Fair and dynamic proofs of retrievability

Nguyen, Ngoc Tram Anh

2014

Nguyen, N. T. A. (2014). Fair and dynamic proofs of retrievability. Student research paper, Nanyang Technological University.

https://hdl.handle.net/10356/105806

Fair and Dynamic Proofs of Retrievability

Nguyen Ngoc Tram Anh
School of Computer Enginnering

Assoc Prof Ng Wee Keong
School of Computer Engineering

Abstract – One of the big trends of computing nowadays is cloud computing. More users rely on cloud computing for data storage because of its usefulness and mobility. However, the cloud server may be untrustworthy and may modify or manipulate the outsourced data without our knowing. The scheme we are presenting in this paper has the ability to spot any abnormality in the data stored and also retrieve back the lost data in some cases. It also supports dynamic data (data that can be modified, inserted or deleted). That means the authentication data can be updated as soon as the outsourced data are changed. The scheme also protects an honest cloud server over a dishonest client.

Keywords – cloud security, cloud computing, data security, data storage, dynamic data, proof of retrievability

1 INTRODUCTION

Cloud computing has been adopted and used more and more over the years. Not only is it useful for business corporations to employ Software-as-a-Service or Database-as-a-Service over the cloud, normal users make use of cloud storage to sync and store their data and files over multiple devices. While cloud computing has its obvious advantages, putting data onto the cloud poses a serious risk of data manipulation by the cloud service provider. Data owners, in most cases, do not back up their data before storing in the cloud. Hence, any data modification done by the cloud service provider may not get notified by the data owner. This paper will introduce a method to periodically check for data integrity over the cloud, without the data authenticator (i.e. data owner or third-party authenticator) having to download the whole file.

Proof of retrievability (POR) is a cryptography paradigm to convince the data owner that his outsourced data can be retrieved in the event that some data is lost or malignantly modified. Some existing proofs of retrievability only deal with static data, that is, data that cannot be modified. With current demand for cloud storage, data is constantly modified, inserted or deleted (dynamic data). This paper will present a POR for dynamic data.

Fairness is another problem posed in a cloud storage setting. In here, the client may pose as a dishonest side and accuse the honest cloud service provider of malignantly changing the data. This paper will present a method to protect the cloud service provider of such scenario.

2 BUILDING BLOCKS

2.1 RANGE-BASED 2-3 TREE

2-3 tree is a self-balanced tree data structure where each parent has either 2 or 3 child nodes [1]. All the key values are kept at the leaf nodes. The original values to be kept inside the non-leaf node of a 2-3 tree are the value of the maximum key in the left subtree and that in the middle subtree. Search, insertion and deletion are constant in logarithmic time.

Range-based 2-3 tree is based on the property of a 2-3 tree, with additional information stored in non-leaf nodes to aid easy searching and modifying for authentication of dynamic data.

2.2 ERROR-CORRECTING CODE

Error-correcting code is a fundamental tool for retrievability ability. Raw data file is first applied with an error-correcting code method before any other cryptography and authenticating methods are used.

One common way of error-correcting is duplicating data such that when a certain number of bits (below a predefined threshold) is lost, we can always recover the data back.

2.3 HASH-COMPRESS-SIGN

Hash and compress is to minimize the authentication info that needs to be stored alongside with the actual data in the cloud server. Not only is small authentication data preferred by the cloud storage provider (less storage space taken up by the extra data), it is also preferred by the third-party authenticator or client because of less communication overhead of downloading the authentication data for checking.

Signing the data at the last step is to maintain the "fair" component in our POR. Only the client, knowing the secret key, can sign the data. The server will only keep the signed data. If a dishonest client wants to accuse the server of changing the data, the server can prove that his signed data and raw data matched with the secret key from the client.

3 CONSTRUCTING RANGE-BASED 2-3 TREE

Range-based 2-3 tree is an authenticated data structure supporting fast update of authenticator data when inserting, modifying or deleting a file block in a file [2]. Insertion, deletion or modification of a file block, (which is a leaf of the 2-3 tree) will only take logarithmic time of total number of nodes. And so is updating the authentication value, which is the value at the root of the tree.

3.1 CONTENT OF 2-3 TREE NODES

Instead of storing the max key values of the subtree of each node, we store the max leaf index (called *max(v)*) under the subtree at the node. Furthermore, each node of the range-based 2-3 tree consists of ($l(v)$, $r(v)$, $x(v)$) where

- $l(v)$ is the height of node v counting from the leaves, with the leaves having $l(v) = 1$
- $r(v)$ is the range value of node v. It is the total number of leaves under the subtree at v. $r(v) = 1$ if v is a leaf
- $x(v)$ is the authentication value at node v, where *x(v) = collision-resistant hash function of (l(v)||r(v)||x(child1)||x(child2)||x(child3)). Child1*, *child2*, *child3* are the 3 child nodes of v. If v has only 2 child nodes, *x(child3)* is null

All the key values are put at the leaves and are arranged in order from left to right. The content $(l(v), r(v), x(v))$ of a leaf node at ith order is $(1, 1, e_i)$ where e_i is the value of the ith element.

3.2 THE USE OF RANGE-BASED 2-3 TREE

Given a data file F divided into blocks of equal length F_i, $F = (F_1, F_2,\ldots,F_n)$ where n is the total number of blocks. We put these blocks into the leaves of a range-based 2-3 tree; order of the blocks is preserved. One file makes up of one tree. Each leaf node contains $(1, 1, e_i)$ where e_i is the content of F_i.

Follow the content of node in part 3.1; construct the remaining of the tree from the given leaves, until we can find a root. *x(root)* is the authentication value of the whole tree. The client constructs the tree when he first uploads the file into the cloud server, and will only keep the *x(root)* value. The entire tree is uploaded onto the cloud server for future updates of file blocks and their affected ancestor nodes.

One advantage of the range-based 2-3 tree over other authenticated data structures is the ease in inserting and deleting of key values at the leaves. Inserting a file block in the middle of the file means inserting a leaf at ith order out of m leaves ($0 \leq i < m$). Range-based 2-3 tree, with its range and max index values at each node, makes it (i) easy to identify the leaf, given its index position and (ii) flexible in inserting and deleting a leaf as number of nodes affected is *log(n).* Take insertion of file block F'_i after block F_i as example. Based on the range and max index values, we trace the tree starting from the root until we reach the desired leaf. This path of nodes is called Proof Path. F'_i will be the child of the node that has only 2 child nodes. Inserting F'_i will then only involve updating the max index *max(v),* range value *r(v)* and authentication value *x(v)* of the nodes in proof path. If the parent nodes of F_i and F_{i+1} already contain 3 child nodes, we need to add a new level of nodes to accommodate the insertion and balance the tree from here. Number of nodes affected is still around *log(n).* A similar arrangement can be deduced for the case of deletion and modification. Hence, range-based 2-3 tree is a suitable authenticated data structure for dynamic data.

4 MAIN CONSTRUCTION

Below is the step-by-step of this Fair and Dynamic Proof of Retrievability scheme:

4.1 INITIALIZATION

4.1.1 Key Generation

1. Generate 2 private key $k_1, k_2 \in_R \{0, 1\}^l$ with l as a security parameter
2. Generate a pair of private and public key *(PK, SK)* from RSA cryptography
3. The client keeps (k_1, k_2, *SK*) and the server will keep *PK*

4.1.2 Authentication Parameter Calculation

1. Given file F, divide F into blocks of equal length $F = (F_1, F_2, \ldots, F_n)$. Note that size of F_i should not be too large to prevent number format overflow during calculation.
2. Create two random blocks F_0 and F_{n+1}. Now $F = (F_0, F_1, F_2,\ldots, F_n, F_{n+1})$
3. Apply Error-Correcting Code to each file block in F. Call the output $F' = (F'_0, F'_1,\ldots, F'_{n+1})$
4. Apply collision-resistant hash function to F' where $H_i = h(F'i)$ (for $0 \leq i \leq n+1$). Denote $H = (H_0, H_1,\ldots, H_{n+1})$
5. Construct a range-based 2-3 tree T from H where H_i is the leaf
6. For $1 \leq i \leq n-1$, $\lambda_i = h(H_i // H_{i+1})$.
7. Calculate λ for $\lambda = \bigodot_{i=0}^{i=n-1} \lambda_i$ (where Θ is a group operation)
8. Sign λ with the client's SK to get δ, $\delta = \text{Sign}(SK, \lambda)$
9. Compute integrity tag σ for each of the block in F':
$$\sigma_i = f_{k1}(H_i) + \sum_{j=1}^{z} f_{k2}(j)F'ij$$
10. The client will keep ($x(root)$, λ, fid) where fid is a unique id for the file F
11. Send (F', fid, $(\sigma_0,\ldots, \sigma_{n+1})$, T, λ, δ) over to the server through a secure channel

4.2 DATA MODIFICATION

We will examine the case for insertion of block F'_i into F. Note that insertion at block i^{th} means the new block will be inserted after block i^{th}.

1. Client **C** sends (I, i, α) where I stands for insertion, i is the position of insertion block and α is the Error-Correcting Code encoded of the new block
2. Server **S** runs through the 2-3 tree of the file and returns proof path for leaf at index i and $i+1$ (including H_i and H_{i+1})
3. Client **C** update λ, δ and integrity tag as shown:

$$\lambda' = \lambda \odot h^{-1}\,(H_i||H_{i+1}) \odot h^{-1}\,(H_i||h(\alpha)) \odot h^{-1}\,(h(\alpha)||H_{i+1})$$

$$\delta' = Sign(SK, \lambda')$$

$$\sigma_\alpha = f_{k1}\left(h(\alpha)\right) + \sum_{j=1}^{z} f_{k2}(j)\alpha_j$$

4. Client **C** then sends the info (δ', σ_α, (I, i, α), fid) to the server
5. The server **S** updates δ' and add the new block after i^{th} leaf in the tree
6. The server **S** compute proof path for leaf at index i and i' and sends the proof path over to the client **C**
7. The client **C** verifies proof path and updates $x(root)$

4.3 VERIFICATION

The client or an entrusted third-party authenticator can periodically check for data integrity on the cloud without having to download the whole file back for verification. These following steps will show how we can enable blockless authentication (blockless means we do not have to download the file block in order to check for its integrity). Note that blockless authentication is preferred not only for its small communication overhead but also for protection of data if the client wants to delegate the task of authenticating over to a third-party authenticator.

1. The client **C** chooses c random elements I = $\{\alpha_1, \ldots, \alpha_c\}$ where c and α_i must be smaller than the total block number of a file. ($1 \le i \le c$)
2. Client **C** continues choosing c random elements β_i for $1 \le i \le c$
3. Client **C** sends the challenge C = $\{(\alpha_i, \beta_i)\}$ to the server
4. The server **S** computes the following parameters:

$$U_j = \sum_{i \in I} \beta_i F'_{ij} \qquad 1 \le j \le z$$

$$\sigma' = \sum_{i \in I} \beta_i \sigma_i$$

$$H_i = h(F'_i) \qquad i \in I$$

$$\pi_i = \text{proof path for leaf at index } i \quad i \in I$$

5. Server **S** then sends response $R = \{u_j, \sigma', H_i, \pi_i\}$ to client **C**
6. Client **C** output true if these two conditions hold:
   - Verify if $x(root)$ and π_i match
   - $\sigma' = \sum_{i \in I} \beta_i f_{k1}(H_i) + \sum_{j=1}^{z} f_{k2}(j) u_j$ (1)

4.4 EVALUATION

The above method is an effective way to authenticate dynamic data for cloud storage. However, too many multiplications between file blocks and random numbers are used (e.g. section 4.1.2 step number 9) that the length of file blocks and random numbers (which underlies security) is directly proportional to computation overhead. If we choose a long file blocks for better security, it will incur lots of computation overhead (long computation time and require strong computation machine to prevent number format overflow). If we choose a short length for file blocks, it will reduce security level of our authentication data. The integrity tag σ will get smaller, hence more vulnerable to bruteforce attack.

Another problem might pose when implementing this scheme is the construction of range-based 2-3 tree. It is very hard to program the tree in an efficient way to minimize space taken.

5 IMPLEMENTATION

From the construction steps in section 4, a simplified version is written in Java and tested for its practicality.

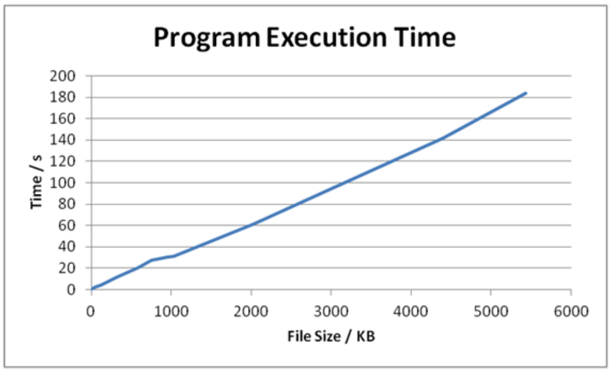
5.1 OVERVIEW

1. Key generation is generated using KeyPairGenerator class from SUN [3]
2. A file is divided into blocks of 2 bytes
3. Hash function is implemented using MessageDigest class [4]
4. For simplicity, addition is used for group operation
5. Signing is used with Signature class in Java [5]
6. The program successfully verifies data at (1)

5.2 RESULT

We run the program against multiple sizes of different file types (jpg, wma, txt). The table below is the average time taken to compute the authentication parameters and verification process. The experiment is run on an Intel Core i5 CPU 2.53 GHz and 4GB RAM, with Java SE 7.

Chart 1 Execution time against File Size



Note that the time here includes the initialization process where a lot of time is spent dividing file into blocks and hashed. The verification process itself takes less than 1 millisecond even for the biggest file that was tested. This means that the periodical spot checking of file integrity will only take minimal time to execute.

6 CONCLUSION

Above is a fair and dynamic way for authenticating of data in a cloud storage setting. It is considered a scheme in Proof of Retrievability by using Error-Correcting Code. Range-based 2-3 tree enables easy management for data insertion, modification and deletion.

Future work may want to look into real-time verification whenever user makes a database query. To make sure the returned results are accurate and untampered, verification should be done on the tables involved.

ACKNOWLEDGMENT

I would like to thank Prof Ng Wee Keong for his guidance and for giving me the opportunity to do research.

I wish to acknowledge the funding support for this project from Nanyang Technological University under the Undergraduate Research Experience on CAmpus (URECA) programme.

REFERENCES

[1] University of Wisconsin-Madison, [Online]. Available: http://pages.cs.wisc.edu/~vernon/cs367/notes/10.23TREE.html.

[2] Q. Zheng, "Fair and Dynamic Proofs of Retrievability," ACM, pp. 237-248, 2011.

[3] Oracle, "KeyPairGenerator (Java Platform SE 7)," [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/security/KeyPairGenerator.html.

[4] Oracle, "MessageDigest (Java Platform SE 7)," [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html.

[5] Oracle, "Signature (Java Platform SE 7)," [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/security/Signature.html.