# Enabling Security in Cloud Storage SLAs with CloudProof

Raluca Ada Popa
*MIT*

Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang
*Microsoft Research*

## Abstract

Several cloud storage systems exist today, but none of them provide security guarantees in their Service Level Agreements (SLAs). This lack of security support has been a major hurdle for the adoption of cloud services, especially for enterprises and cautious consumers. To fix this issue, we present *CloudProof*, a secure storage system specifically designed for the cloud. In CloudProof, customers can not only *detect* violations of integrity, write-serializability, and freshness, they can also *prove* the occurrence of these violations to a third party. This proof-based system is critical to enabling security guarantees in SLAs, wherein clients pay for a desired level of security and are assured they will receive a certain compensation in the event of cloud misbehavior. Furthermore, since CloudProof aims to scale to the size of large enterprises, we delegate as much work as possible to the cloud and use cryptographic tools to allow customers to detect and prove cloud misbehavior. Our evaluation of CloudProof indicates that its security mechanisms have a reasonable cost: they incur a latency overhead of only $\sim$15% on reads and writes, and reduce throughput by around 10%. We also achieve highly scalable access control, with membership management (addition and removal of members' permissions) for a large proprietary software with more than 5000 developers taking only a few seconds per month.

## 1 Introduction

Storing important data with cloud storage providers comes with serious security risks. The cloud can leak confidential data, modify the data, or return inconsistent data to different users. This may happen due to bugs, crashes, operator errors, or misconfigurations. Furthermore, malicious security breaches can be much harder to detect or more damaging than accidental ones: external adversaries may penetrate the cloud storage provider, or employees of the service provider may commit an insider attack. These concerns have prevented security-conscious enterprises and consumers from using the cloud despite its benefits [16].

These concerns are not merely academic. In June 2008, Amazon started receiving public reports that data on its popular Simple Storage Service (S3) had been corrupted due to an internal failure; files no longer matched customers' hashes [11]. One day later, Amazon confirmed the failure, and cited a faulty load balancer that had corrupted single bytes in S3 responses "intermittently, under load." Another example of data security violation in the cloud occurred when Google Docs had an access-control bug that allowed inadvertent sharing of documents with unauthorized readers [28]. Even worse, LinkUp (MediaMax), a cloud storage provider, went out of business after losing 45% of client data because of administrator error.

None of today's cloud storage services—Amazon's S3, Google's BigTable, HP, Microsoft's Azure, Nirvanix CloudNAS, or others—provide security guarantees in their Service Level Agreements (SLAs). For example, S3's SLA [1] and Azure's SLA [23] only guarantee availability: if availability falls below 99.9%, clients are reimbursed a contractual sum of money. As cloud storage moves towards a commodity business, security will be a key way for providers to differentiate themselves. In this paper, we tackle the problem of designing a cloud storage system that makes it possible to detect violations of security properties, which in turn enables meaningful security SLAs.

The cloud security setting is different from the setting of previous secure storage or file systems research. The first difference is that there is a financial contract between clients and the cloud provider: clients pay for service in exchange for certain guarantees and the cloud is a liable entity. In most previous work [3, 4, 10, 20], the server was some group of untrusted remote machines that could not guarantee any service. The second difference is that scalability is more important, as it is one of the primary promises of the cloud. Enterprises are important customers for the cloud; they have many employees requiring highly scalable access control and have large amounts of data.

We identify four desirable security properties of cloud storage: confidentiality, integrity, write-serializability, and read freshness (denoted by C, I, W, F). If a customer has such security guarantees, his data is confiden-

tial, cannot be modified by any unauthorized party, is consistent among updates made by authorized users, and is fresh as of the last update.

We design, build, implement, and evaluate CloudProof, a secure and practical storage system specifically designed for the cloud setting. Our first novelty is the idea and the mechanism of enabling customers *to prove to third parties when the cloud violates the IWF properties*. (Confidentiality is not included because customers can provide it to themselves by encrypting the data they store on the cloud.) This enabling of proofs is in addition to detecting the violations and is not present in previous work. It includes the fact that *the cloud can disprove false accusations made by clients*; that is, in CloudProof, clients cannot frame the cloud. We believe that such proofs are *key* to enabling security in SLAs with respect to these three properties. Customers and cloud can now establish a financial contract by which clients pay a certain sum of money for the level of security desired; customers have assurance that the cloud will pay back an agreed-upon compensation in case their data security is forfeited because they can prove this violation. Without such proofs, the cloud can claim a smaller amount of damage to protect itself against significant financial loss and clients can falsely accuse the cloud. These proofs are based on *attestations*, which are signed messages that bind the clients to the requests they make and the cloud to a certain state of the data. For every request, clients and cloud exchange attestations. These attestations will be used in a lightweight auditing protocol to verify the cloud's behavior.

The second novelty is CloudProof, the system as a whole, in which we put engineering effort to maintain cloud scalability while detecting and proving violations to all three IWF properties and providing access control. Previous work did not provide detection for both write-serializability and freshness at the same time. In addition, most related work has not been designed with cloud scalability in mind and we argue that they are not readily extendable to provide it. Our design principle is to *offload as much of the work as possible to the cloud, but verify it*. Therefore, access control, key distribution, read, write, file creation, and ensuring the aforementioned security properties are delegated to the cloud to the extent possible. To enable this delegation, we employ cryptographic tools from the literature: to achieve scalable access control, we use in a novel way key rolling and broadcast encryption. We also have a novel way to group data by access control list into "block families" that allows us to easily handle changes in access control.

CloudProof targets most applications that could benefit from the cloud: large departmental or enterprise file systems, source code repositories, or even small, personal file systems. These tend to be applications that can tolerate larger client-cloud latency (which is an inherent result of the different geographic locations of various clients/organizations with respect to the cloud). Yet, a surprising number of applications benefit from the cloud. For example, the Dropbox service uses S3 storage to provide backup and shared folders to over three million users. The SmugMug photo hosting service has used S3 since April 2006 to hold photos, adding ten terabytes of data each month without needing to invest in dedicated infrastructure. AF83 and Indy500.com use S3 to hold static web page content.

Any security solution for cloud storage must have a limited performance impact. We have prototyped CloudProof on Windows Azure [22]. In Section 9 we report experiments that measure the latency and throughput added by CloudProof compared to the storage system without any security. In microbenchmarks, for providing all four of our properties, we add $\sim$0.07s of overhead ($\sim$15%) to small block reads or writes, and achieve only $\sim$10% throughput reduction and 15% latency overhead for macrobenchmarks. One can audit the activity of a large company during a month in 4 min and perform membership changes (adding and revoking member permissions) for a source code repository of a very large proprietary software that involved more than 5000 developers in a few seconds per month. Overall, our evaluations show that we achieve our security properties at a reasonable cost.

## 2 Setting

CloudProof can be built on top of conventional cloud storage services like Amazon S3 or Azure Blob Storage. The storage takes the form of key-value pairs accessed through a `get` and `put` interface: the keys are block IDs and the values are the contents of the blocks. Blocks can have any size and can vary in size.

There are three parties involved in CloudProof:

1. *(Data) owner:* the entity who purchases the cloud storage service. A data owner might be an enterprise with business data or a home user with personal data.
2. *Cloud:* the cloud storage provider.
3. *(Data) users:* users who are given either read or write access to data on the cloud. A user might be an employee of an enterprise or family members and friends of a home user.

The data owner is the only one allowed to give access permissions to users. The access types are read and read/write. Each block has an access control list (ACL), which is a list of users and their accesses to the block. (One can easily implement a group interface by organizing users in groups and adding groups to ACLs.) When talking about reading or modifying a block, a *legitimate* user is a user who has the required access permissions to the block. We assume that the data owner and the cloud have well-known public keys, as is the case with existing

providers like Amazon S3.

## 2.1 Threat Model

The cloud is *entirely untrusted*. It may return arbitrary data for any request from the owner or any user. Furthermore, the cloud may not honor the access control lists created by the owner and send values to a user not on the corresponding access control list. A user is trusted with the data he is given access to. However, he may attempt to subvert limits on his permission to access data, possibly in collusion with the cloud. An owner is trusted with accessing the data because it belongs to him. However, the users and the owner may attempt to falsely accuse the cloud of violating one of our security properties.

We make standard cryptographic assumptions for the tools we use: existential unforgeability under chosen message attack for public-key signature schemes, collision-resistance and one-way function property for hash functions, and semantic security for symmetric encryption schemes.

## 2.2 Goals

Let us first define the security properties CloudProof provides. *Confidentiality (C)* holds when the cloud or any illegitimate user cannot identify the contents of any blocks stored on the cloud. *Integrity (I)* holds when each read returns the content put by a legitimate user. For example, the cloud cannot replace some data with junk. *Write-serializability (W)* holds when each user committing an update is aware of the latest committed update to the same block. W implies that there is a total order on the writes to the same block. *Freshness (F)* holds if reads return the data from *the latest committed write.* Note that we cannot guarantee that each block retrieved was the most recently received by the cloud, because, upon two parallel writes to the block, the cloud can pretend to have received them in a different order or network delays can arbitrarily reorder such requests. Instead, we aim to guarantee that the last committed write (for which the cloud acknowledged receipt to the client, as we will see) will be visible during read. A violation to the security of a user is when the IWF properties do not hold.

CloudProof has the following four goals.

*Goal 1:* Users should detect the cloud's violations of integrity, freshness, and write-serializability. Users should provide confidentiality to themselves by encrypting the data they store on the cloud.

*Goal 2:* Users should be able to *prove* cloud violations whenever they happen. Any proof system has two requirements: (1) the user can convince a third party of any true cloud violation; and (2) the user cannot convince a third party when his accusation of violation is false.

*Goal 3:* CloudProof should provide read and write access control in a scalable (available) way. Since we are targeting enterprise sizes, there may be hundreds of thousands of users, many groups, and terabytes of data. We want to remove data owners from the data access path as much as possible for performance reasons. Owners should be able to rely (in a verifiable way) on the cloud for key distribution and access control, which is a highly challenging task.

*Goal 4:* CloudProof should maintain the performance, scalability, and availability of cloud services despite adding security. The overhead should be acceptable compared to the cloud service without security, and concurrency should be maintained. The system should scale to large amounts of data, many users per group, and many groups, since this is demanded by large enterprise data owners.

In the remainder of this paper, we show how CloudProof achieves these goals.

# 3 System Overview

In this section, we present an overview of our system; we will elaborate on each component in later sections.

CloudProof's interface consists of get(BlockID blockID) and put(BlockID blockID, byte[] content). BlockID is a flat identifier that refers to a block on the cloud. The get command reads content of a block, while the put command writes in the block identified by blockID. Creation of a block is performed using a put with a new blockID and deletion is performed by sending a put for an empty file. CloudProof works with any cloud storage that exports the following key/value store interface and can sign messages verifiable by other parties using some known public key.

The design principle of CloudProof is to *offload as much work as possible to the cloud, but be able to verify it*. The cloud processes reads and writes, maintains data integrity, freshness, and write-serializability, performs key distribution (protected by encryption), and controls write accesses. Users and the owner verify the cloud performed these operations correctly.

We put considerable engineering effort into keeping the system scalable. The data owner only performs group membership changes and audits the cloud. Both tasks are offline actions and lightweight, as we will see in Section 9. Thus, the data owner is not actively present in any data access path (i.e. put or get), ensuring the service is scalable and available even when the data owner is not. Moreover, access control is delegated to the cloud or distributed. As part of access control, key distribution is delegated in a novel way using broadcast encryption and key rolling to the cloud: the data owner only needs to change one data block when a user is revoked rather than all the blocks the user had access to. Most data reencryption is distributed to the clients for scalability. Importantly, we strived to keep accesses to different blocks running in parallel and avoided the typical serialization

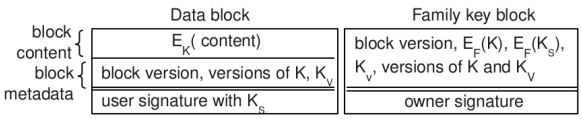


Figure 1: Layout of block formats in a family and of the key block. $K$ is the read access key, $K_S$ is the signing key, and $K_V$ is the verification key. $E_F()$ means broadcast encryption to the authorized users in the ACL of a certain block family. The block version is an integer for each block that is incremented by one with every update to the block committed on the cloud.

so often encountered in security protocols (e.g., computations of Merkle trees over multiple blocks upon data write).

The key mechanism we use is the exchange of *attestations* between the owner, users, and the cloud. When users access cloud storage through the get or put interface, each request and response is associated with an attestation. Attestations keep users and the cloud accountable and allow for later proofs of misbehavior, as we discuss in Section 5.

We divide time into *epochs*, which are time periods at the end of which data owners perform auditing. At the end of each epoch, owners also perform membership changes such as addition or removal of members. (Owners can also change membership during an epoch and clients will have to check if any keys have changed due to revocations.) Each epoch has a corresponding *epoch number* that increases with every epoch. If the system uses fixed-length epochs, clients can easily derive the current epoch identifier independently from the current time.

Briefly, CloudProof detects and proves IWF violations as follows. Clients check data integrity based on attestations they get from the cloud, as described in Section 6. The owner checks W and F during the *auditing* process. For auditing efficiency, each block has a certain probability of being audited in an epoch. During the epoch, users send the attestations they receive from the cloud to the owner. If they are disconnected from the owner, they can send these attestations any time before the epoch's end; if they are malicious or fail to send them for any reason, there is no guarantee that their gets returned correct data or their puts took effect. The owner uses these attestations to detect any violations of WF and construct a proof that convinces a third party whenever the cloud misbehaved, as we will discuss in Section 7.

# 4 Access Control

In this section, we focus only on access control; as such, we consider that the cloud does not modify the data placed by authorized users and does not provide stale or inconsistent views to users: we provide mechanisms to enforce these properties separately in Sections 6 and 7.

If the owner adds or removes a user from the ACL of a block, then that user gains or loses access to that block.

We introduce the term **block family** to describe the set of all blocks that have the same ACL. If the owner of a block changes its ACL, the block will switch to a different family. Since all blocks in a family have identical ACLs, when we need a key to enforce access control we can use the same key for every block in a family.

As mentioned, the cloud is not trusted with access control. At the same time, having the owner perform access control checks for every user get or put would be costly and unscalable. We thus follow our design principle of verifiably offloading as much work as possible to the cloud.

**Read Access Control.** To prevent unauthorized reads, all the data stored on the cloud is encrypted with a secure block or stream cipher, e.g., AES in counter mode. We denote the secret key of the cipher as the *read/get access key*. Clients with read access will have the key for decryption as described in Section 4.1 and thus will be able to access the data. Blocks in the same block family will use the same read access key.

**Write Access Control.** We achieve write access control with a public key signature scheme, as follows. For each block family, we have a public *verification key* and a private *signing key*. The verification key is known to everyone, including the cloud, but the signing key is known only to users granted write access by the ACL. Each time a user modifies a block, he computes its *integrity signature*, a signature over the hash of the updated block using the signing key. He sends this signature along with his write request, and the cloud stores it along with the block. The cloud provides it to readers so they can verify the integrity of the block.

Since the verification key is known to the cloud, it can perform write access control, as follows. Whenever a user attempts to update a block, the cloud verifies the signature and only allows the update if the signature is valid. Note that, if the cloud mistakenly allows a write without a valid signature, this failure will be detected by future data users reading the block. The mechanism of attestations, described later, will allow those users to prove this cloud misbehavior.

## 4.1 Key distribution

Our key distribution mechanism ensures that users can acquire the keys they need to access the blocks they are authorized to access. To offload as much work as possible to the cloud, the cloud performs this key distribution verifiably. We achieve this goal by employing in a novel way broadcast encryption and key rolling. **Broadcast encryption**([5, 12]) allows a broadcaster to encrypt a message to an arbitrary subset of a group of users. Only the users in the subset can decrypt the message. Encrypting creates a ciphertext of size $O(\sqrt{\text{total no. of users in the group}})$. **Key rotation** [18] is a scheme in which a sequence of keys can be produced

from an initial key and a secret master key. Only the owner of the secret master key can produce the next key in the sequence, but any user knowing a key in the sequence can produce all earlier versions of the key (forward secrecy).

For each block family, the owner places one block on the cloud containing the key information for that family. This block is called the *family key block*. Only the data owner is allowed to modify the family key block. Recall that all blocks in a family share the same ACL, so each key block corresponds to a particular ACL that all blocks in its family share. Figure 1 illustrates the layout of blocks in a family and of the key block. The purposes of the various terms in the figure are explained in the rest of this section.

Using broadcast encryption, the data owner encrypts the read access key so that only users and groups in the ACL's read set can decrypt the key. This way, only those users and groups are able to decrypt the blocks in the corresponding family. The data owner also uses broadcast encryption to encrypt the signing key so that only users and groups in the ACL's write set can decrypt the key. This way, only those users and groups can generate update signatures for blocks in the corresponding family.

We do not try to prevent a data user giving his read access key for a block family to a data user who is not authorized for that block family. The reason is that authorized data users can simply read the information directly and give it to others. Solving this problem, e.g., with digital rights management, is beyond the scope of this paper.

## 4.2 Granting/revoking access

The owner may want to revoke the access of some users by removing them from certain groups or from individual block ACLs. When the owner revokes access of a user, he should make sure that the data user cannot access his data any more. For this goal, there are two options, each appropriate for different circumstances. *Immediate revocation* that the revoked user should not have access to any piece of data from the moment of the revocation. *Lazy revocation* means that the revoked user will not have access to any data blocks that have been updated after his revocation. The concept of lazy revocation is in [13] and is not new to us.

When a block's ACL changes, that block must undergo immediate revocation. That is, the block must switch to a new family's key block, the one corresponding to its new ACL, and the block needs to be immediately re-encrypted with that new key block.

In contrast, when a group's membership changes, then all blocks with ACLs that include that group must undergo revocation. Using immediate revocation in this case would be too expensive, as it would involve immediately re-encrypting all the blocks in one or more block families, a potentially immense amount of data. Furthermore, such an approach may be futile because a malicious revoked data user could have copied all the blocks for which he had access. Instead, we use lazy revocation, as follows.

Using key rotation, the owner rolls the keys forward to a new version for each of the families corresponding to the affected ACLs. However, the blocks with those ACLs do not need to be re-encrypted right away; they can be lazily re-encrypted. The owner only needs to update the family key blocks with the new key information. This means the work the owner has to do upon a membership change is *independent of the number of files in the block family*. Broadcast encryption has complexity $O(\sqrt{\text{no. of members in ACL}})$, which we expect to be manageable in practice, as we will show in Section 9.

When a user accesses a block, he checks whether the version of the read access key in the family key block is larger than the version of the key with which the current block was encrypted. If so, the data user re-encrypts the block with the new key. Re-encrypting with a different key does not incur any overhead since all writes require a re-encryption. Therefore, the burden of the revocation is pushed to users, but without them incurring any additional re-encryption overhead.

We can see that our division of storage into block families makes revocation easier. If, in contrast, there were multiple Unix-like groups for a block, one would need to store encryptions of the signature and block encryption key for every group with each block. Besides the storage overhead, this would require many public key operations whenever a member left a group because the key would need to be changed for every regular group. This process would be slower and more complex.

# 5 Attestations

In this section, we describe the structure and exchange of the attestations[1]. The attestations are key components that allow the clients to prove cloud misbehavior and the cloud to defend himself against false accusations.

Every time the client performs a get, the cloud will give the client a *cloud get attestation*. Every time a client performs a put, the client will give the cloud a *client put attestation* and the cloud will return a *cloud put attestation*. Intuitively, the role of the attestations is to *attest* to the behavior of each party. When the client performs a get, the cloud attaches to the response an attestation; the attestation is similar to the cloud saying "I certify that I am giving you the right data". When the client performs a put, he must provide a client put attestation which intuitively says "I am asking you to overwrite the existing

[1] CloudProof's attestations should not be confused with attestations from trusted computing, which are different mechanisms.

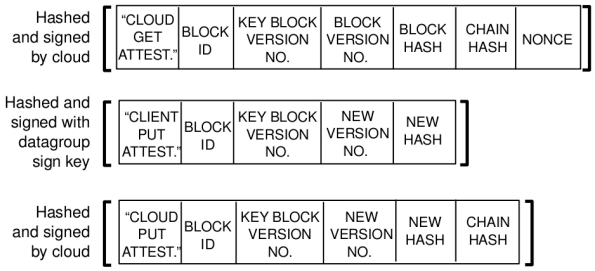


Figure 2: The structure of the attestations. The elements listed in each attestation are concatenated, a hash is computed over the concatenation and the result is signed as shown in the figure. The first field indicates the type of the attestation. The hash in client attestations is a hash over the block content and metadata, while the hash in the cloud attestation includes the integrity signature as well; the new hash is the value of this hash after an update. The nonce is a random value given by the client.

data with this content". The cloud must answer with a cloud put attestation saying "The new content is committed on the cloud".

## 5.1 Attestation Structure

Each attestation consists of concatenating some data fields, and a signed hash of these data fields. Since we are using signatures as proofs, it is important that they be non-repudiable (e.g. *unique signatures* [21]). Figure 2 illustrates the structure of the attestations. The *block version number* and *current hash* are used for write-serializability and the *chain hash* is used for freshness. The chained hash of an attestation is a hash over the data in the current attestation and the chain hash of the previous attestation.

$$\text{chain hash} = \text{hash}(\text{data}, \text{previous chain hash}) \quad (1)$$

It applies to both put and get attestations. The chain hash thus depends not only on the current attestation but also on the history of all the attestations for that block so far (because it includes the previous chain hash).

We say that that two attestations $A$ and $B$ *chain correctly* if the chain hash in $B$ is equal to the hash of $B$'s data and $A$'s chain hash. The chain hash creates a cryptographic chain between attestations: it is computationally infeasible to insert an attestation between two attestations in a way that the three attestations will chain correctly. We can see that the client put attestation is the same with the integrity signature described in Section 4 and provided with every write.

The purpose of the nonces is to prevent the cloud from constructing cloud attestations ahead of time before a certain write happens. Then the cloud could provide those stale attestations to readers and thus cause them to read stale data. Because of the cryptographic chain induced in the attestations by the chain hash, the cloud cannot insert an attestation $C$ between two other attestations $A$ and $B$; thus, the cloud can give stale data to a client and pass the attestation audit test only if he produced and saved attestation $C$ before he gave out $B$. A client gives a nonce to the cloud only upon making a request; the cloud cannot know this nonce before request $C$ is made (the nonce is randomly chosen from a large field) so the cloud will not have been able to construct an attestation for this nonce ahead of time.

The structure of the attestation depends on the security level desired. If the user does not want write-serializability and freshness, the attestations are not needed at all. If the user does not want freshness, the chain hash can be removed.

The *attestation data* is the data in the put client attestation before the hash and signature are computed. Note that the integrity signature discussed in write access control (Sec. 4) and stored with each block in Figure 1 is the last client put attestation to that block.

## 5.2 Protocols for Exchange of Attestations

**Get:**

1. *Client:* Send the get request, block ID and a random nonce to the cloud.
2. *Cloud:* Prepare and send back the entire block (including metadata and the attached integrity signature) from Fig. 1, the new chain hash, and the cloud get attestation.
3. *Client:* Verify the integrity signature and the cloud attestation (it was computed over the data in the block, chain hash, and nonce). Do not consider the get finished until all these checks succeed.

**Put:**

1. *Client:* Send the entire new block (content and metadata) and the client put attestation.
2. *Cloud:* If the client's attestation is correct (new hash is a hash of block content, datagroup verification key verifies the signature), send back the chained hash and put attestation. Store the new block (together with the client attestation).
3. *Client:* Verify the attestation. Consider a write committed only when this check succeeds.

# 6 Confidentiality and Integrity

In this section, we describe the techniques we use to ensure confidentiality and integrity. These techniques have been extensively used in previous work so we will not linger on them; rather, we explain how to integrate them in our proof mechanism and formalize the guarantees they provide.

**Confidentiality (C).** As mentioned earlier, we achieve confidentiality by having the clients encrypt the content placed on the cloud. Note that even though the cloud cannot gain information from the encrypted data, the cloud can deduce some information from the access patterns of the users to the data (e.g., frequency of reading a certain block). There has been significant theoretical work on masking access patterns [15, 27], but efficient such pro-

tocols are yet to be found so we do not make an attempt to mask them.

**Integrity (I).** As mentioned in write access control (Section 4), each time a user puts a block on the cloud, he must provide a signed hash of the block. Similarly, each time a user reads a block, he checks the signed hash using the public verification key available in the key block. Note that the special blocks used to store keys are also signed in this way, so their integrity is assured in the same way that all blocks are.

*Detection of I Violation:* the integrity signature on a block does not match the block's contents.

*Proof of Violation of I:* the block with violated integrity and the attestation from the cloud.

Recall that the get attestation from the cloud contains a hash of the block content with the integrity signature and is authenticated with the cloud's signature. If this hash of the block content does not verify with the hash from the integrity signature, it means that the cloud allowed/performed an invalid write, and the attestation itself attests to the cloud's misbehavior.

A client cannot frame an honest cloud. When the cloud returns an attestation, it is signed by the cloud so the client cannot change the contents of the attestation and claim the cloud stored a tampered block. Also, if a client falsely claims that a different verification key should be used, the cloud can exhibit the owner's signed attestation for the key block. This suffices because the data block and the key block include the version of the verification key.

## 7 Write serializability and Freshness

To detect deviations from W and F, the data owner periodically *audits* the cloud. The owner performs the auditing procedure at the end of each epoch.

A successful audit for a certain block in an epoch guarantees that the cloud maintained freshness and write-serializability of that block during the particular epoch.

The data owner assigns to each block some probability of being audited, so an audit need not check every block in every epoch. If a block is very sensitive, the owner can assign it a probability of one, meaning that the block will be audited in every epoch. If a block is not very important, the owner can assign it a smaller probability.

We cannot hide the rate at which a block is audited, since the cloud can simply observe this. However, the cloud cannot be allowed to know exactly which epochs will feature a block's audit, since if it did, it could undetectably misbehave with regard to that block during other epochs. Users, in contrast, need to be able to figure out these epochs because they need to send cloud attestations to the data owner in exactly these epochs. Thus, we use a technique that ensures that only users who know the read access key for a block can determine the epochs in which it will be audited. Specifically, we audit a block whenever:

$$\mathsf{prf}_{\mathsf{read\ key}}(\mathsf{epoch\ number},\ \mathsf{blockID}) \mod N = 0,$$

where prf is a pseudorandom function [14], $N$ is an integer included in plain text in the block metadata by the owner. If a probability of audit $p$ is desired, the data owner can achieve this by setting $N = \lfloor 1/p \rfloor$. Note that while hashes are used in practice for the same purpose as we use a prf, hashes are not secure for such usage because their cryptographic specification only guarantees collision-resistance and not necessarily pseudorandomness, as needed here.

We do not try to prevent against users informing the cloud of when a block should be audited (and thus, the cloud misbehaves only when a block is not to be audited). Auditing is to make sure that the users get correct data and their puts are successful. If they want to change the data, they can do so using their access permissions and do not have to collude with the cloud for this. As mentioned before, the owner should ensure (via other means) that users do not use their access permissions on behalf of unauthorized people.

When a block is meant to be audited, clients send the owner the attestations they receive from the cloud. Clients do not need to store attestations. The owner separates these attestations by block and sorts them by version number. This generally requires little processing since the attestations will arrive in approximately this order. The attestations for the same version number are sorted such that each two consecutive attestations satisfy Equation (1); we envision that the cloud could attach a sequence number to the attestations to facilitate this sorting. If some clients do not send attestations because they fail or are malicious, they have no guarantees on whether they read correct data or their put got committed. All clients sending attestations will have such guarantees. The clients cannot frame the cloud by not sending attestations. The owner will ask the cloud to provide cloud attestations for any missing attestations in the sequence and an honest cloud will keep copies for the duration of an epoch. Alternatively, the cloud can only keep copies of the attestations' data without signatures, which it can reconstruct on-demand, thus storing less. If the cloud cannot provide these, the cloud is penalized for non-compliance with the auditing procedure.

Once the owner has the complete sequence of attestations, it performs checks for write-serializability and freshness, as we describe in the subsections below. After auditing, the owner and the cloud create a Merkle hash tree of the entire storage, exchange attestations that they agree on the same Merkle value, and *discard all attestations or attestation data* from the epoch that just ended. The Merkle hash can be computed efficiently using the hashes of the blocks that were modified and the hashes

of the tree roots of the blocks that were not modified in this epoch. The owner updates the family key block if there were any membership changes.

Interestingly, we will see that auditing only makes use of public keys. As such the owner can outsource the auditing tasks (e.g. to a cloud competitor).

Due to space constraints, the presentation of the theorems and proofs below is in high-level terms, leaving a rigorous mathematical exposition for a longer paper.

## 7.1 Write-serializability (W)

*Requirement on the cloud*. During the epoch, the cloud is responsible for maintaining the write-serializability of the data. That is, the cloud must make sure that every put advances the version number of the most recently stored block by exactly one. If a client provides a put for an old version number, the cloud must inform the client of such conflict. It is now the client's choice to decide what action to take: give up on his own change, discard the changes the cloud informs him of, merge the files, etc.

One attack on write-serializability is a *fork attack* (introduced in [20]). The cloud provides inconsistent views to different clients, for example, by copying the data and placing certain writes on the original data and other writes on the copy. If two clients are forked, they will commit two different updates with the same version number on the cloud.

*A correct write chain of attestations* is a chain of put cloud attestations where there is exactly one put for every version number between the smallest and the largest in the sequence. Moreover, the smallest version number must be one increment larger than the version number of the block at the beginning of the epoch.

*Detection of W Violation:* The sequence of put attestations do not form one correct write chain.

The following theorem clarifies why this statement holds. It assumes that users send all the write attestations they get from the cloud to the owner. If they do not send all these attestations, the theorem holds for every complete interval of version numbers for which attestations were sent.

**Theorem 1.** *The cloud respected the write-serializability requirement for a block in an epoch iff the cloud's put attestations form one correct write chain.*

*Proof.* First, let us argue that write-serializability implies one chain. If the cloud respects write-serializability, it will make sure that there are no multiple writes for the same version number and no version number is skipped; therefore, the attestations form a correct chain.

Now, let us prove that one chain implies write-serializability. A violation of this property occurs when a client performs an update to an old version of the data. Suppose the current version of the data on the cloud is $n$ and the client is aware of $m < n$. When the client places a put, the version number he uses is $m+1 \leq n$. Suppose the cloud accepts this version and provides a cloud attestation for $m+1$. Since $m+1 \leq n$, another put with version $m+1$ committed. If that put changed the block in a different way (thus inducing a different new hash in the attestation), the owner will notice that the attestations split at $m+1$. □

Note that for W, the auditor does not check chain hashes and thus it does not need get attestations.

*Proof of Violation of W:* The broken sequence of write attestations as well as the cloud attestation for the current family key block.

This constitutes a proof because cloud attestations are unforgeable. A client cannot frame an honest cloud. A proof of violation consists of attestations signed by the cloud; thus the client cannot change the contents of the attestations and create a broken sequence.

## 7.2 Freshness (F)

*Requirement on the cloud*. During the epoch, the cloud must respond to each get request with the latest committed put content and compute chain hashes correctly (based on the latest chain hash given) for every cloud attestation.

A correct chain of attestations is a correct write chain with two additional conditions. First, the hash in each read attestation equals the new hash in the write attestation with the same version number. Second, the chain hash for an attestation and the chain hash of the previous attestation in the sequence satisfy Eq. (1).

*Detection of F Violation:* The attestations do not form one correct chain.

**Theorem 2.** *The cloud respected the freshness requirement iff the attestations form one correct chain.*

*Proof.* It is easy to see that if the cloud respected the freshness requirement, the attestations will form one chain. Each attestation will be computed based on the latest request.

Let us show that if the freshness requirement is violated, we do not have a correct chain. We proceed by contradiction assuming that we have a correct chain. There are two key points we will use. The first is that each chain hash the cloud gives to a client is dependent on some randomness the client provides. This is the random nonce for get or the new hash for put. The cloud cannot compute the chain hash before it has this randomness. The second point is that the value of a chain hash recursively depends on all the history of chain hashes before it.

Let $A$ be an attestation, and $B$ the attestation preceding it in the chain. Assume the cloud violated the freshness requirement when answering the request corresponding

to $A$, but the attestations form a correct chain for contradiction purposes. Thus, $B$ was not the latest request performed before $A$; instead, let $C$ be this request. Thus, the cloud gave out attestations $B$, $C$, and $A$ in this order. $C$ must come somewhere in the correct attestation chain. It cannot come after $A$ because the chained hash of $C$ will have to depend on the chain hash of $A$. Due to the client-supplied randomness and hardness assumptions on the hash function (random oracle), the cloud can compute the chain hash for $A$ only when he gets the $A$ request which happens after the $C$ chain hash was given out. If $C$ comes before $A$ in the sequence, it must come before $B$ because we assumed that the freshness requirement was violated at the time of this request. This means that $B$'s chain hash depends on $C$'s chained hash, which is not possible because the cloud would not know $C$'s client-provided randomness when he has to answer to $B$. Therefore, we see that the chain cannot be correct when freshness is violated. □

*Proof of Violation of F:* The broken sequence of attestations as well as the cloud attestation for the current family key block.

A client cannot frame an honest cloud. A proof of violation consists of attestations signed by the cloud; thus, the client cannot change the contents of the attestations and create a broken sequence.

### 7.3 Discussion

For efficiency, many storage servers allow gets to proceed in parallel, while serializing puts. CloudProof also allows the bulk of the get operations to happen concurrently, and serializes only the computation of the chain hash for gets to the same block. For example, consider multiple concurrent gets for the same block. These gets can proceed in parallel (e.g., contacting nodes in the data center, performing disk accesses, preparing result); the cloud only needs to make sure that updates to the chain hash for the same block do not happen concurrently. Updating a hash takes very little time, so the loss in concurrency is small.

CloudProof protects W, F, and I with respect to each block in part. In some cases, one might want these properties to be satisfied with respect to a collection of blocks, for example, so that different blocks are updated consistently. CloudProof can satisfy this requirement by viewing the block collection of interest as one block and having one attestation for this collection of blocks. CloudProof will serialize accesses to this collection of blocks in the same way as for one block.

The protocol for freshness requires the cloud to update a chain hash and store some metadata upon any get operation, so a malicious client unauthorized to read the data can DoS a server computationally and storage-wise. A solution to this attack is to require each client to provide a correct "client get attestation" signed with the read access key to the cloud when performing a get. An evaluation of this scenario is future work.

One might wander if CloudProof protects against the server forking the view of users over two different epochs, while not attempting any fork entirely in any of the two epochs. Indeed, CloudProof detects this attack if the block is audited in the two epochs; the owner will detect the fork in the later epoch as follows. Consider that a block is at version 1, a user modifies it to version 2 in epoch 1 and then, in a later epoch, say epoch 3, another user updates the block; during the second update, the server is malicious and does not inform the second user of the update of user 1 and allows the second user to update to version 2 as well. Recall our Merkle hash checkpoint of the entire storage described in Section 7; after auditing the block in epoch 1, at the end of the epoch, the owner will record in the Merkle hash tree that this block was at version 2 at the end of epoch 1 because it received the attestation from the first user. Therefore, the owner will expect the block to be at version 2 at the beginning of epoch 2 and also at the beginning of epoch 3 (because no operation was performed on the block in epoch 2). During auditing of epoch 3, the owner sees a cloud attestation for version 2 that has a different hash from the one the Merkle tree checkpoint records, and thus discovers the fork.

## 8 Implementation

We implemented CloudProof on top of Microsoft's Windows Azure [22] cloud platform. Our implementation consists of about 4000 lines of C#. CloudProof only relies on a get/put and sign/verify interface from the cloud, which makes it easy to adapt to other cloud systems.

**Background on Azure.** First, we give the needed background on Azure [32]. Azure contains both a storage component and a computing component. CloudProof uses the blobs and queues storage services. Blobs are key/value stores mapping a blob identifier to a value. Queues are used for reliable messaging within and between services.

The compute component consists of *web* and *worker* roles, which run on virtual machines with different images of Windows Server 2008. The web roles are instances that can communicate with the outside world and receive HTTP requests. The worker roles are internal running instances which can communicate with the web roles (via storage nodes) and access storage. There is no guarantee whether the worker, web or storage nodes are collocated on the same machine. Furthermore, the storage and compute components are provided as different services so they may be located in different data centers, although Azure allows us to specify an affinity group for

our storage and compute nodes.

CloudProof consists of four modules. The *data user/client* is a client-side library that exports get and put interface. A data user uses this library to perform get and put calls to the cloud. It exchanges blocks and attestations with the cloud. The *cloud* runs on top of Azure and responds to get and put requests and exchanges attestations with the client. The *data owner/enterprise* runs on a data owner's premise. This is a library to be used by the data owner. It serves to add or remove permissions to users or groups and for auditing. It interacts with the cloud component to update family key blocks. If the data owner wants to get or put data blocks, it needs to create a client instance for itself. As mentioned, the owner can outsource the auditing task to another party, the *auditor*. It collects attestations and audits them according to the algorithms in Section 7.

Let us explain how a request processing proceeds. The clients and the owner send HTTP requests to the web roles, which then place the requests in a queue, together with a blobID. The workers poll the queue for requests to process. The worker roles place the response into the blob with the given blob ID. Web roles poll response blobs periodically to get the reply for the request they made.

The cryptographic algorithms used are the .NET implementations of SHA-1 for hashing, AES for symmetric encryption and 1024 bit RSA for signing. Any of these schemes can be easily substituted with more secure or faster ones.

## 9 Evaluation

In this section, we evaluate the performance of CloudProof. We investigate *the overhead CloudProof brings to the underlying cloud storage system without security* to establish whether the security benefits come at a reasonable cost. We are not interested in examining or optimizing the performance of Azure itself because CloudProof should be applicable to most cloud storage systems.

CloudProof targets large enterprise storage systems, source code repositories, and even small, personal file systems. These tend to be applications that can tolerate larger client-cloud latency (which is an inherent result of the different geographic locations of various clients/organizations with respect to the cloud).

The design of CloudProof allows *separability to five security modes*, each corresponding to the addition of a new security property: no security, just confidentiality (C), confidentiality and integrity (CI), the previous two with write-serializability (CIW), and full security (CIWF). We will see how performance is impacted by adding each security property. For these experiments, the client-side machine is an Intel Duo CPU, 3.0 GHz, and 4.0 GB RAM.

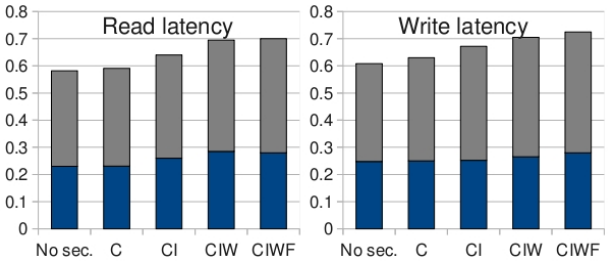

Figure 3: End-to-end and effective read and write latency. Each bar indicates the end-to-end latency incurred, with the lower section showing the effective latency.

**Microbenchmarks.** We observed that a large factor of performance is given by the latency and bandwidth between the client and the cloud's data center (which seems to be on the other coast from the data we get), which do not depend on our system. Therefore, we will also include "effective" measurement results which are computed by subtracting the client-cloud round-trip time from the end-to-end latency.

To evaluate latency, we perform 50 reads and 50 writes to different blocks (4 KB) and compute the average time per operation. Figure 3 shows our latency results. We can see that the overhead increases with every addition of a security requirement. The overhead added by W is caused by the creation, verification and handling the attestations; adding a chain hash for F causes a small overhead increase. The overhead of our scheme (CIWF) as compared to the no security case is $\approx 0.07s$ which corresponds to 17%. With respect to confidentiality only (minimal security), CIWF is $\approx 14\%$.

Let us understand which components take time. Each request consists of client preprocessing (0.5%), network round trip time between server and client (36%), cloud processing (62%), and client post-processing (1.5%). We can see that most time is spent at the server. This is because the web and worker roles communicate with each other using the storage nodes, and all these nodes are likely on different computers. We specified affinity groups to colocate nodes close to each other, but we do not have control over where they are actually placed.

CloudProof introduces a delay of 70 ms per operation at the client and server in total, out of which $\approx 30$ ms is at the server. This delay consists mostly of the overhead of signing and verifying attestations. Each time a client or the cloud issue an attestation, they have to sign it and the other party has to verify it. Besides sign-verify operations, the other operations performed, comparisons and hashes, impose very small overhead.

*Batching multiple attestations in one signature at the server.* In our current implementation, the server performs one sign operation for each attestation. However, if one wants to reduce the work of the server per attestation (30 ms), one can optimize our implementation by

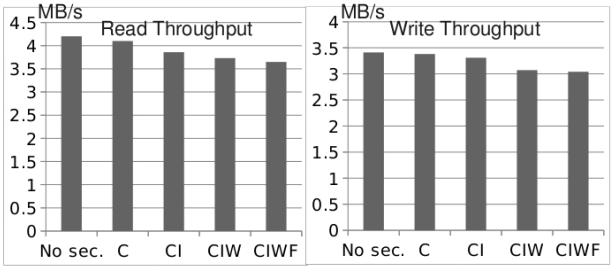

Figure 4: Effective read and write throughput.

having the server batch multiple attestations together and sign them all with only one sign operation. Here is how the server can perform such batch signing. Consider that the server has $k$ attestations, $A_1, \ldots, A_k$, that are supposed to be returned signed to $k$ users, $U_1, \ldots, U_k$ (not necessarily different), after performing only one signature on the batch. The server can compute a Merkle hash on the $k$ attestations, MH. Hashing is a fast operation. The server then signs MH to obtain sig(MH). Finally, the server returns the following information to user $U_i$: the attestation $A_i$, all the hashes in the Merkle tree relevant to $A_i$ (all the hashes from $A_i$ up to the root, MH, inclusive, and all the siblings of these hashes) and sig(MH). The user verifies this signature by making sure that the hash of $A_i$ and the other relevant hashes yield MH, as well as that the signature verifies to be a signature by the cloud on MH. Due to the properties of the Merkle hash, this approach is as secure as if the cloud signed each attestation $A_i$ individually. Bandwidth does not increase significantly because the additional data the cloud provides to the client are hashes (128 bits long) and there are only $2\log k$ of them.

In this manner, the cloud can batch together any number of attestations it has available for signing while paying the cost of only one signature. For example, if the cloud batches together 10 attestations, the overhead of the server per attestation becomes less than 5 ms.

Figure 4 shows the effective throughput incurred in our system. The graphs are generated by reading and writing a large file (sent to one cloud worker), 50 MB, and dividing its size by the effective duration of the request. We generated effective measurements because the throughput between the client and the cloud was a bottleneck and the results were similar for and without security (almost no overall overhead). We can see that the throughput overhead (due to cryptographic operations mostly) for write is 11% and for read is 12%, which we find reasonable.

An important measure of scalability is how the system scales when workers are added. We obtained a VIP token for Azure that allows creation of 25 workers (given the early stage of Azure deployment, it is hard to get more workers for free). We spawn 25 clients that can run in parallel and each such client sends two requests to the cloud (a request is sent only after the previous request finished); this experiment is repeated many times. We measure how many pairs of requests are satisfied per second as we increase the number of worker roles. Figure 5 shows our results. We can see that each pair of requests is satisfied in about 1s which makes sense given the latency results presented above. We can see that the scaling is indeed approximately linear. This makes sense because we designed our security protocols to allow all requests to proceed in parallel if they touch different blocks.

Moreover, for scalability, our access control scheme should be able to support efficiently many members and blocks. Our scheme has the benefit that revocation and addition do not depend on the number of blocks in a family. It only depends on the square root of the number of members with access to the block family. As we can see in Figure 7, the square root factor is almost irrelevant; the reason is that the constants multiplying the square root are much smaller than the constant latency of read and write. We can see that for large enterprises with $100,000$ employees, group addition and revocation remain affordable even if every single employee is present on an access control list.

A block family is created when a block's ACL is changed to an ACL that no other block has. The owner must put a new family key block on the cloud. If the membership change from the previous ACL (if one existed) to the current one is not large, the owner can just copy the family key block for the previous ACL and perform the appropriate revocations and additions. Otherwise, the owner should choose new keys, compute broadcast encryptions to the new members, and put the resulting data on the cloud. For an ACL of 1000 new members, these operations sum up to $\approx$ 20s. Any blocks added to the block family will not require key block family changes. The block just needs to be encrypted with the proper key.

Figure 6 shows the performance of auditing. For example, it takes about 4 minutes to verify $10^8$ attestations. This number corresponds to a large company of $10^5$ employees each making about 1000 changes in an epoch of a month. Furthermore, auditing can be easily parallelized because it just involves checking each consecutive pairs of attestations in a chain so the chain can be split in continuous sections, each being distributed to a different core. In fact, auditing can be performed on an alternate cloud, which, due to market competition, will have incentive to run the audit correctly. The duties of the owner are just auditing and membership changes, so we can see that, even for enterprise storage, a lightly used commodity computer would be enough to handle all of the owner's duties.

**Storage Overhead.** The total overhead per block in CloudProof is 1120 bits if we use RSA signatures

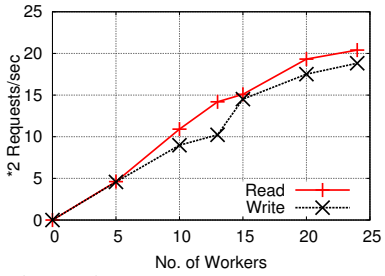


Figure 5: Pairs of requests executed per second at the cloud as depending on the number of workers.

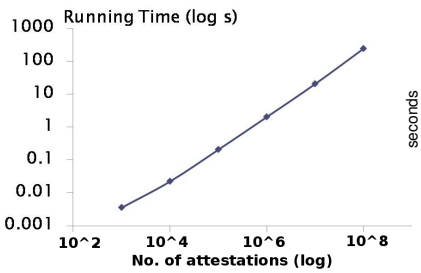


Figure 6: Auditing performance for one owner node.

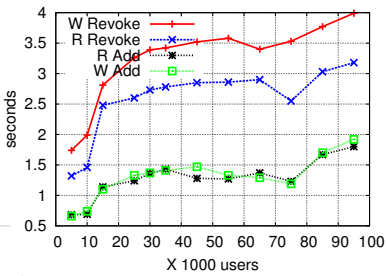


Figure 7: The duration of a membership update as depending on the number of members with access to a block family. R denotes read and W denotes write.

or 256 bits if we use short signatures (Boneh-Lynn-Shacham). A block in Azure may be arbitrarily large, so this overhead may be a small percentage of large blocks. The family key block consists of $1120 + 1024 * \sqrt{n} + 1024 * n$ bits, where $n$ is the number of users in a family. Had we used a more efficient broadcast encryption scheme, the linear factor would be removed. In a commercial source repository trace (presented below), we found that the maximum value of $\sqrt{n}$ was 14 and that there were 30 total groups. Therefore, the storage overhead for family key blocks is less than 26.4 KB for the lifetime of the trace. All attestations are about 1300 bits (or about 400 bits with a short signature). The cloud only needs to keep the latest put client attestation for each block and the unsigned attestation data ($\approx$ 258 bits) for all attestations in an epoch for auditing purposes.

**Macrobenchmarks.** To determine how many users are in real enterprise groups and to understand the frequency of access control changes, we obtained traces of group membership changes in a version control repository for a very large widely used commercial software (whose name we cannot disclose) that has more than 5000 developers. From these traces, we computed the dates of all *revocation events*: changes to group membership where at least one member was deleted from the group. For each event, we computed the size of the group after deletions of members. As described, the square root of this group size controls the time required to compute new keys for the block family. As we showed in Figure 7, our system can compute new keys for groups of 100,000 users in less than 4 seconds. In particular, we found that computing keys for all revocation events in a month, assuming all groups in our trace have 250 members, took an average time of less than 1.6 seconds. This shows that our system is capable of easily handling the group sizes and frequency of revocations in this real application.

We looked at the commit histories for two large open source projects hosted on Github: Ruby on Rails and Scriptaculous. Both projects are widely used and under active development. Source code repositories require integrity guarantees: adversaries have broken into such repositories in the past to corrupt software which is then widely distributed. Distributed development also benefits from F and W.

| | week max | week avg | month max | month avg |
|---|---|---|---|---|
| RoR | 55.7 | 9 | 93 | 38 |
| ST | 4.2 | 0.91 | 8.2 | 2.4 |

Table 1: Maximum and average storage requirements in KB per epoch for all epochs with at least one client put request. RoR stands for "Ruby on Rails," while ST stands for "Scriptaculous."

To create benchmarks, we looked at the history of all commits to all files in Ruby on Rails for six months from July 2009 to November 2009, and all files in Scriptaculous for one year from November 2008 to November 2009. Reads are unfortunately not logged in the traces; however, from microbenchmarks, we can see they have a similar overhead to writes. We used the history provided in each repository to identify the size of each file after each commit and the identity of the committer. We then replayed this history, treating each commit as a separate put whose key is equal to the file name and with a value of the appropriate size. Figure 8 shows the results for the both traces. The overall overhead for Ruby on Rails is 14% and for Scriptaculous is 13% for providing all security properties. These results show that we can achieve our properties with modest overhead, and that our microbenchmarks are a good predictor of our overhead for these applications.

For storage overhead, we first computed the number of distinct files in each repository in the state it was at the end of the trace. For Ruby on Rails we find 5898 distinct files, totaling 74.7 megabytes. At 1120 bits of overhead per file, CloudProof requires 806 KBs of storage for metadata, an overhead of roughly 1.1%. For Scriptaculous we find 153 distinct files, totaling 1.57 MB, yielding a storage overhead for CloudProof of 1.3%.

We then evaluated the cloud storage required to hold attestations from clients at different epoch lengths for both traces. We considered the extreme case where the epoch is equal to the duration of the trace: i.e., the cloud keeps all attestations from all clients. For the Ruby on Rails trace we have 71241 total client puts, which requires 2.2MB of storage on the cloud for 258 bits per

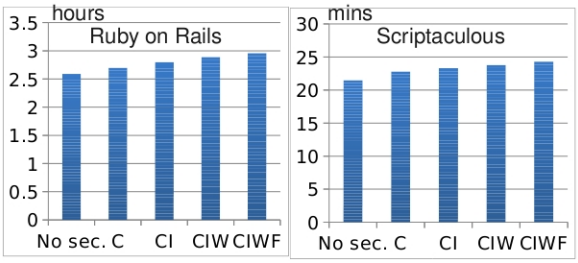

Figure 8: Runtime for macrobenchmarks.

attestation. For the Scriptaculous trace, we have 2345 total client puts, which requires 73KB of storage. We then looked at two cases: the epoch is one week and 30 days. Table 1 shows the results. We see that the amount of storage required for attestations in both traces is low, under 100KB, even for a long epoch time of 30 days.

The choice of epoch is a tradeoff between cloud storage overhead and audit time. Our trace results show that for the source repositories we considered, even an epoch length of six months to a year requires modest overhead. Shorter epochs require less than 100 kilobytes of storage. This appears reasonable, especially given that the storage for attestations may be append-only and need not be highly available. Other applications, of course, may require more puts during an epoch and therefore incur more overhead for storage. With our current implementation, however, one gigabyte of storage is sufficient to hold attestations for over 33 million puts; we believe this should be sufficient to allow a reasonable epoch for most applications. As we showed, the time to audit is not a gating factor for the length of an epoch, because audits can be carried out in a short amount of time.

From the results above, we conclude that CloudProof has a reasonable overhead and is practical for real storage applications with security needs.

## 10 Related Work

The related work can be divided in four parts.

**Existing Cloud Systems**. Today, there exist a plethora of cloud storage systems (Amazon S3, Google BigTable, Microsoft Azure, Nirvanix CloudNAS, etc.); however, these systems do not guarantee security. The cloud system that provides most security, to the best of our knowledge, is Tahoe [31]. It is a peer-to-peer file system that allows users to check data integrity and provides access control. However, Tahoe does not detect violations to W and F, nor does it provide proofs of violation.

**Secure File and Data Storage Systems** include related systems [4], [3], [10], [18], [20]. In comparison to these systems, CloudProof brings the three main contributions listed below. These systems do not provide these properties, not because of shortcomings in their design, but because they were designed for a different setting than the cloud: mostly for personal storage hosted on some remote untrusted and not liable servers. We also argue that they are not easily extendable to achieve our desired properties, which illustrates that there was a need for a new design for the cloud setting.

(1) They detect, but *do not prove* server misbehavior and *do not keep the client accountable to the server*. CloudProof provides such guarantees to enable a financial contract between the client and the cloud regarding security.

(2) CloudProof maintains the *high scalability* of the cloud because it is a crucial aspect of the cloud promise and large enterprises are important customers of the cloud. Our access control scheme is especially scalable.

(3) CloudProof provides detection (and proofs) to both *write-serializability and freshness*. This is because an enterprise affords to dedicate a compute node for auditing and membership changes. Most previous work cannot detect violations to both properties.

SiRiUS [10] is perhaps the most related previous work to CloudProof. In SiRiUS, each user stores his file system on a remote untrusted server and can detect integrity violations to his data. SiRiUS does not guarantee write serializability: two users can read a file at the same time, place updates subsequently with the second user ignorantly overwriting the first user's update. SiRiUS does not offer freshness as we define it in Section 2. It offers a weaker type of F: fetched data can be stale if it is not older than a certain time interval. Furthermore, SiRiUS does not scale to enterprise sizes. Each user has a separate file system that also contains files from other user's file systems for which the user has write access. When checking the freshness of a file upon read, users need to check the file system of each user with write access, verify a Merkle tree for that access, and decide who has the newest version. In an enterprise setting, some files can be accessed by thousands of users so this approach would not scale. Moreover, to ensure freshness, SiRiUS requires each user to re-sign the top of his Merkle hash every few minutes or seconds. This is not a reasonable assumption we believe, because the users are not necessarily online all the time. Moreover, in SiRiUS, users can defeat their access permissions. Unauthorized users can delete data or seize unattained permissions by overwriting the metadata of some files with their own. These attacks occur because Sirius's premise is not to change the server software and allow it to run on any remote file system implementation such as NFS. In contrast, cloud providers can always install software of their choice on their nodes if they want to provide security. Finally, SiRiUS does not provide proofs needed in our cloud setting.

Plutus [18] uses key rolling, but every time a user realizes that the key has changed, he contacts the owner to ask the key. This approach would demand more resources from an enterprise to satisfy the requests of all their employees. We combine key rolling with broadcast encryption and the notion of block families to achieve

key distribution without involving the owner/enterprise.

SUNDR [20] is a secure file system that offers fork consistency by having clients check histories of snapshots of file versions (VSLs). First, SUNDR does not provide read access control and it is not clear how to enhance it with scalable read access control. Our key distribution protocol uses broadcast encryption and key rolling to support fast and scalable read access, and handles user read access revocation efficiently. Second, a straightforward addition of W to SUNDR, would be unscalable. If SUNDR sends all VSLs to the owner for auditing, SUNDR could achieve write-serializability easily because the owner would notice a fork attack. However, each VSL entry includes version numbers for all files in the file system, and there can be many such files. In contrast, CloudProof's attestations only contain information about the block accessed. On the other hand, one positive aspect of SUNDR in this regard is that it can order updates across files, whereas CloudProof can only order updates within a file. We made this tradeoff for scalability. Third, CloudProof provides improvements regarding freshness over SUNDR. Even with auditing at the owner, SUNDR would not achieve freshness for clients that are only readers. The server can give a client a prefix of the current history, thus not informing him of the latest write. Since the reader will not perform a write, a fork will not occur. SUNDR can be easily extended to provide proofs of integrity violation, but providing freshness violation proofs seems harder. Finally, SUNDR does not scale to enterprise-sizes because of the long history chain of signatures that clients must check for every fetch. For highly-accessed files or many users and files, version snapshots can grow large and many.

**Cryptographic Approaches**. Kamara and Lauter [19] investigate cryptographic techniques useful in the cloud setting. They mention proofs of data possession or retrievability (POR) [17], [2], [25], [9], which allow a server to prove to the owner of a file (using sublinear or even constant space usage) that the server stores the file intact (the file has integrity) and can retrieve it. HAIL [6] allows a distributed set of servers to prove file integrity and retrievability. [29] allows updates and enables public verifiability of the integrity of the data at the cloud.

Such work is very useful for proving integrity of archived data; however, it is not sufficient as a holistic cloud systems solution. The fact that the file is correct and retrievable on the cloud does not mean that the cloud will respond with correct data upon a request. For example, suppose that the user requests the cloud to prove that the file is intact and the cloud successfully does so. Then, a few users request various blocks of the file; the cloud can return incorrect data (junk or stale) and there is no mechanism in place for checking this. Having each client ask the cloud for a POR before a get is too expensive. Also, most of these schemes deal with archived data and are not efficient on updates. They either require some non-negligible overhead of preprocessing when placing a file on the server [17] or that all updates be serialized [29] and thus have poor scalability. Moreover, they do not provide write-serializability or freshness (and thus, no proofs of violations for these properties). For instance, the cloud can overwrite updates and retrieve stale data because the integrity checks will not fail. Lastly, these schemes do not provide access control and are not designed for many concurrent accesses by different users.

**Byzantine Fault Tolerance** (e.g., [7]) proves correctness of query execution at remote server replicas given that the number of Byzantine (faulty, malicious) replicas is at most a certain fraction. However, this approach is not applicable to the cloud setting because all the nodes in a data center belong to the same provider. If the provider is malicious, all the nodes are malicious. Furthermore, most nodes are likely to be collocated geographically and run the same distribution of software and likely crash from similar factors. One idea is to use BFT with multiple cloud providers. This approach will indeed decrease the chance of security problems; however, clients will have to pay all the cloud providers, and, if the data gets lost at all parties, the client has no remuneration assurance. Chun et al. [8] also uses the concept of chained attestations, which they store in a trusted-hardware log; their goal is to prevent equivocation by Byzantine clients and ultimately improve performance of commit.

**Secure Audit Trails and Logs**. Research in secure digital audits aims to verify the contents of a file system at a specific time in the past. For example, in [24], a file system commits to the current version of its contents by providing a MAC on its contents to a third-party. At a later time, an auditor can check that the file system still contains the old version using the MAC token.

There has also been work on secure logging [30], [26]. In this work, a trusted machine writes encrypted logs that cannot be read or modified undetectably by an outsider. This work does not consider a large number of users concurrently accessing the data, there is no read and write access control (one key allows both read and write), the log is typically just appendable and it is not optimized for writing in the middle, and a malicious outsider manipulating the order in which updates and reads are performed on the logging machine can compromise W and F.

Moreover, in all this work, the owner cannot convince a third party of some security violation.

## 11 Conclusions

We propose proofs of security violations for integrity, write-serializability and freshness as a tool for guaranteeing security in SLAs. We build a secure cloud storage system that detects and proves violations to these prop-

erties by combining cryptographic tools in a novel way to obtain an efficient and scalable system. We demonstrate that CloudProof adds reasonable overhead to the base cloud service.

## References


[1] AMAZON. Amazon s3 service level agreement, 2009. http://aws.amazon.com/s3-sla/.

[2] ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z., AND SONG, D. Provable data possession at untrusted stores. In *ACM CCS* (2007).

[3] BINDEL, D., CHEW, M., AND WELLS, C. Extended cryptographic filesystem. In *Unpublished manuscript* (1999).

[4] BLAZE, M. A cryptographic file system for unix. In *ACM CCS* (1993).

[5] BONEH, D., GENTRY, C., AND WATERS, B. Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys. *Lecture Notes in Computer Science* (2005).

[6] BOWERS, K. D., JUELS, A., AND OPREA, A. HAIL: A High-Availability and Integrity Layer for Cloud Storage. *CCS* (2009).

[7] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS* (2002).

[8] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested Append-Only Memory: Making Adversaries Stick to their Word. *SOSP* (2009).

[9] DODIS, Y., VADHAN, S., AND WICHS, D. Proofs of Retrievability via Hardness Amplification. *TCC* (2009).

[10] E.-J. GOH, H. SHACHAM, N. M., AND BONEH, D. Sirius: Securing remote unstrusted storage. In *NDSS* (2003).

[11] FERDOWSI, A. S3 data corruption?, 2008. http://developer.amazonwebservices.com/connect/thread.jspa?threadID=22709&start=0&tstart=0.

[12] FIAT, A., AND NAOR, M. Broadcast encryption. *Proc. of Crypto* (1993).

[13] FU, K. Cepheus: Group sharing and random access in cryptographic file systems. Master's thesis, MIT, 1999.

[14] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to Construct Random Functions. *JACM* (1986).

[15] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

[16] Cloud security still the biggest concern/hurdle for google, microsoft, verizon. www.taranfx.com/blog/.

[17] JUELS, A., AND KALISKI, B. PORs: Proofs of retrievability for large files. In *ACM CCS* (2007).

[18] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *USENIX FAST* (2003).

[19] KAMARA, S., AND LAUTER, K. Cryptographic cloud storage. In *ACM Workshop on Cloud Security* (2009).

[20] LI, J., KROHN, M., MAZIERES, D., AND SHASHA, D. SUNDR: Secure untrusted data repository. In *OSDI* (2004).

[21] MICALI, S., RABIN, M. O., AND VADHAN, S. P. Verifiable Random Functions. In *IEEE FOCS* (1999).

[22] MICROSOFT CORPORATION. Windows Azure. http://www.microsoft.com/windowsazure.

[23] MICROSOFT CORPORATION. Windows Azure Pricing and Service Agreement, 2009. http://www.microsoft.com/windowsazure/pricing/.

[24] PETERSON, Z. N. J., BURNS, R., AND ATENIESE, G. Design and Implementation of Verifiable Audit Trails for a Versioning File System. *USENIX FAST* (2007).

[25] SACHAM, H., AND WATERS, B. Compact Proofs of Retrievability. *ASIACRYPT* (2008).

[26] SCHNEIER, B., AND KERSEY, J. Cryptographic Support for Secure Logs on Untrusted Machines. *USENIX Security* (1998).

[27] SION, R., AND CARBUNAR, B. On the computational practicality of private information retrieval. In *NDSS* (2007).

[28] THOMSON, I. Google docs leaks private user data online, 2009. http://www.v3.co.uk/vnunet/news/2238122/google-docs-leaks-private.

[29] WANG, Q., WANG, C., LI, J., REN, K., AND LOU, W. Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing. *ESORICS* (2009).

[30] WEATHERSPOON, H., EATON, P., CHUN, B.-G., AND KUBIATOWICZ, J. Antiquity: Exploiting a Secure Log for Wide-area Distributed Storage. *ACM SIGOPS* (2007).

[31] WILCOX-O'HEARN, Z., AND WARNER, B. Tahoe - the least authority file system, 2008. http://allmydata.org/~zooko/lafs.pdf.

[32] WINDOWS AZURE PLATFORM. Azure SDK, 2009. http://msdn.microsoft.com/en-us/library/dd179367.aspx.