealed that the client software of several file storage services, including Dropbox, stores local unencrypted manifest files, written in SQLite format. These files include an entry for each file stored by the service, which includes the hash value of the file. A malicious software can leak to Alice the hash value of this file itself. Alice can use this value to recover the manifest file, and then use the hash values stored in the manifest file to recover all files stored by Bob. The implication is that a leakage of a short hash value, which is either 20 or 32 byte long (depending on the service), leaks all of Bob's files, in a way which is essentially undetectable.

Accidental leakage. Even without malicious software, when a short value is used to represent a big file there always exists a risk of it leaking (e.g., by swapping the value into disk via storage-area network, storing that value in a "control file" that may be left behind in a directory, etc.) If many clients have the same file (e.g., a department-wide document), it may become quite likely that one of them will accidentally leak the short summary of that document, thus allowing unauthorized access to it via the storage service.

Implementations of the attack

We stress that the attacks that we describe are not just theoretical. Independently of our work, the attack was very recently discovered by Wladimir van der Laan, who started an open-source project called Dropship that uses the Dropbox storage servers as a CDN. The Dropship project was withdrawn after Dropbox had asked "in a really civil way" that its creator take down the source code, since they "felt that it is a violation of the TOS (Terms of Service)" [1, 18].[1] This attack against Dropbox, as well as a similar attack that was implemented against another major file storage service, was also discovered and implemented by Dorrendorf and Pinkas [7]. (The name of the second service is withheld since the relevant company has not yet been notified.). Another recent and independent implementation of the attack on Dropbox was described by Mulazzani et al. [16].

Effect on Open APIs for Remote Storage. Given the many avenues, described above, for the attacker to get hold of the hash values of files, it is clear that using only a small hash as a proxy for a large file is a security vulnerability in systems that use client-side deduplication. This vulnerability must be addressed before we can attempt to create open APIs for remote-storage that support client-side deduplication, since such APIs have to be usable in environments that are sensitive to some of the attacks above. We thus view a solution to this problem as an enabler for the creation of advanced open storage APIs.

1.1 Proofs of Ownership (PoWs)

To solve the problem of using a small hash value as a proxy for the entire file, we want to design a solution where a client proves to the server that it indeed has the file. We call a proof mechanism that

[1] It is interesting to note that even though the Dropbox service allows its users to share files by storing them on their public folder, Dropbox has actively tried to shutdown the Dropship service. This might be due to the fact that normal file sharing through the public folder is bandwidth limited, and since Dropship is better suited for sharing copyright infringing content.

prevents such leakage amplification a *proof of ownership* (PoW). We note that this is somewhat similar to proofs of retrievability (PORs) [14, 17] and proofs of data possession (PDPs) [3] with a role reversal (the client is the prover rather than the server). However, this role reversal between client and server is significant, and in particular the advanced POR and PDP protocols in the literature are inapplicable in our setting, see more discussion in Section 1.3.

Some important requirements that constrain the solution in our setting are discussed next:

Public hash function. To support cross-user deduplication, all clients must use the same procedure for identifying duplicate files. Hence this procedure must be **public** (since it is implemented in every client), which means that a determined attacker can learn it.

Bandwidth constraints. The protocol run between the server and client must be **bandwidth efficient**. Specifically, it must consume a lot less bandwidth than the size of the file (otherwise the client could just send the file itself to the server).

Server constraints. The server typically has to handle a huge number of files. (For example, Mozy reports storing more than 25 Petabytes of data.) The files themselves are stored on a secondary storage with a large access time, and the server can store only a small amount of data per file in fast storage (such as RAM). Moreover, the server cannot afford to retrieve the file or parts of it from secondary storage upon every upload request. The solution must therefore allow the server to store only an extremely **short information per file**, that will enable it to check claims from clients that they have that file, without having to fetch the file contents for verification.

Client constraints. The client should be able to efficiently prove to the server that it knows the file in question. This could be hard, e.g., if the file is very large and does not fit in local memory. We therefore seek a solution where the client can compute the proof by making a **single pass** over the file (and using a reasonable amount of memory).

On the other hand, during the computation of the proof there should be **no very-short state** from which the proof can be computed. Otherwise the very-short state itself can serve as a proxy for the large file, and the system will remain vulnerable to some of the atacks above.

1.1.1 Our security definition

For security we want to ensure that as long as the file has a lot of min-entropy (from the attacker's perspective), the attacker has only a small chance of convincing the server. We stress that in many cases, an attacker may have some **partial information** about the file, e.g., it may know the format of the file, its language, or the contents of large portions in the file. Hence we require security even when the entropy of file (from the attacker's perspective) is much smaller than the file length, as long as there are still a lot of uncertainty left in the file.[2]

Defining what it means for the file to have a lot of min-entropy from the attacker's perspective turns out to be somewhat subtle. The formal definition roughly follows the CDN/malicious-software attack scenarios. Specifically, we consider an attacker that does not know all of the file, but has accomplices who have the file. The accomplices can interact with the storage server, and can help the attacker in arbitrary ways, subject to only two constraints. One constraint is that the total number of bits that the accomplices send to the attacker is much smaller than the initial min-entropy of the file (hence the file still has a lot of min-entropy even after all that communication). The other constraint is that the accomplices can only help the attacker during an off-line phase before the proof protocol begins, since otherwise the attacker could just relay messages back and forth between the server and an accomplice who has the file. (The no-man-in-the-middle restriction seems justified in many of the attack scenarios from above.) This formulation can be viewed as an instance of the "bounded retrieval model" [6, 9].

[2] Note that the security also holds for computational notions of entropy, e.g., even if the adversary holds an encrypted version of the file.

1.2 Our Solutions

In this work we consider a few different formal formulations of the intuitive requirement above and associated protocols that realize them. The first solution is the most stringent in terms of security, but lacks in efficiency (mainly of computation time and memory requirements). The next solutions two solutions each improve on these aspects at the cost of relaxations on the security guarantees.

A general solution. Our most stringent security requirement says that as long as the min-entropy in the original file *minus the number of bits sent to the adversary by the accomplices* is more than the security parameter, the adversary should not be able to convince the server that it has the file (except with insignificant probability). This notion can be achieved by adopting the Merkle-tree-based proof-of-retrievability protocol. Namely, we first encode the file using an erasure code, such that it is possible to recover the whole file from any (say) 90% of the encoded bits. We then build a Merkle tree over the encoded file, and have the server ask for a super-logarithmic number of leaves, chosen at random.

From the properties of the erasure code, if the adversary is missing any of the file then there are at least 10% of the leaves that it does not know. Moreover, if the file has high min-entropy from the adversary's perspective then it cannot even guess the value of these 10% leaves with any noticeable chance for success. Hence, it will be caught with overwhelming probability.[3] This solution is described in Section 3.

A more efficient solution using universal hashing. The above solution is not as efficient for the client as we would like. First, computing the erasure code requires random access to bits of the file, hence for large files that do not fit in RAM it entails too many disk accesses. Also, the server must ask for a number of leaves which is super-logarithmic in the security parameter, which means that the communication complexity of the proof protocol is $\omega(k \log k \log n)$ for an n-leaf tree and security parameter k.

We therefore seek more efficient protocols, even at the price of settling for a somewhat weaker form of security. Specifically, we relax our security requirement by having an upper-threshold T such that if the attacker gets T bits from its accomplices then it is allowed to convince the prover (even if T is much smaller than the entropy of the file). This allows us to have a mechanism in which the client only needs $O(T)$ bits of internal memory in order to process the file, even if the file itself is much larger, and the communication in the protocol depends (polylogarithmically) on T and not on the size of the file. In realistic scenarios, we may get adequate security by setting T to be a rather large value (but not huge), e.g., 64 MByte. With this relaxation, a good solution would be to use universal hashing to hash the potentially large file into a *reduction buffer* of size at most 64MByte, and then implement the Merkle-tree protocol on the hashed values. This solution is described in Section 4.

[3] Without erasure codes, an adversary could have very high success probability on large files with small entropy, since the small entropy might mean that the adversary knows most of the file. The codes ensures that even a single unknown block of a large file will be extended to many unknown blocks in the encoded file.

A streaming scheme and implementation The more efficient solution can be implemented with reasonable memory, but universal hashing with such large output size is very expensive to compute. We, thus, optimize for a more practical solution for which we can still offer meaningful security analysis, albeit under some restrictions. We may argue that in realistic settings the input distribution (from the attacker's perspective) is not arbitrary, but rather is taken from a specific class of distributions. Specifically, these distributions capture cases in which there are parts of the file that the attacker may know and other parts that are essentially random. We formulate this condition (which is a generalization of "block-fixing" distributions), and describe a protocol that can be proved secure for this class of input distributions. Roughly speaking, for these common input distributions we can replace the universal hashing with a procedure that generates a good erasure code over the reduction buffer.

We stress that it is impossible to verify experimentally the assumption about the input distribution, since it is an assumption about the view of an attacker. Still it seems a reasonable assumption to make in many settings (e.g., even files in a known format contain some file specific information that is essentially random, see examples in Section 2.2.1). We also stress that we do not know of any efficiently computable input distributions under which there is an attack on our streaming scheme. It is interesting to either prove that no such distributions exist (maybe in the random-oracle model) or to exhibit one.

We implemented this streaming scheme and measured its expected effect on performance of the system (see details of the implementation and test results in Section 5.3). For large files (over 2GByte) the overhead of our protocol above reading the file and computing its SHA256, is less than 50%, and this figure drops as the file grows. The overhead for smaller files is larger, yet it is still very beneficially to run the protocol when considering the saved data transfer times. When considering an average 5Mbps network, the protocol takes no more than 4% of the time it would have taken to transfer the file via the network (and less than 1% for large files). Even with a fast 100Mbps network, client-side deduplication with our protocol beats sending the entire file to the server already for files as small as 125KByte.

1.3 Related Work

As we mentioned above, proofs-of-ownership are closely related to proofs of retrievability (POR) [14, 17, 19] and proofs of data possession (PDP) [3]. The two main differences are that (a) proofs of retrievability/data-possession often use a pre-processing step that cannot be used in proofs of ownership, and (b) our security notion is weaker than that of proofs of retrievability. We now elaborate on these two points:

No pre-processing step. In PORs/PDPs the input file is pre-processed by the client by embedding some secrets in it, which makes it possible for the server to later prove that it has the file by replying to the client queries with answers that are consistent with these secrets. In our setting, a new client comes with the only original file itself, we cannot embed secrets in it for the purpose of proofs. This rules out solutions based on MACs or signatures, for example the schemes from [3, 4, 14, 17].

Weaker security guarantee. Our definition is not in the style of proofs-of-knowledge [11], in that we do not require extractions. Instead, we assume that the file is chosen from a distribution and talk about the success probability of the attacker given that distribution. In this sense, our definition is somewhat reminiscent of the Storage-enforcing Commitment of Golle et al. [12]. It is obvious that our security notion is implied by the stronger extraction property. It is also not hard to show an example of a protocol which is secure according to our notion but does not have an efficient extractor.[4]

Another well studied, related problem is that of verifying the integrity of memory contents in computing platforms and ensuring that no memory corruption affects the performed computations [10]. Many of these schemes utilize Merkle trees [15] for memory and data authentication.

2. Preliminaries

We briefly survey the tools that we use in our solutions and state our security definition. Below we denote by $[n]$ the set $\{1, 2, \ldots, n\}$.

Error correction codes. An $[n, k, d]$ code represents k-bit values using n-bit codewords which at at least d apart. We identify the code with an encoding function $E : \{0,1\}^k \to \{0,1\}^n$, and it has the property that for any two different values $x, y \in \{0,1\}^k$, the Hamming distance between $E(x)$ and $E(y)$ is at least d. The smallest distance between any two codewords is called the minimum-distance of the code. It is clear that a code with minimum-distance d can recover from any $d-1$ erasures.

Pairwise independent hashing. A family H of functions from $\{0,1\}^k$ to $\{0,1\}^n$ is pairwise independent (aka 2-universal) if for every two different values $x, y \in \{0,1\}^k$ and every two strings $a, b \in \{0,1\}^n$, when choosing at random $h \in_R H$ the probability that h maps x to a and y to b is exactly 2^{-2n}. An example of a pairwise independent family is the family of affine functions, $H = \{(x \mapsto Ax + b) : A \in \{0,1\}^{k \times n},\ b \in \{0,1\}^n\}$.

Collision resistant hashing. A collision-resistant hash function has shorter output than input, and yet it is hard to find efficiently two inputs that maps to the same output. Formally, a family of functions $\mathcal{H}$ is collision resistant if no efficient algorithm can find, on input a random $h \in \mathcal{H}$, two different inputs $x \neq y$ such that $h(x) = h(y)$ (except with insignificant probability). In practice we use a single function that accepts arbitrary-size input and has a fixed-size output (e.g., SHA256 with 256 bits of output).

2.1 Merkle Trees

A Merkle tree provides a succinct "commitment" to a large buffer, such that it is later possible to "open" and verify individual blocks of the buffer without giving the entire buffer. To construct a Merkle tree we split the input buffer into blocks, then group the blocks in pairs and use a collision-resistant hash function to hash each pair. The hash values are then again grouped in pairs and each pair is further hashed, and this process is repeated until only a single hash value remains. This results in a binary tree with the leaves corresponding to the blocks of the input buffer and the root corresponding to the last remaining hash value. (When the number of blocks in the input buffer is 2^h, the resulting tree is a height-h complete binary tree.)

We denote by $MT_{h,b}(X)$ the binary Merkle tree over buffer X using b-bit leaves and the hash function h. For each node in the tree $n \in MT_{h,b}(X)$, we denote by v_n the value associated with that node. That is, the value of a leaf is the corresponding block of X, and the value of an intermediate node $n \in MT_{h,b}(X)$ is the hash $v_n = h(v_l, v_r)$ where v_l, v_r are the values for the left- and right-children of n, respectively. (If one of the children of a node is missing from the tree then we consider its value to be the empty string.)

[4] For example, applying the solution of Section 3 to a one-way permutation of the file rather than to the file itself. One can extract the permuted file but cannot extract the file itself (still this adheres to the requirement of PoWs, since to answer correctly many bits have to be leaked).

For a leaf node $l \in MT_{h,b}(X)$, the *sibling path* of l consists of the value v_l and also the values of all the *siblings of nodes on the path from l to the root*. Given the index of a leaf $l \in MT_{h,b}(X)$ and a sibling path for l, we can compute the values of all the leaves on the l-root path itself in a bottom-up fashion by starting from the two leaves and then repeatedly computing the value of a parent as the hash of the two children values. (The order of the children in the hash computation is determined by the bits of the binary representation of the index of l.)

We say that an alleged sibling path $P = (v_l, v_{n_0}, v_{n_1}, \ldots, v_{n_i})$ is *valid with respect to* $MT_{h,b}(X)$ if i is indeed the height of the tree and the root value as computed on the sibling path agrees with the root value of $MT_{h,b}(X)$. Note that in order to verify that a given alleged sibling path is valid, it is sufficient to know the number of leaves and the root value of $MT_{h,b}(X)$. (We also note that any two different valid sibling paths with respect to the same Merkle tree imply in particular a collision in the hash function.)

The Merkle-tree proof protocol. All our solutions use the following basic proof protocol, between a verifier that has the root value of a Merkle tree $MT_{h,b}(X)$ (and the number of leaves in the tree), and a prover who claims to know the underlying buffer X. The verifier simply chooses some number of leaf indexes and ask the prover for the value of the corresponding leaves. The prover replies with these leaves and with a sibling path for each one of them, and the verifier accepts if all these sibling paths are valid.

Below we use $MTP_h(v, s, u)$ to denote the Merkle-tree protocol (with respect to hash function h) where the verifier knows the root value v and number of leaves s, and asks to see u leaves of the tree (with their sibling paths).

The Merkle-tree lemma. Our security proofs rely on the following well-known Merkle-tree lemma, which says that every prover that passes the Merkle-tree protocol with high enough probability can be converted into an extractor that extracts most of the leaves of the tree. We provide the proof in the appendix for self-containment.

Lemma 1 (Merkle-tree lemma)**.** *There exists a black-box extractor K with oracle access to a Merkle-tree-prover, that has the following properties:*

1. *For every prover P and $v \in \{0,1\}^*$, $s, u \in \mathbb{N}$, and $\delta \in [0,1]$, $K^P(v, s, u, \delta)$ makes at most $u^2 s(\log(s) + 1)/\delta$ calls to its prover oracle P.*
2. *Fix any hash function h and input buffer X with s leaves of b-bits each, and let v be the root value of $MT_{h,b}(X)$. Also fix some $u \in \mathbb{N}$ and a prover (that may depend on h, X and u) $P^* = P^*(h, X, u)$.*
   Then if P^* has probability at least $(1-\alpha)^u + \delta$ of convincing the verifier in the Merkle-tree protocol $MTP_h(v, s, u)$ (for some $\alpha, \delta \in (0, 1]$), then with probability at least $1/4$ (over its internal randomness) the extractor $K^{P^*}(v, s, u, \delta)$ outputs values for at least a $(1-\alpha)$-fraction of the leaves of the tree, together with valid sibling paths for all these leaves.

2.2 Proofs of Ownership

Proof-of-ownership is a protocol in two parts between two players on a joint input F (which is the input file). First the verifier summarizes to itself the input file F and generates a (shorter) verification information v. Later, the prover and verifier engage in an interactive protocol in which the prover has F and the verifier only has v, at the end of which the verifier either accepts or rejects. Hence a proof-of-ownership is specified by a summary function $S(\cdot)$ (which could be randomized and takes the input file F and a security parameter), and an interactive two-party protocol $\Pi(P \leftrightarrow V)$.

Validity. The scheme $\mathcal{P} = (S, \Pi)$ is valid if (a) S and Π are efficient, and (b) for every input file $F \in \{0,1\}^n$ and every value of the security parameter n, it holds that $\Pi(P(F, 1^n) \leftrightarrow V(S(F, 1^n))) \Rightarrow \mathsf{accept}$ with all but a negligible probability (in n).

Efficiency. The main efficiency parameters of a proof-of-ownership are (a) the size of the summary information $v = S(F, 1^n)$, (b) the communication complexity of the protocol Π, and (c) the computation complexity of computing the function S and of the two parties in Π (all with respect to the file size $|F|$ and the security parameter n). We seek solutions where the dependence on the security parameter is linear, and where the computation complexity of S and P is linear in $|F|$ and everything else is at most (poly) logarithmic in $|F|$.

Another efficiency parameter that can be important is the space complexity of S, P when viewed as one-pass algorithms that access the input file in a streaming fashion. (This is significant since for large files we would like to read them sequentially from disk only once, and we do not have enough memory to store them all.)

2.2.1 Security of Proofs-of-Ownership

As we explained in the introduction, our goal is to ensure that the file does not have a "small representation" that when leaked to an attacker allows the attacker to obtain the file from the server. Ideally, we would like the smallest representation of the file to be as long as the amount of entropy in the file itself. There are three regimes of parameters to keep in mind here:

Low-entropy files. These could be either small files, or large files that are mostly known. For example, consider an MS-Word document describing an employee in a company. The file describing two different employees may be very large, but they differ on very few fields. Hence from the point of view of employee A, the file of employee B has very small entropy. In this case we want to ensure that employee A cannot use the storage server to get the file of employee B without knowing the (very few) fields that separate the two files.

Medium-entropy files. These are files that have quite a lot of uncertainty, but still much less than their actual size. For example, consider an attacker that has a new movie with an invisible digital watermarking, and wants to get the non-watermarked movie from the storage server. Here the movie itself could be many megabytes long, and the only thing that the attacker does not know is the exact embedding of the watermarks which could be just a few kilobytes.

In this case we want to ensure that even if some information about the watermarks leaks to the attacker (by accident or malice), as long as not all the information leaked the attacker cannot use the storage server to get the non-watermarked movie. (We remark that security in this setting is the most technically complicated of the three.)

High-entropy files. These are large files that are mostly unknown to the attacker. For example, one can think of attackers that try to use the storage server as a content-distribution network (CDN): One client uploads the file to a server, then want to send to a second client just a short message that allows the second client to get the file from the server. In this case we would like to ensure that the "short message" cannot be very short, ideally it has to be about as long as the file itself.

To capture security in these various settings, our attack scenario postulates an arbitrary distribution $\mathcal{D}$ from which the input file is chosen, $F \xleftarrow{\$} \mathcal{D}$. The file is then given to the storage server who runs the summary function to get $v = S(F)$.

Next there is a "learning phase" in which the adversary can set-up arbitrarily many client accomplices and have them receive the file F and run the proof protocol with the server. (The server plays

the honest verifier with input v and the accomplices follow any arbitrary protocol set by the adversary.) The accomplices can also interact with the adversary during the same "learning phase."

Then the game moves to a "proof phase", in which the adversary engages in the proof protocol with the server, trying to prove that it has the file. (We can even iterate between "learning phases" and "proof phases", as long as the adversarial clients cannot talk to the adversary concurrently with any run of the proof protocol between the adversary and the server.

Our first (and strongest) security definition seamlessly captures security in all of these settings. Roughly it says that as long as the min-entropy in the original distribution is sufficiently larger than the number of bits sent to the adversary by the accomplices, the adversary should not be able to convince the server that it has the file.

Definition 1. *Fix parameters $s \in \mathbb{N}, \varepsilon \in [0,1]$. A scheme $\mathcal{P} = (S, \Pi)$ is a* strong proof of ownership with slackness s and soundness error ε *if for any $t \in \mathbb{N}$ and any input distribution $\mathcal{D}$ with t bits of min entropy, and any adversary as above that receives less than $t - s$ bits from the accomplices during the game, the adversary has probability of convincing the storage server that it has the file that is at most negligible in s more than ε.*

In Section 3 we present our basic scheme that satisfies Definition 1.

Our second definition relaxes the restriction that the proof fails unless the accomplices send all the entropy of the file to the adversary. Instead, we have a leakage threshold T (set by the designers of the system) such that we allow the attacker to convince the server if it receives T or more bits from the accomplices, regardless of the entropy of the original file. In terms of the three parameter regimes above, this does not effect much the low- or medium-entropy cases (since we can set the threshold to a few megabytes), but it weakens the defense in the high-entropy case. For example, if we set the threshold at 64Mbyte then the server can still be used as a CDN, except that the "short" message that clients must send to each other in order to use it will be 64Mbyte long.[5]

Definition 2. *Fix parameters $s, T \in \mathbb{N}, \varepsilon \in [0,1]$. A scheme $\mathcal{P} = (S, \Pi)$ is a* proof of ownership with leakage threshold T, slackness s, and soundness error ε, *if for any $t \in \mathbb{N}$, any input distribution $\mathcal{D}$ with t bits of min entropy, and any adversary as above that receives less than $\min(T, t-s)$ bits from the accomplices during the game, the verifier rejects the proof with all but negligibly more than ε probability.*

In Section 4 we present a more efficient scheme that satisfies Definition 2. Our last definition further relaxes the security requirement, in that we do not insist on protecting every input distribution, but just distributions taken from some class (which we believe captures all the "input distributions that appear in practice"). The streaming scheme in Section 5 adheres to this definition.

Definition 3. *Fix parameters $s, T \in \mathbb{N}, \varepsilon \in [0,1]$ and a class $\mathcal{CD}$ of distributions. A scheme $\mathcal{P} = (S, \Pi)$ is a* proof of ownership with respect to $\mathcal{CD}$ with leakage threshold T, slackness s, and soundness error ε, *if for any $t \in \mathbb{N}$, any input distribution $\mathcal{D} \in \mathcal{CD}$ with t bits of min entropy, and any adversary as above that receives less than $\min(T, t-s)$ bits from the accomplices, the adversary has at most negligibly more than ε probability of convincing the storage server that it has the file.*

Note that our definitions and solutions also generalize to a computational analog of entropy, i.e. the distribution is computationally indistinguishable from one with the relevant entropy. The reason is that since the verifier in a proof-of-ownership can be efficiently simulated, then an efficient strategy that fools the proof on a pseudo-entropy distribution, can be used to distinguish between the pseudo-entropy and real entropy distributions.

[5] See Section 6 for a way to add protection against CDN attacks also to our more efficient protocols from Sections 4 and 5.

3. A Merkle-Tree-based Solution

Our first solution is also a proof of retrievability protocol: in a nutshell it works by encoding the file using an erasure code, and then building a Merkle-tree over the encoded file. Specifically, let $E : \{0,1\}^M \to \{0,1\}^{M'}$ be an erasure code, resilient to erasure of upto α fraction of the bits (for some constant $\alpha > 0$). Namely, from any $(1-\alpha)M'$ bits of $E(F)$ it is possible in principle to completely recover the original $F \in \{0,1\}^n$. For this application *it is not important that the recovery procedure be efficient*, in fact any code $E(\cdot)$ with minimum distance greater than $\alpha M'$ will do.

In addition, let H be a collision resistant hash function with output length of n bits (e.g., SHA256 with $n = 256$), and we denote by $MT_{H,b}(X)$ the binary Merkle tree over buffer X using b-bit leaves and the hash function H.

The Basic Construction. We have parameters $b = 256$ (the Merkle-tree leaf size), ε (the desired soundness bound), and α (the erasure recovery capability of the code). We use the collision-resistant hash function $H(\cdot)$ and the α-erasure-code $E(\cdot)$.

On M-bit input file $F \in \{0,1\}^M$, the verifier computes the encoding $X = E(F)$ and then the Merkle tree $MT_{H,b}(X)$ and keeps only the root of the tree (and the number of leaves) as verification information. During the proof protocol, the verified chooses at random u leaf indexes, $\ell_1, \ldots, \ell_u$, where u is the smallest integer such that $(1-\alpha)^u < \varepsilon$. The verifier asks the prover for the sibling-paths of all the leaves, and accepts if all the sibling paths are valid with respect to $MT_{H,b}(X)$.

Theorem 1. *The basic construction is a strong proof-of-ownership protocol as per Definition 1 with soundness $(1-\alpha)^u$.*

Proof. Assume that we have an adversary A that breaks the strong proof-of-ownership property. Specifically, assume that the erasure code can correct erasures of up to α-fraction of the input and that the adversary A runs at most some k proof protocols and succeeds in convincing the server with probability better than $t(1-\alpha)^u + \delta$ for a noticeable δ. (The probability is taken over choosing $H \in \mathcal{H}$ and $F \in D$ and over the internal randomness of the adversary and server.) We then show a collision attack on $\mathcal{H}$ that succeeds with probability at least $\delta/8k$.

The collision finder gets as input a random $H \in \mathcal{H}$, it chooses $F \in \mathcal{D}$, runs the proof-of-ownership game with the prover as per Definition 1, and whenever the adversary begins a new interaction with the server (in the role of a prover P^*), the collision finder uses the extractor K^{P^*} from Merkle-tree lemma, trying to extract a $(1-\alpha)$-fraction of the leaves of the tree. If these $1-\alpha$ fraction of the leaves differ than the corresponding leaves in the encoding of F then the forger extract a collision for H from the Merkle tree, since both the paths in the real tree $MT_{H,b}(X)$ and the paths given by P^* are valid.

It is left to analyze the success probability of this collision finder. On one hand, we know that the adversary (and therefore also the extractor) is missing at least s bits of (min-)information about F, even when we throw-in the Merkle-tree root as additional information about F, and hence the probability of the extractor outputting $(1-\alpha)$-fraction of the encoding of F (which determine the entire file F via the encoding $E(\cdot)$) is at most 2^{-s}.

On the other hand, we know that in at least one of the k runs of the protocol the prover P^* succeeds in convincing the verifier with probability at least $(1-\alpha)^u + \delta/k$. Hence with probability at least

$\delta/2k$ over the choice of $H \in \mathcal{H}$, this function H still leaves the adversary with probability greater than $(1-\alpha)^u + \delta/2k$ of convincing the verifier in this run. For such H, by Lemma 1 the extractor will have probability of at least 1/4 to output $(1-\alpha)$-fraction of the leaves. Hence for such H's we will find collisions with probability at least $1/4 - 2^{-s}$, and the overall collision probability is at least $\frac{\delta}{2k} \cdot (\frac{1}{4} - 2^{-s}) \approx \delta/8k$, which is still noticeable. □

4. Protocols with Small Space

The problem with using the generic solution from the previous section is that good erasure codes for very large files are expensive to compute. In particular, it seems that computing a good erasure code require random access to the bits of the file, which means that for files larger than the available memory this computation entails many accesses to the disk.

We get around this problem by introducing a security/space tradeoff: Namely we allow the designer of the system to specify a memory-bound L such that (a) the client only needs roughly L bits of memory during the computation even for a very large files, but (b) leaking more than $O(L)$ bits may allow the adversary to cheat in the proof (even if the file has much higher entropy).

The idea is that instead of encoding the file with an erasure code, we hash it down to L bits using a hash function with public randomness, and then do the Merkle tree protocol over the hashed value. What makes this solution nontrivial is the fact that the adversary can choose the leakage functions depending on the public randomness, for example the adversary can ask to see $L/2$ bits of the hash value. In a little more details, we consider a game where the file is chosen from an arbitrary distribution (that does not depend on the hash function), then we choose the hash function at random, and then the adversary sees the hash function and can ask the accomplices for leakage that depends on it.

4.1 Using Pairwise Independent Hashing

One solution that works is to use an arbitrary pairwise-universal hash family H that maps M bit files to (say) L bits. In is instructive to think of the setting $M \gg L \gg s$, and about input distribution $\mathcal{D}$ with $k = L + 2s$ bits of min-entropy. Note that with high probability over choice of $h \in H$ it is likely that $h(F)$ has high min-entropy, but this is not enough for us. For example, an adversary can try to find a large set of blocks in the buffer $h(F)$ such that the projection of $h(F)$ onto these blocks has small min entropy. Then the adversary can ask for leakage only on these blocks, and hope that all the queries during the Merkle-tree hash will fall in this large set of blocks.

To show that this line of attack does not work we need to prove that when h is chosen at random from a pairwise independent family, then with high probability *every* large subset of the blocks of $h(F)$ has high min-entropy. This is proved in Lemma 2, whose proof is given in Appendix A.2.

Lemma 2. *Fix integers b, k, M, L such that $4b$ divides L, $k \leq M$ and $2k < L(\frac{2}{3} - \frac{1}{b})$, and denote $\eta = L(\frac{2}{3} - \frac{1}{b}) - 2k$. Also let H be a pairwise independent family of hash functions from $\{0,1\}^M$ to $\{0,1\}^L$ and let $\mathcal{D}$ be a distribution over $\{0,1\}^M$ with min-entropy of k or more.*

Consider a partition of $\{0,1\}^L$ into b-bit blocks. Then for all but a $2^{-\eta}$ fraction of the functions $h \in H$ it holds for every subset of 2/3 of the blocks that the projection of $h(\mathcal{D})$ onto these blocks has min-entropy of $k-1$ or more.

Given Lemma 2, it is easy to show that the "small-space" protocol based on universal hashing meets our intermediate notion of security from Definition 2. The proof is nearly identical to the proof of Theorem 1 and is omitted here.

Theorem 2. *A protocol where the input file is first hashed to an L-bit buffer using pairwise-independent hash function and then we run the Merkle-tree protocol on the resulting L-bit buffer is a proof-of-ownership as per Definition 2, with leakage threshold $T = L(\frac{1}{3} - \frac{1}{2b})$.*

5. A Streaming Protocol

Although it is possible in principle to implement pairwise-independent hashing of M bits into an L-bit buffer in a one-pass fashion using only roughly L bits of memory (e.g., if we use linear hashing), such implementation would be prohibitively expensive for large M, L. In this section we present a variant of the space-efficient protocol that can be implemented much more efficiently, and for which we can still argue security in realistic settings. We "pay" for the efficiency by only being able to prove security for a more restrictive set of input distributions, and only under a (reasonable) assumption about the linear code that we obtain.

We present our scheme as a series of modifications to the general universal hashing scheme using linear hashing. In the basic universal hashing scheme the file is viewed as a bit vector and multiplied by a random Boolean matrix. Namely, each bit in the file is XORed to a random subset of locations in the buffer (approximately to half of the locations). The main modifications that we make are as follows:

Blocks vs. bits: Viewing the file as a bit vector is very taxing performance wise, as we need to process the bits individually. Instead we operate on blocks of B bits at a time (for example, $B = 512$ bits to match the block size of SHA256). For a file of M bits we denote the number of blocks by $m = \lceil M/B \rceil$, and the L-bit buffer holds $\ell = \lceil L/B \rceil$ blocks. All operations in the final solutions will be XOR operations on blocks of bits, allowing for high performance.

Theoretically, this choice could severely hurt the properties of our encoding (since now the first bits of the blocks never interact with the second bits of the blocks, etc.) But practically we can justify this choice under the assumption that real life uncertainty about file contents comes at the granularity of blocks, rather than uncertainty of single bits. Roughly, the attacker may have some parts of the file that it knows and others that it does not, and these parts are not individual bits but rather they are larger blocks. We thus prove the security of our scheme with respect to a input distributions that come in blocks, and argue that this should still provide adequate security in practical settings. See Section 5.1 for a precise description of the input distributions that we consider.

Sparse reduction and mixing phases: Even when working with full blocks, adding every input block to a large number of random locations requires an order of $m\ell$ operations, which is still too expensive. (Even for not-so-long files of size $\approx \ell$ blocks requires work that is quadratic in the buffer length, $O(\ell^2)$.) Instead we aim for a solution that takes as close to $m + \ell$ operations as we can get. Our encoding solution consists of two phases:

First we have a *reduction phase* in which the m-blocks file's content is reduced into the *reduction buffer* of ℓ blocks. In this stage each block from the file is XORed to a *constant* number of random locations in the buffer (where the constant is as small as 4 in our implementation). The small constant is significant for performance, but it means that we have poor diffusion. For example starting from a file with only a single unknown block we end up with a buffer with only four unknown blocks.

To overcome this, we run a *mixing phase* on the buffer (this stage works on the buffer alone, not the original file). In this phase, we make several passes over the buffer, each time XORing every block into four other random locations in the buffer. This phase resembles the reduction phase but takes as input current blocks

from the buffer rather than input blocks from the original file. Intuitively, this "unbalanced Feistel" structure should give us good diffusion if we run enough passes, since each unknown block will eventually "contaminate" all the blocks in the buffer. (We evaluated the required number of passes experimentally, and observed that 4-5 passes are enough to get very good diffusion, see Section 5.2.)

Cyclic shifts on blocks: Simply XORing full blocks onto the buffer causes an unknown block to cancel itself if it is XORed an even number of times. Hence in general we expect each unknown block to influence only 50% of the blocks in the buffer, even after many passes. To get better diffusion (without paying in performance) we apply a very simple cyclic shift to the block before XORing it. Specifically, in our implementation (with $B = 512$-bit blocks) we XOR each block into four location, with shifts of 0,128,256 and 384 positions, respectively. (These shift values were chosen to give good performance on machines with word-size up to 128 bits.) With this twist each unknown block will influence roughly $15/16$ of the blocks in the output buffer as opposed to only 1/2.

Using self contained pseudorandomness: As described so far, the transformation can be viewed as a fixed linear mapping from an m-block file to an ℓ-block buffer. However, the description of this mapping is quite large: for each block position in the file/buffer we need to specify four indexes in the buffer where this block should be XORed. Note that the server and all clients have to agree on this mapping, so we cannot just let each of them choose it at random as they go along. One option may be to choose this randomness "once and for all" and hard-wire it in the code, but this would entail choosing as many random values as the size of the largest possible file that we may want to process and storing all these values with the code.

Alternatively, we can generate this mapping pseudo-randomly from a small description, but then we need to make sure that computing the mapping itself does not consume too much resources. In out implementation we use a speedup that allows us to get the mapping "for free": Observe that the client must anyway hash the input file, so that the server could compare the hash value and decide if that file already exists in the database. Hashing the file (say with SHA256) require applying the underlying SHA256 compression function to each block of the input file, thus computing a 256-bit chaining value that is used when hashing the next block. In our implementation these intermediate IV values double also as the specification of our mapping. Namely, we derive from each intermediate chaining value the four indexes that tell us where in the output buffer to XOR the current block. Under the *random oracle* model (where SHA256 is assumed to approximate a random function), this gives us the random mapping that we need (and we store these indexes and use them again during the mixing phase). This trick greatly improves performance, since the mapping can now be computed from the file using quantities that we have anyway.

Note that we are using a different set of pseudorandom choices for different files, which heuristically could help security, but we are using it here as a performance optimization trick rather than a security enhancement mechanism. Indeed below we analyze the security for the simpler setting of a fixed random map.

5.1 Security of the Streaming Scheme

As we said above, for this implementation we can only prove security for a restrictive class of input distributions. This class is a (slight generalization of) *block-fixing distribution*: Essentially, from the attacker's perspective there are blocks in the input file that are completely random, and others that are fully known. We generalize block-fixing distributions somewhat by allowing the random blocks to be chosen from a low-rank linear space, this captures for example the case where the same random block appears in the file several times. We argue that these generalized block-fixing distributions are a reasonably good approximation to input distributions that can be found in practice, in that a real-world attacker may know some parts of a file (e.g., fixed formatting), and may even know that some unknown parts must be equal to each other (e.g., the same information appears multiple times), but typically cannot know much more complicated information about the file.

A good way of thinking about this generalized bit-fixing sources is that a length-M input file with k-bits of (min)-entropy is obtained by choosing a k-bit random vector $\vec{w} \in \{0,1\}^k$ and computing the file itself as $\vec{f} \leftarrow \vec{w} \cdot A + \vec{b}$ where $A \in \{0,1\}^{k \times M}$ and $\vec{b} \in \{0,1\}^M$ are chosen by the adversary (and A has full rank). This is generalized to B-bit blocks if each bit in w is replaced by a random B bit block but the operation of A remains the same (except applied to full blocks rather than bits).

Our hashing function can therefore be thought of as taking the "adversarially encoded" $\vec{f} = \vec{w} \cdot A + \vec{b}$ and "re-encoding" it as $\vec{r} \leftarrow h(\vec{f})$. If h itself is linear, $h(\vec{f}) = \vec{f} \cdot C$ (with $C \in \{0,1\}^{M \times L}$), then we can view the hashed image as a (coset of a) linear code, $E(\vec{w}) = \vec{w} \cdot AC + (\vec{b}C)$. We show below that the resulting scheme will be secure as long as the linear code generated by the matrix AC has large minimum distance.

Theorem 3. *Consider a PoW scheme as described with the following parameters: ℓ is the number of blocks in the buffer, t is the number of challenges on Merkle-tree leaves, and C is the matrix representing the reduce and mix transformation from m blocks to ℓ blocks. If the matrix C in the scheme is chosen so that* for every *full-rank $k \times m$ matrix A it holds with high probability (over C) that the code generated by the rows of AC has minimum distance at least some d, then the scheme is a secure proof-of-ownership with soundness $(\frac{L-d+1}{L})^t$ with respect to generalized block-fixing distributions with min-entropy k.*

The proof of Theorem 3 is given in Appendix A.3. For the security of our protocol we argue that when choosing a matrix C as a product of a random sparse reduction matrix R and many Feistel mixing steps M_i, we get a good code with high probability (for every full rank A). For example, if the code has minimum distance $\geq \ell/3$ then we get soundness of $(2/3)^t$. Unfortunately, it seems very hard to analyze the error-correction properties of such a construction, so we can only argue security under the unproven assumption that we get a good code. (In fact we can get by with a slightly weaker condition: Instead of requiring that the code AC does not have low-weight words, it is enough to assume that an efficient adversary cannot find a low-weight word in this code.)

5.2 Implementation

We turn to describe the actual implementation and parameters that we chose for our scheme, following the techniques described above.

Block size: We use 512-bit blocks, which is also the block size of SHA256.

Buffer size: For smaller files uses a buffer roughly the size of the file.[6] For large files we limit the buffer to size 64MByte. This seems sufficient for most applications (leaking 64MByte is quite a bit), and at the same time fits comfortably in main memory of contemporary machines.

Usage of Hash IVs: Hash IVs are computed in the reduction phase and then the same IVs are used again in the mixing phase. Each IV contains 256 bits and we typically use 80 (or less) of these bits as

[6] The buffer size (in blocks) is rounded up to the next power of 2, to get a full binary Merkle tree over the buffer.

Input: An M-bit file, broken into $m = \frac{M}{512}$ blocks.

Initialize buffer. A buffer is allocated with ℓ blocks of 512 bits, where $\ell = \min\{2^{20}, 2^{\lceil \log_2 m \rceil}\}$. (Note that 2^{20} blocks is 64MByte.)

Also allocate an IV buffer of m 256-bit IVs, set $IV[0] =$ SHA256-IV. We view each $IV[i]$ as both a 256-bit value and as an array of 4 indexes into Buffer.

Reduction phase. For each block $i \in [m]$:

1. Compute $IV[i] = \text{SHA256}(IV[i-1]; File[i])$

2. For $j = 0$ to 3:

3. $Block =$ Cyclic Shift of $File[i]$ by $j * 128$ bits.

4. XOR $Block$ into location $IV[i][j]$ in Buffer

Mixing phase. Repeat 5 times:

1. For each block $i \in [\ell]$, For $j = 0$ to 3:

2. $Block =$ Cyclic Shift of Buffer$[i]$ by $j * 128$ bits.

3. If $i \neq IV[i][j]$ then
XOR $Block$ into location $IV[i][j]$ in Buffer

Figure 1. Outline of the implemented construction.

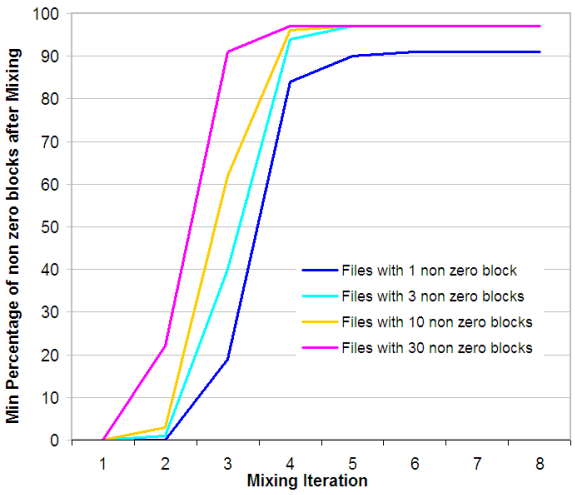

Figure 2. Mixing iterations. The number of mixing passes needed to get full diffusion, when the input file has different numbers of unknown blocks.

indexes for 4 locations in the buffer (there are at most 2^{20} blocks in the reduction buffer). If the buffer is longer than the input file then we use more bits from the IVs as indexes.

Number of iterations: The number of required iterations was tested empirically and shown in Figure 2. We see a convergence to affecting $15/16$ of the blocks within $4 - 5$ iterations. For the smaller buffer sizes, the mixing is slightly faster, and in addition, if a larger number of blocks is initially unknown, then the mixing may be as fast as 3 iterations. We set the number of iterations to 5 to accommodate for the largest buffer size (64MByte) and as low as one unknown block.

Number of challenge leaves: We set the number of challenge leaves in the Merkle tree to 20. Under the assumption that the code gives distance of $0.1L$ this will translate to a soundness error on the order of 2^{-66} if no leakage from accomplices is allowed, and to an error on the order of 2^{-20} if leakage of up to $0.4L$ is allowed. Note that the error does not have to be negligibly small, since the service can block de-duplication attempts for a file once it has failed a PoW more than, say, 3 times.

5.3 Performance Evaluation

We implemented a prototype of the PoW protocol, and ran it to evaluate performance and asses the PoW scheme benefits. We ran the protocol on random files of sizes 16KByte through 16GByte. The measurements were performed on an Intel machine with Xeon X5570 CPU running at 2.93GHz. We implemented the protocol in C++ and used the SHA256 implementation from Crypto++ [5]. We considered the following aspects of the implementation:

Client time (C_{time}). This includes the time for reading the file from disk, computing its SHA256 value, performing the reduction and mixing phases, and finally computing the Merkle-tree over the resulting buffer.

Our measurements show that the XORs during the reduction phase add less than 28% time over the insecure solution of just computing the SHA256 value of the file. The mixing and Merkle-tree times are relatively significant for small files (about 3 fold slowdown compared to only computing the SHA256 value), but once the file grows beyond 64MByte these times remain constant (844 and 1158ms. respectively) because these procedures are only applied to the 64MByte buffer. [7]

For files larger than 2GByte the overhead of our scheme (vs. only computing SHA256) is less than 50%. When further increasing the file size this overhead continues to decrease and the time to compute SHA256 becomes more dominant. These results are depicted in Figure 3, and the full set of numbers are given in Table A.1

Server time (S_{time}). This is the time spent by the server checking the Merkle tree authentication signatures. We observed that the overhead of these tests is very low, validating 20 sibling paths takes approximately 0.6 ms.

Network time (N_{time}). This is the estimated time required to transfer the protocol data. The overall size of the message transmitted by the protocol for 20 challenges is less than 20K. Thus, the network transmission time for a 5Mbps network is less than 0.1ms and is negligible.

Time savings by using deduplication. In Figures 4 and 5 we demonstrate the time-saving for the client by using deduplication (including the overhead of our scheme) compared to a policy of always sending the entire file to the server. We considered two network settings, one with an average speed for a fast network (5Mbps) [2] and the other being an extremely fast network (100Mbps). We observed that for the 5Mbps network, the PoW scheme always consumes less time than transmitting the file over the network (even for files of about 15KBytes). For the 100Mbps network deduplication (including the PoW) is faster than transmitting the file for files larger than 64KBytes. Furthermore, the benefit from using the PoW increases with the file size, e.g., for files larger than 1GByte the time consumed by PoW comprises of less than 1% and 20% of the upload time (for 5Mbps and 100Mbps respectively).

Overall, by using PoWs one can get the benefits of client-side deduplication without sacrificing security. The benefits become higher as files grow and for better deduplication ratio, as well as for slower networks.

[7] Note that the mixing and Merkle-tree computations can be performed after the client has sent the SHA256 value to the server and is waiting for the server to check if the file is already in the database and can be de-duplicated. Hence, in practice this time may be swallowed by the network latency.

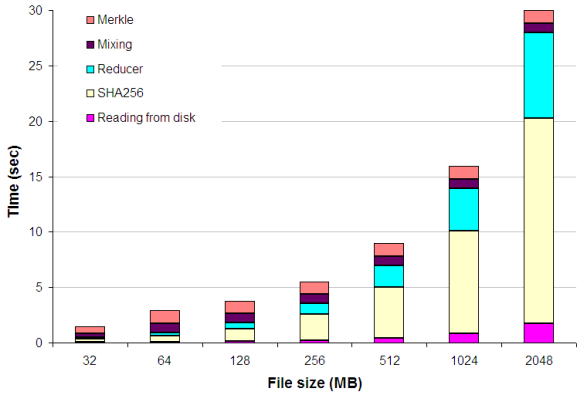

Figure 3. Performance of PoW phases. The running times of the different stages of the PoW algorithm compared to the time to read the file from the disk or calculate its SHA256.

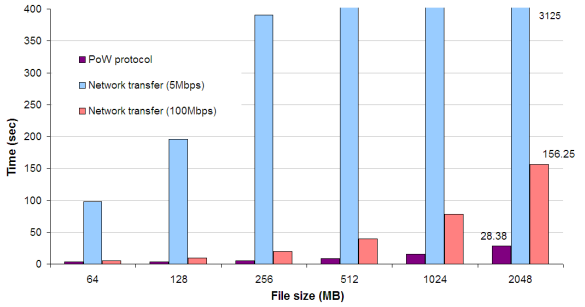

Figure 4. PoW overall performance. Comparison of the overall time consumed by the PoW scheme to the time to send the file over the networks with 5Mbps and 100Mbps upload speed. The overall running times of the PoW scheme are calculated as $T = C_{time} + S_{time} + N_{time}$.

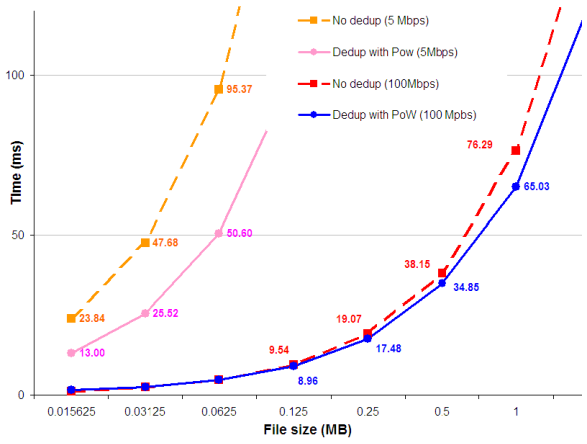

Figure 5. Benefits of deduplication with PoW Comparisons between systems that perform deduplication with PoW to those that always transmit the data over the 5Mbps and 100Mbps networks respectively. We consider a relatively conservative workload in which only 50% of the data is deduplicated and we pay the overhead of the reduction phase of the PoW scheme for remaining 50% of the files.

6. Additional Comments and Conclusions

In this work we put forward the notion of proof-of-ownership, by which a client can prove to a server that it has a copy of a file without actually sending the file. This can be used to counter attacks on file-deduplication systems where the attacker obtains a "short summary" of the file and uses it to fool the server into thinking that the attacker owns the entire file. We gave three definitions for security in this setting and three matching protocols, the last of which is very practical. Our streaming protocol allows the designer of the scheme to set a threshold for how "short" a summary can a file have (e.g., 64MBytes in our implementation). This seem suitable for the attack scenarios of common hash functions, malicious software, or accidental leakage that were described in the introduction.

For CDN attacks the protection of our protocol is not quite as strong as one could perhaps hope for, but we point out that a very simple fix can augment this streaming system also against CDN-type attacks: simply use two Merkle trees, one over the file itself and the other over the encoded buffer (and the server can keep only the hash of the two roots). Roughly speaking, the buffer-Merkle-tree addresses the low-entropy and medium-entropy cases from Section 2 while the file-Merkle-tree addresses the high-entropy case. This composed solution can be implemented using small space, and the running time will be about twice the ones that are reported in Section 5.3, since computing a Merkle tree is about twice as expensive as computing plain SHA256 over the same file.

The threshold-free notion of security from Definition 1 does not capture the added security of this composed solution, since there are setting of parameters that it does not address. For example, for a 16GByte file with 128MByte of entropy, an attacker can use only 64MByte of leakage – i.e., the encoded buffer – and has a reasonable chance of convincing the server. This is because, unlike the described solution, the Merkle-tree queries into the file itself are unlikely to hit the small fraction of the file that the attacker does not know. However this setting which is not addressed does not seem to appear much in typical applications.

We remark that the new attacks that we consider and our solutions to them are more relevant for file-level de-duplication than for block-level deduplication. (Indeed, if an attacker can learn a hash value for each 8KByte block of the file, then it can probably learn also the blocks themselves and does not need to fool the storage server.) Note, however, that the attack remains relevant (and our solution useful) when a service uses both file- and block-level de-duplication, as is likely to be the case in practical systems.

Acknowledgment: We thank Dalit Naor for her support throughout this work.

References

[1] Dropship - dropbox api utilities. `https://github.com/driverdan/dropship`, April 2011.

[2] Akamai. *The State of the Internet*, 3rd Quarter 2010. www.akamai.com/stateoftheinternet.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS '07*, pages 598–609. ACM, 2007.

[4] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *ACM CCS '09*, pages 187–198. ACM, 2009.

[5] W. Dai. *Crypto++ Library, 5.6.1, Jan, 2011.* http://www.cryptopp.com/.

[6] G. DiCrescenzo, R. J. Lipton, and S. Walfish. Perfectly Secure Password Protocols in the Bounded Retrieval Model. In *3rd Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 225–244. Springer, 2006.

[7] L. Dorrendorf and B. Pinkas. Deduplication volnurabilities in major storage services. Under preparation, 2011.

[8] M. Dutch and L. Freeman. *Understanding data de-duplication ratios*. SNIA, February 2009. http://www.snia.org/.

[9] S. Dziembowski. Intrusion-Resilience Via the Bounded-Storage Model. In *3rd Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 207–224. Springer, 2006.

[10] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Transactions on Computational Science IV, LNCS*, pages 1–22, March 2009 2009.

[11] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[12] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *"Financial Cryptography '02"*, volume 2357 of *LNCS*, pages 120–135. Springer, 2003.

[13] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services, the case of deduplication in cloud storage. *IEEE Security and Privacy Magazine, special issue of Cloud Security*, 2010.

[14] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *ACM CCS '07*, pages 584–597. ACM, 2007.

[15] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[16] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 8 2011.

[17] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT '08*, pages 90–107. Springer-Verlag, 2008.

[18] K. Thomas. Dropbox: A file sharer's dream tool? *PC World*, April 2011. `http://www.pcworld.com/businesscenter/article/226280/`.

[19] 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'09*, pages 355–370. Springer-Verlag, 2009.

A. Proofs

A.1 Proof of the Merkle Tree Lemma

Proof. (of Lemma 1) This lemma follows from standard hardness-amplification arguments, namely a prover that succeeds in answering u leaves with probability better than $(1-\alpha)^k$ can be converted to one that can answer a single query with probability better than $(1-\alpha)$. We provide the details below.

For the fixed X and h, we say that "$P^*(l_1,\ldots,l_u)$ *is valid*" if P^* replies with valid sibling paths for all the leaves when queried on leaf indexes $l_1,\ldots l_u$, . For a leaf index $l\in[s]$ and a query index $i\in[u]$, we say that "l is i-good" if when the i-th leaf index is l and all the others are chosen at random, the prover gives a valid answer with high enough probability:

$$l \text{ is } i\text{-good} \;\Leftrightarrow\; \Pr_{l_1\ldots l_u}[P(j_1\ldots j_{i-1},j,j_{i+1}\ldots j_u) \text{ is valid}] \geq \delta/u$$

For $i\in[u]$ denote $\mathsf{Good}_i = \{l\in[s] : l \text{ is } i\text{-good}\}$. We prove that there exists at least one query-index $i\in[u]$ for which $|\mathsf{Good}_i|/s \geq 1-\alpha$. By contradiction, if not then we have

$$\begin{aligned}
&\Pr_{l_1\ldots l_u}[P(l_1,\ldots,l_u) \text{ is valid}] \\
&\quad= \Pr[P(l_1,\ldots,l_u) \text{ valid and all } l_i\text{'s are } i\text{-good}] \\
&\qquad+ \Pr[P(l_1,\ldots,l_u) \text{ valid and } \exists\, i \text{ s.t. } l_i \text{ not } i\text{-good}] \\
&\quad\leq \Pr[\text{all } l_i\text{'s are } i\text{-good}] \\
&\qquad+ \sum_{i=1}^{u} \Pr[P(l_1,\ldots,l_u)\text{valid \& } l_i \text{ not } i\text{-good}] \\
&\quad\leq \prod_{i=1}^{u}\frac{|\mathsf{Good}_i|}{s} + \sum_{i=1}^{u}\Pr\left[\begin{array}{r} P(l_1,\ldots,l_n) \text{ valid} \\ \mid l_i \text{ not } i\text{-good}\end{array}\right] \\
&\quad< (1-\alpha)^u + u\cdot(\delta/u) \;=\; (1-\alpha)^u+\delta
\end{aligned}$$

but we know that $\Pr_{l_1\ldots l_u}[P(l_1\ldots l_u) \text{ is valid}] \geq (1-\alpha)^u+\delta$, so we have contradiction. We can now describe the extractor K:

1. for $i=1$ to u, for $l=1$ to s
2. repeat $\lceil u(\log(s)+1)/\delta\rceil$ times:
3. choose at random $l_1\ldots l_u\in[s]$ and query $P(l_1\ldots l_{i-1},l,l_{i+1}\ldots l_u)$
4. output sibling paths for all the leaves for which P ever gave any valid sibling path.

Clearly, if l is i-good then during the loop with i,l the prover will return a valid answer with probability at least $1-e^{-\log s-1} > 1-\frac{1}{e\cdot s}$. Hence when we arrive at the index i for which $|\mathsf{Good}_i|/s \geq 1-\alpha$, we will find valid sibling paths for an $(1-\alpha)$-fraction of the leaves, except with probability smaller than $1/e$. □

Note that the lemma does not rely on the hash function h being collision resistant. However, when used with a collision-resistant hash function, the lemma implies that the leaf values that K outputs are consistent with the original input buffer X (otherwise we get a collision).

A.2 Pairwise-Independent Hashing

To prove Lemma 2 we first prove that hashing a file with k bits of entropy into a buffer of size more than $2k$ bits does not lose much entropy.

Lemma 3. *Fix integers k,M,S such that $k\leq M$ and $2k<S$. Also let H be a pairwise independent family of hash functions from $\{0,1\}^M$ to $\{0,1\}^S$ and let $\mathcal{D}$ be a distribution over $\{0,1\}^M$ with min-entropy of k or more. Then for all but a 2^{2k-S} fraction of the functions $h\in H$ it holds that the distribution $h(\mathcal{D})$ (over $\{0,1\}^S$) has min-entropy $k-1$ or more.*

Proof. Let $\mathcal{D}(x)$ denote the probability mass of the point x under $\mathcal{D}$. For any $h\in H$, denote by $\mathsf{Col}(h)$ the "nontrivial collision probability" of $h(\mathcal{D})$, namely

$$\begin{aligned}
\mathsf{Col}(h) &\stackrel{\text{def}}{=} \Pr_{x,y\in\mathcal{D}}[x\neq y \text{ and } h(x)=h(y)] \\
&= \sum_{x\neq y}\mathcal{D}(x)\mathcal{D}(y)\cdot\chi[h(x)=h(y)]
\end{aligned}$$

(where $\chi[h(x)=h(y)]$ is the indicator variable which is 1 when $h(x)=h(y)$ and zero otherwise). We will show that for all but a 2^{2k-S} fraction of the functions $h\in H$ we have $\mathsf{Col}(h)\leq 2^{-2k}$, and that the min-entropy of $h(\mathcal{D})$ is at least $k-1$ for every such function h.

Running Times of Our Scheme

Size (MB)	Disk Read (ms)	SHA256 (ms)	SHA256 & Reducer (ms)	Mixing (ms)	Merkle scheme (ms)	PoW Total (ms)
0.015625	0.06	0.34	0.36	0.10	0.61	1.09
0.03125	0.08	0.61	0.66	0.18	1.13	2.00
0.0625	0.12	1.15	1.26	0.36	2.17	3.88
0.125	0.21	2.20	2.54	0.74	4.25	7.68
0.25	0.39	4.28	4.91	1.59	8.39	15.19
0.5	0.72	8.46	9.85	3.64	16.68	30.85
1	1.40	16.71	19.61	7.73	24.19	53.05
2	2.77	32.82	38.84	13.86	36.27	92.72
4	5.46	65.57	75.09	20.60	72.37	171.94
8	10.88	89.99	113.94	58.83	144.65	327.69
16	21.60	170.54	212.07	171.80	289.36	707.68
32	37.19	316.64	431.67	393.68	578.99	1482.29
64	56.88	602.21	849.63	844.03	1158.18	3020.11
128	111.17	1164.32	1678.60	845.24	1157.80	3850.34
256	207.07	2340.62	3312.91	844.67	1157.97	5484.15
512	405.68	4639.60	6573.86	844.08	1157.77	8745.30
1024	801.13	9256.90	13127.82	844.40	1157.83	15298.99
2048	1771.56	18515.19	26211.31	843.71	1157.85	28381.91
4096	6475.74	37004.00	52411.50	844.50	1157.90	54582.35
8192	13010.00	74108.86	104835.00	843.62	1157.92	107006.87
16384	26085.41	151079.76	209760.43	844.70	1158.23	211931.98

Table 1. Performance Measurements. Time measurements of our implementation. The leftmost column present the file size in MegaBytes. The next two columns present our measurements for the time required to read a file from a disk or calculate its SHA256. The last three columns present the time consumed by the Reduction and Mixing stages as well as the total time of the PoW algorithm, see Section 5.3. The measurements were done on Intel Xeon CPU X5570, 2.93GHz, running a C++ code, using Crypto++ for SHA256 calculations [5].

Note that the expected value of $\mathsf{Col}(h)$ over a random choice $h \in \mathcal{D}$ is

$$\begin{aligned} E_h[\mathsf{Col}(h)] &= \sum_{h\in H}\frac{1}{|H|}\sum_{x\neq y}\mathcal{D}(x)\mathcal{D}(y)\chi[h(x)=h(y)] \\ &= \sum_{x\neq y}\mathcal{D}(x)\mathcal{D}(y)\sum_{h\in H}\frac{\chi[h(x)=h(y)]}{|H|} \\ &= \sum_{x\neq y}\mathcal{D}(x)\mathcal{D}(y)\Pr_{h\in H}[h(x)=h(y)] \\ &\overset{(*)}{=} 2^{-S}\sum_{x\neq y}\mathcal{D}(x)\mathcal{D}(y) \;\leq\; 2^{-S}, \end{aligned}$$

where Equality $(*)$ follows from pairwise independence of H. Since the expected value of $\mathsf{Col}(h)$ is at most 2^{-S}, it follows from Markov's inequality that $\Pr_{h\in H}[\mathsf{Col}(h) > 2^{-2k}] < 2^{2k-S}$.

Consider now a fixed function $h \in H$ for which $\mathsf{Col}(h) \leq 2^{-2k}$, let $z \in \{0,1\}^m$ be an arbitrary point, and we show that the probability mass of z under $h(\mathcal{D})$ is at most 2^{1-k}. Namely, $\sum_{x\in h^{-1}(z)}\mathcal{D}(x) \leq 2^{1-k}$. Denote the pre-image set of z by $h^{-1}(z) = \{x_1, x_2, \ldots, x_N\}$, where the x_i's are ordered by their probability mass under $\mathcal{D}$. Namely $2^{-k} \geq \mathcal{D}(x_1) \geq \mathcal{D}(x_2) \geq \ldots \geq \mathcal{D}(x_N)$. Then we have

$$\begin{aligned} 2^{-2k} &\overset{(a)}{\geq} \sum_{i\neq j}\mathcal{D}(x_i)\mathcal{D}(x_j) = \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\mathcal{D}(x_i)\mathcal{D}(x_j) \\ &\overset{(b)}{\geq} \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\mathcal{D}(x_{i+1})\mathcal{D}(x_j) \\ &= \sum_{i=1}^{N-1}\left(\mathcal{D}(x_{i+1})^2 + \sum_{j=i+2}^{N}\mathcal{D}(x_{i+1})\mathcal{D}(x_j)\right) \\ &\geq \sum_{i=2}^{N}\mathcal{D}(x_i)^2 \overset{(c)}{\geq} \sum_{i=1}^{N}\mathcal{D}(x_i)^2 - 2^{-2k}, \end{aligned}$$

where inequality (a) follows from $\mathsf{Col}(h) \leq 2^{-2k}$, inequality (b) is because $\mathcal{D}_{x_i} \geq \mathcal{D}(x_{i+1})$, and inequality (c) follows from $2^{-k} \geq \mathcal{D}(x_1)$. We therefore conclude that $\sum_{i=1}^{N}\mathcal{D}(x_i)^2 \leq 2^{1-2k}$. This implies $\left(\sum_{x\in h^{-1}(z)}\mathcal{D}(x)\right)^2 = \sum_{i=1}^{N}\mathcal{D}(x_i)^2 + \sum_{i\neq j}\mathcal{D}(x_i)\mathcal{D}(x_j) \leq 2^{1-2k} + 2^{-2k} = 3\times 2^{-2k}$, and therefore $\sum_{x\in h^{-1}(z)}\mathcal{D}(x) \leq \sqrt{3}\times 2^{-k} < 2^{1-k}$. □

Lemma 2. *Fix integers b, k, M, L such that $4b$ divides L, $k \leq M$ and $2k < L(\frac{2}{3} - \frac{1}{b})$, and denote $\eta = L(\frac{2}{3} - \frac{1}{b}) - 2k$. Also let H be a pairwise independent family of hash functions from $\{0,1\}^M$ to $\{0,1\}^L$ and let $\mathcal{D}$ be a distribution over $\{0,1\}^M$ with min-entropy of k or more.*

Consider a partition of $\{0,1\}^L$ into b-bit blocks. Then for all but a $2^{-\eta}$ fraction of the functions $h \in H$ it holds for every subset of 2/3 of the blocks that the projection of $h(\mathcal{D})$ onto these blocks has min-entropy of $k-1$ or more.

Proof. Note that for every fixed subset of half the blocks, the projection of H onto these blocks is itself a pairwise independent family of hash functions from $\{0,1\}^M$ to $\{0,1\}^S$ where $S = 2L/3$. It follows from Lemma 3 that for all but a 2^{2k-S} fraction of the function $h \in H$, the projection of $h(\mathcal{D})$ onto these blocks has min-entropy at least $k-1$.

Since there are L/b blocks then there are less than $2^{L/b}$ subsets of half the blocks, hence by the union bound the probability (over choosing $h \in H$) that the projection of $h(\mathcal{D})$ on any of these subsets has less than $k-1$ min entropy is at most $2^{2k-S} \cdot 2^{L/b} = 2^{2k-(2L/3)+(L/b)} = 2^{2k-L(2/3-1/b)} = 2^{-\eta}$. □

A.3 Security of the Streaming Scheme

We begin by stating a well-known lemma about linear codes:

Lemma 4. *Let $L > k$ and let $M \in \{0,1\}^{k \times L}$ be a binary matrix such that the code generated by the rows of M has minimum distance at least some d. Also let $\vec{z} \in \{0,1\}^L$ be some fixed vector and consider the distribution obtained by choosing a random $w \in \{0,1\}^k$ and outputting $\vec{w} \cdot M + \vec{z}$. Then the projection of z onto any $L - d + 1$ coordinates has k bits of (min-)entropy.*

Proof. Fix some set of $L - d + 1$ coordinates and consider the corresponding subset of columns of M and coordinates in $\vec{z}$, which we denote by M' and $\vec{z'}$. We show that the distribution on $\vec{w} \cdot M' + \vec{z'}$ is uniform over a set of size 2^k, hence it has k bits of (min-)entropy. To see this, notice that the rows of M' must be linearly independent: Otherwise there would be a nonzero vector $\vec{w^*} \neq \vec{0}$ such that $\vec{w^*} \cdot M' = \vec{0}$, so the only possibly-nonzero coordinates in $\vec{w^*} \cdot M$ are the $d-1$ coordinates that are missing in M', hence $\vec{w^*} \cdot M$ would have Hamming weight at most $d-1$, which is a contradiction. This means that the mapping $\vec{w} \mapsto \vec{w} \cdot M' + \vec{z'}$ is one-to-one. Hence the distribution over $\vec{w} \cdot M' + \vec{z'}$ (for a random $\vec{w}$) is uniform over a set of size 2^k. □

Corollary 5. *Consider input distribution uniform over $\vec{f} = \vec{w} \cdot A + \vec{b}$, and a reduction function $h(\vec{f}) = \vec{f} \cdot C$. If the code generated by the rows of AC has minimum distance at least some d then no adversary can convince the server in one execution of the proof protocol from Section 5 with prob. noticeably better than $(\frac{L-d+1}{L})^t$.*

Proof. (sketch) The Merkle-tree lemma says that any prover that convinces the verifier with probability noticeably better than $(\frac{L-d+1}{L})^t$ can be converted into an extractor that outputs the value of a fraction $(\frac{L-d+1}{L})$ of the leaves (together with accepting sibling paths). On the other hand, Lemma 4 says that every set of $(\frac{L-d+1}{L})$ of the (256-bit) leaves must at least $256k$ bits of min-entropy. Hence even after the accomplices leak to the adversary all but $256(k-2)$ bits and the server gives it the Merkle-tree root (consisting of 256 bits), the adversary (and thus the extractor) are missing at least 256 bits. Hence the probability that the leaves that the extractor outputs equal to the original file is at most 2^{-256}, and hence with probability $1 - 2^{-256}$ we get a hash collision in the Merkle tree. □

As an immediate corollary we get Theorem 3.