Enabling Non-repudiable Data Possession Verification in Cloud Storage Systems

Authors: Zhen Mo, Yian Zhou, Shigang Chen, Chengzhong Xu
Outline

• Introduction
• Security Concerns
• Previous Work -- Basic Approach
• Problem Definition
• Our Scheme
• Experimental Results
Introduction

Google

Dropbox

Microsoft

Box
System Model

Clients

Servers
Security Concerns

- **Data integrity**
  - Clients lose the local control of their data by outsourcing their data to the cloud
  - Cloud storage system is maintained by a third party who cannot be totally trusted (semi-honest)

- **Dynamic updating**
  - Clients should be able to perform dynamic update operations, e.g., modification, deletion and insertion, on their data
Previous Work -- Basic Approach

- There are many previous work, like skiplist based DPDP from Erway or Merkle Hash Tree based scheme from Wang.
- Divide file $F$ into $n$ data blocks: $m_1, m_2, ..., m_n$
- Create tags of the $n$ data blocks: $t_1, t_2, ..., t_n$.
- Create signatures of the $n$ data blocks: $\sigma_1, \sigma_2, ..., \sigma_n$.
- Construct a data structure to protect the tags.
- Outsource the data blocks, the signatures and the data structure to the cloud server. Keep some metadata.
- Probabilistic verification: Query a subset of the data blocks. The server returns an aggregated block and an aggregated signature and the partial data structure.
- Verify the aggregated block and the signature.
- How to ensure the signatures are updated and the correct?
- Verify the partial data structure.
Basic Approach

- Verify:
  - The value of the root.
  - The index of each leaf.

\[ h(n_i) = H(m_i) \]
\[ h_b = H(h(n_1) || h(n_2) || r_b) \]
Our Concerns

- More Efficient Dynamic updating
  - Use the coordinate instead of the index
  - Design a new Coordinate Merkle Hash Tree (CMHT) to assist the verification procedure

- Non-repudiation
  - When a user claims a data loss, how can we be sure that the user is correct and honest about the loss?
  - Design a scheme that allows the cloud and the users to supervise each other.
Problem Definition

- Design a data-possession verification scheme that:
  
  (1) Integrity Verification: probabilistically detect whether some of the blocks have been lost or corrupted at the cloud server.
  
  (2) Non-repudiation: make sure that the cloud can fend off false claims of client data loss.
Integrity Verification

- **Block Tag**: For each block $m_i$, we define its tag as
  \[ t_i = H(m_i) \| c_i, \]
  where $H$ is a collision-resistant hash function and $c_i$ is a coordinate value assigned to $m_i$.

- **Homomorphic Signature**: The client chooses $N = pq$, where $p$ and $q$ are two large primes and $g$ is an element of high order in $\mathbb{Z}_N$.
  \[ \sigma_i = t_i \cdot g^{m_i} \mod N, \]
  where $t_i$ is the tag of $m_i$. 
CMHT

- Coordinate Merkle Hash Tree (CMHT):

\[ l(w_x) = H(l(w_{\text{left}}) \ || \ l(w_{\text{right}})) \]
\[ l(w_x) = t_j \]
Deletion

Replacement node

Delete leaf node $w_4$
Insertion

Insert a new node $w^*$
Non-repudiation

- Meta data: Let T be the current time stamp. We define the meta data as follows:
  \[ M = \{ l(w_r), T, n, \sigma_M \} \]
- \( \sigma_M \) is a signature signed by both the client and the server using their private keys, \( sk_c \) and \( sk_s \)
  \[ \sigma_M = \text{Sign}_{sk_s}(\text{Sign}_{sk_c}(l(w_r) \| T \| n)) \]
- Unforgeability: neither the client nor the server can forge the meta data.
- Distinguishability: it is easy to resolve the dispute by authenticating the signatures with their public keys and comparing the time stamps.
Preprocess

- The client creates a pair of keys: pk\textsubscript{c}, sk\textsubscript{c}. The server also creates a pair of keys: pk\textsubscript{s}, sk\textsubscript{s}.
- The client divides file F into n blocks: m\textsubscript{1}, m\textsubscript{2}, ..., m\textsubscript{n}.
- The client generates a set of homomorphic signatures on the blocks: Φ = \{σ\textsubscript{i}\}, where i from 1 to n.
- The client constructs a CMHT.
- The client generates a signature: Sign\textsubscript{skc}(l(w\textsubscript{r}) \parallel T \parallel n), where T is a time stamp.
- The server verifies the signature and signs the signature: σ\textsubscript{s} = Sign\textsubscript{sk}\textsubscript{s}(Sign\textsubscript{skc}(l(w\textsubscript{r}) \parallel T \parallel n)). Store Ms.
- The server sends the signature back. The client takes the signature as σ\textsubscript{c}. Store Mc.
- The client outsource F, Φ and CMHT to the cloud server.
Verification

\[ R_k = \{ (c_i, v_i) \} \]

Query

\[ P: \mu = \sum_{i \in I} v_i m_i \in \mathbb{Z}_p, \sigma = \prod_{i \in I} \sigma_i^{v_i}, \Gamma \]

Generate a proof

\[ \text{Verity } \sigma = (\prod_i t_i^{v_i}) g^\mu \]

Verify the proof

\[ \text{Verity } \Gamma \]
The judge receives evidences \( \mathcal{E}_c \) and \( \mathcal{E}_s \) from the client and the server. The judge first verifies the correctness of the proof \( P \). Then it checks the signatures in \( M_c \) and \( M_s \). If both signatures are correct, it compares the time stamps to determine whose evidence is valid.
Client Cache

- To further improve the client’s performance, we may cache the upper levels of the tree on the client side.
- The upper levels account for a very small fraction of the whole tree, but it can significantly reduce the amount of computation and communication between the client and the server.
- For a file of $10^6$ data blocks, if we cache the top 10 levels of CMHT at the client side (which accounts for less than 0.1% of the whole CMHT tree), we can reduce the communication overhead by half.
Experimental Results

- We implement cloud storage servers on Amazon EC2.
- We use an ordinary desktop computer as a client.
- The size of the file used in our experiments is 1GB.
- We evaluate the performance of our scheme and compare it with DPDP under different block sizes.
- We let the client cache the upper half of the levels in the CMHT or in the skiplist of DPDP. The cached data is about 0.1% of the tree (or skiplist).
- Detect a 1% file corruption with 99% confidence needs querying a constant number of 460 blocks.
- Our average overhead is 31% - 50% of DPDP’s in the left plot, and our maximum overhead is 13% - 46% of DPDP’s in the right plot.
Communication overhead

Query a block

Insert a block

Modify a block

Delete a block

Block size (KB)

Block size (KB)

Block size (KB)

Block size (KB)
Our scheme reduces the average computational overhead by up to 58.5% and the maximum computational overhead by up to 78.7%, when comparing with DPDP.
Thank You!

Question?

Authors: Zhen Mo, Yian Zhou, Shigang Chen, Chengzhong Xu