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
Motivation

Cloud storage provides extensive resources, scalability, and reliability

A main concern is security
  - Data leakage/corruption due to bugs, hackers, employees
  - Many customers perceive security as main concern
Security properties

- Confidentiality (C): only authorized users can read data
- Integrity (I):
  - Each get returns the content put by an authorized user
- Write-serializability (W):
  - Each user committing an update is aware of the latest update to the same block
- Freshness (F):
  - Each get returns the data from the latest committed put

Problem: cloud services do not guarantee security in SLAs

Need proofs of misbehavior
A secure storage system for the cloud:

1. Security mechanisms needed for SLAs with security:
   - Detection of violations for integrity, write-serial., and freshness (IWF)
   - Publicly-verifiable proofs of violation for IWF
     - Any external party can be convinced of cloud misbehavior
     - Users cannot falsely accuse cloud

2. Scalable design of security mechanisms
   - Scalable access control using modern cryptographic tools
Model

Data owner
- assigns permissions to users (R, RW)
- may try to frame the cloud

Data users
- may attempt to bypass permissions

Application

Cloud
- fully untrusted

get/put blocks
For each block:

- **Confidentiality**: Owner gives a secret key for encryption, $sk$, to allowed readers.
- **Integrity**: Owner gives public key pair for signing, $SK$, $PK$ to allowed writers.

**Problems:**

- No detection for write-serial., freshness
- No proofs of violation
- Access control/key distrib. not scalable

*In this talk*

*See paper*
Detection and proofs of violation for IWF

- **Attestations**
  - User
  - Cloud
  - `get(block id)`
  - `block content, cloud-get-attestation`
  - `put(block id, content), client-put-attestation`
  - `cloud-put-attestation`

- Proofs verifiable by any outside party
- Non-repudiable signature scheme [Micali et. al., ’99]
- Each party verifies attestation signatures
Auditing

- **Integrity**: users check attestations from cloud

- **W and F**: Owner does probabilistic auditing
  - Time divided in epochs (e.g., day)
    - Only owner and authorized users know in which epochs a block is audited

<table>
<thead>
<tr>
<th>Block</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>0.5</td>
</tr>
<tr>
<td>B2</td>
<td>1</td>
</tr>
<tr>
<td>B3</td>
<td>0.1</td>
</tr>
<tr>
<td>B4</td>
<td>0.2</td>
</tr>
<tr>
<td>B5</td>
<td>0.2</td>
</tr>
</tbody>
</table>
During the epoch

Users → Cloud

B2

... , cloud-get-attestation

get..

put ...

cloud-put-attestation

Users → Cloud

B4

... , cloud-put-attestation

put..

put ...

cloud-put-attestation

Data owner

cloud-get-attestation
cloud-put-attestation
cloud-put-attestation
cloud-put-attestation
At the end of epoch

- For the blocks to audit:
  - Owner requests all cloud-attestations from the cloud
  - Audits attestations from clients and from cloud
  - Audit guarantees write-serial. and freshness for entire epoch
## Attestation Structure

<table>
<thead>
<tr>
<th>Block ID</th>
<th>Version NO.</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLOUD GET ATTEST.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CLOUD PUT ATTEST.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>USER PUT ATTEST.</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Hashed and signed by cloud
- Hashed and signed by user with SK
- Hashed and signed by cloud
Integrity

Hashed and signed by cloud

“CLOUD GET ATTEST.” | BLOCK ID | BLOCK VERSION NO. | BLOCK HASH | ...

Hashed and signed by cloud

Block

\( \text{Enc}_{sk}[\text{content}] \) version no.

\( \text{Sig}_{SK}[\text{encr. content}] \)

- Detection: signature does not verify
- Proof of violation: attestation
Write-serializability

- **Detection**: Fork in sequence of put attestations
- **Proof of violation**: the forked sequence of attestations

Hashed and signed by cloud

- version 5, hash: xae97
- version 5, hash: x3166
- version 4, hash: xd242
Freshness

- \( \text{chain hash} = \text{hash (data in current attestation, previous attestation)} \)
- **Detection:** attestations do not chain correctly
Detection: attestations do not chain correctly

A1 = (cloud-put-attestation, blockid 5, version 1, hash x18, …)

A2 = (cloud-put-attestation, blockid 5, version 2, hash x22, h(A1, data in A2), …)

A3 = (cloud-get-attestation, blockid 5, version 1, hash x18, h(A2, data in A3)? Detected!

Proof of violation: broken chain of attestations
Implementation

- C#, Windows Azure:
  - Storage component: blobs and queues
  - Compute component: web and worker roles

- Four modules: owner, user, cloud, auditor
- .NET crypto tools: AES, SHA-1, RSA
Evaluation

- What is the overhead at users/cloud?
  - Latency/throughput
- What is the workload of the owner?
  - Access control/auditing
User/server overhead

- Mostly from sign-verify of attestations

- **Delay added per request:** 30 ms at server, 40 ms at user
- Can optimize: e.g., batch many attestations in one signature using a Merkle hash
- **Throughput** scales roughly linearly at server
Owner work

- Two offline tasks:
  - **Key distrib.**: for a widely-used software with > 5000 developers, membership changes take <1.6 sec/month
  - **Auditing** cost is modest and parallelizable

- Detection probability increases exponentially in no. of epochs of violation

- 4 min for $10^8$ attestations
Related work

- **Secure file/storage systems** (e.g., SiRiUS, SUNDR, Plutus):
  - No proofs of violation
  - No W and F detection due to different model
  - Access control not as scalable

- **Proofs of retrievability/possession** (e.g., POR, HAIL)

- **Byzantine fault tolerance** (e.g., BFT)
Conclusions

- **CloudProof** is a secure storage system for the cloud:
  - Detection of WF via auditing
  - Proofs of violation for IWF via attestations
  - Scalable access control using broadcast encryption

Thanks!