SCIENCE CHINA
Information Sciences

• LETTER •

November 2018, Vol. 61 119101:1–119101:3
https://doi.org/10.1007/s11432-017-9286-y

Efficient method to verify the integrity of data with supporting dynamic data in cloud computing

Cheng GUO[1,2], Xinyu TANG[1,2], Yingmo JIE[1,2*] & Bin FENG[1,2]

[1]*School of Software Technology, Dalian University of Technology, Dalian* 116620*, China;*
[2]*Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian* 116620*, China*

Received 14 September 2017/Accepted 27 October 2017/Published online 15 August 2018

Citation Guo C, Tang X Y, Jie Y M, et al. Efficient method to verify the integrity of data with supporting dynamic data in cloud computing. Sci China Inf Sci, 2018, 61(11): 119101, https://doi.org/10.1007/s11432-017-9286-y

Dear editor,

Cloud computing allows users to move local data to a server in the cloud to lighten the loads on their local storage. Because they no longer have physical possession of the data, it is very important to ensure the correctness and consistency of the outsourced data [1,2]. We herein use the Merkle Hash tree (MHT) [3] data structure and the Bloom filter [4] to achieve more effective data verification than previous methods. At the same time, our method can also support dynamic operations and public verification very well.

Preliminaries. The MHT has been proposed as a well-organized structure to reduce the cost of hashing large data structures. It is often used to take a summative snapshot of the storage region to detect data tampering. MHT is constructed as a binary tree in which the value of a node is obtained by concatenating and hashing the values of its child nodes. A Bloom filter is a space-efficient data structure that represents a data set and supports membership queries. Bloom filter is much more efficient than other data structures, which require a greater amount of time to add an element or to check whether an element belongs to a specific set. The time required for the Bloom filter is constant irrespective of the cardinality of the set, and it has a time complexity of $\mathcal{O}(1)$. Bloom proposed the technique for applications in which the amount of source data would require an impractical large hash area of high efficiency, so we can use this method to verify if data belongs to the original dataset or if it has been damaged by an adversary.

Processing procedure. To effectively support public verification without having to retrieve the data blocks themselves, we resort to the MHT and Bloom filter algorithm. Next, we present the main idea of our scheme in detail. We assume that dataset D is divided into n blocks, i.e., $d_1, d_2, \ldots, d_n$.

First, the user generates the MHT using the hash value of data block d_i as the tree leaf node. Different from a traditional MHT, to improve efficiency, we do not generate up to a tree root node; rather, we stop at a proper middle layer, i.e., generate a forest, for every MHT containing part of the data. These nodes are called terminate nodes, and their number is decided by considering their influence on efficiency and space consumption. Figure 1(b) shows one tree in the forest. $h(d_1), \ldots, h(d_8)$ are the authentic data values hashes. $h(c) = h(h(d_1) \parallel h(d_2)), h(a) = h(h_c \parallel h_d)$, and $h(R) = h(h_a \parallel h_b)$. After generating these terminate nodes $h(R_1), h(R_2), \ldots, h(R_i)$, we create a Bloom filter to store them. We create an n-bit array $X[n]$, and set the initial values as $X[j] = 0, 0 \leqslant j < n$. Then we define k different hash functions $H_1(x), H_2(x), \ldots, H_k(x)$. The terminate nodes are made as the k hash func-

* Corresponding author (email: jymsf2015@mail.dlut.edu.cn)

tions' input, respectively. In order to support dynamic data operations, we make use of a transferred Bloom filter — the Counting Bloom filter [4], which expands each bit of array $X[n]$ to a counter. When the hash function value is 1, the corresponding counter of $X[i]$ increases by 1. Therefore, the value domain of every bit expands instead of simply being 0 or 1, as shown in the following formula:

$$\begin{cases} H_1(h_{R_1}) = a \\ H_2(h_{R_1}) = b \\ \quad\vdots \\ H_k(h_{R_1}) = g \end{cases} \longrightarrow \begin{cases} X[a] = 1 \\ X[b] = 1 \\ \quad\vdots \\ X[g] = 1 \end{cases}, \tag{1}$$

$$\begin{cases} H_1(h_{R_2}) = e \\ H_2(h_{R_2}) = f \\ \quad\vdots \\ H_k(h_{R_2}) = g \end{cases} \longrightarrow \begin{cases} X[e] = 1 \\ X[f] = 1 \\ \quad\vdots \\ X[g] = 2 \end{cases}, \tag{2}$$

$$\cdots .$$

Finally, the user uploads data blocks $D = \{d_1, d_2, \ldots, d_n\}$, the MHT, and the Bloom filter $X[n]$ to the cloud service server (CSS).

Verification. If users or a trusted third party (TTP) want to access the data blocks stored remotely in CSS, they must determine whether those data blocks are damaged. First, the verifier challenges the CSS or prover to send chal $= \{i\}$. Upon receiving the challenge, the server returns $h(d_i)$, and the Bloom filter array $X[n]$ to the verifier. In addition, the prover also provides the verifier with a small amount of auxiliary information $\{Au_i\}$, which is the sibling nodes on the path from the leaves $h(d_i)_{0\leqslant i<n}$ to the roots $h(R_i)$ of the MHT. Then the prover responds to the verifier with Resp $= \{h(d_i), X[n], Au_i\}$. After receiving the response from the prover, the verifier generates the corresponding MHT root $h(R_{\text{new}})$ using $\{h(d_i), Au_i\}$. Then the verifier takes $h(R_{\text{new}})$ as the k hash functions' input to compute $H_i(h(R_{\text{new}})), 0 < i \leqslant k$, and then checks to determine whether $X[H_i(h(R_{\text{new}}))]$ is equal to 1 or larger. If the value of array $X[H_i(h(R_{\text{new}}))]$ is positive, the verification is completed, and the data block d_i is stored correctly in the remote server; otherwise, the value of 0 means that d_i has been corrupted, as shown in the following formula:

$$\begin{cases} H_1(h(R_{\text{new}})) = a_{\text{new}} \\ H_2(h(R_{\text{new}})) = b_{\text{new}} \\ \quad\vdots \\ H_k(h(R_{\text{new}})) = j_{\text{new}} \end{cases} \longrightarrow \begin{cases} X[a_{\text{new}}]? \geqslant 1 \\ X[b_{\text{new}}]? \geqslant 1 \\ \quad\vdots \\ X[j_{\text{new}}]? \geqslant 1 \end{cases}. \tag{3}$$

Dynamic data operation. In addition to the verification of the archived files, we also consider the integrity verification about the timely updated data. The following information shows how our scheme correctly and efficiently deals with the problem of dynamic data operations, including data insertion (I), data deletion (D) and data modification (M) for cloud storage.

General data insertion refers to inserting new data blocks into some specified positions in the entire storage system. Assume the user wants to insert block $d\#$ into the i position. First, the user generates a corresponding hash leaf $h(d\#)$ of MHT and then constructs an insertion request message insertion $= (I, i, d\#, h(d\#))$ which is sent to the CSS, where I denotes the insertion operation. When the request is received, the server takes the following actions: (1) storing the data block $d\#$; (2) adding a leaf $h(d\#)$ after leaf $h(d_{i-1})$ in the MHT and generating a new root $h(R')$ based on the updated MHT; (3) taking $h(R')$ as the Bloom filter's input data and computing $H_i(h(R')), 0 < i \leqslant k$, and then setting array $X'[H_i(h(R'))_{0<i\leqslant k}] = 1$. (We used a stack to store the bits with values marked as 1. For example, in the stack value, 100 means the Bloom filter array $X[100] = 1$.) Then, the server responds to the user with a proof for the insertion operation, $P_{\text{update}} = \{AU\#, X'[H_i(h(R'))_{0<i\leqslant k}] = 1\}$. An example of data block insertion is illustrated in Figure 1(a). With the aim of inserting $h(d\#)$ after leaf node $h(d_2)$, only node $h(d\#)$ and an internal node C need to be added to the original MHT, and the server computes the node upward along the path from leaf to root. When the user receives the insertion proof P_{update} from the server, he generates root $h(R_{\text{new}})$ using $\{h(d\#), Au\#\}$ and input $h(R_{\text{new}})$ to get the Bloom filter array bit. Then he determines whether these bits are matched with the values stored in the stack $X'[H_i(h(R'))_{0<i\leqslant k}]$. If there are any mismatches, we output a false result, which means that the insertion operation failed; otherwise, we output the true result and the server updates the MHT-based, relevant computed values and sets the Bloom filter array as $X[H_i(h(R))_{0<i\leqslant k}]$ decreased by 1 and $X[H_i(h(R'))_{0<i\leqslant k}]$ increased by 1. Then, the user deletes block $d\#, h(d\#), h(R_{\text{new}}), P_{\text{update}}$ from the local storage.

Data deletion is similar to the insertion operation. After the user sends a deletion request, the server computes MHT by deleting leaf node $h(d\#)$, updating the node upwards along the path from the leaf nodes to the root node, and generating a new root R', as shown in Figure 1(b). The user also receives $X'[H_i(h(R'))_{0<i\leqslant k}]$. Then, the server

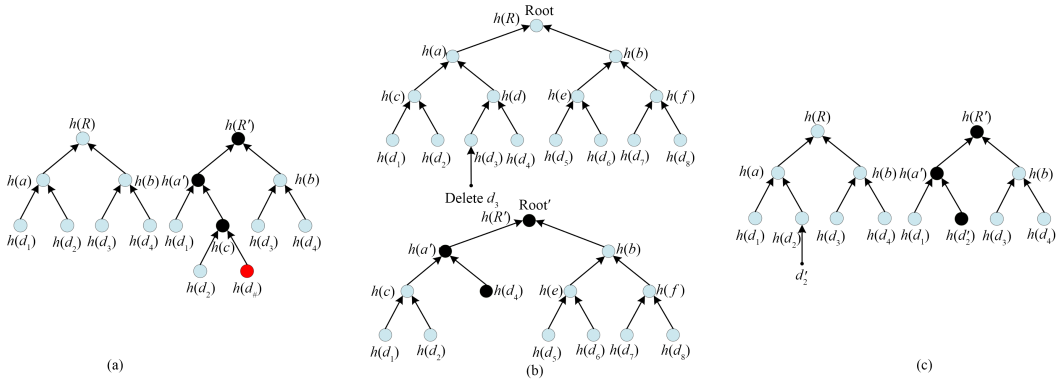

Figure 1 (Color online) Dynamic MHT operations. (a) Insertion operation; (b) deletion operation; (c) modification operation.

returns a deletion proof P_{delete} to the user for forward verification. The details of the protocol are similar to those of the insertion operation. If the verification succeeds, the server deletes block $d\#$, updates MHT, decreases $X[H_i(h(R))_{0<i\leqslant k}]$ by 1, and increases $X[H_i(h(R'))_{0<i\leqslant k}]$ by 1.

Modification means the user wants to replace the specific blocks with new blocks. Modification does not change the logical structure, different from the cases for insertion and deletion. Additionally, modification seems to combine the operations of deletion and insertion. Assume that the user wants to modify the i block d_i with new block $d\#$, as illustrated in Figure 1(c). First, the user generates $h(d\#)$ and constructs an update request message update $= (M, i, d\#, h(d\#))$. Upon receiving this request, the server uses $h(d\#)$ with the MHT to recompute the MHT and generate a new root $h(R')$. Then the server uses $h(R')$ to get $H_i(h(R'))_{0<i\leqslant k}$ and finally returns a modification proof $P_{\text{update}} = \{Au_i, X[H_i(h(R'))_{0<i\leqslant k}]\}$ to the user. The next steps can refer to the data insertion phase.

Performance analysis. We compared the features of the proposed scheme with many other existing schemes. Our scheme simultaneously supports dynamic data and public verification. The computing complexity is between $\mathcal{O}(1) \sim \mathcal{O}(\log n)$, which is less than $\mathcal{O}(\log n)$ in [5]. We then performed an experiment to test the constructing time in different situations. Compared with other schemes, our approach was more efficient in this area. We also conducted experiments for specific data blocks to verify and demonstrate the related efficiency, and the results show that our scheme is much more efficient than the traditional scheme, in particular. Different parameters' impacts were analyzed in these experiments. For detailed experimental results and analysis, please refer to Appendix A.

Conclusion. We proposed an efficient method to verify the integrity and consistency of data by using the MHT and Bloom filter to reduce the computing and transmission overhead. This scheme can support dynamic data and public verification, which makes the scheme much more practical than other existing methods. All users can verify data by themselves or delegate the verification task to a TTP in cases where the users lack the required computing resources.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61501080, 61572095, 61771090).

Supporting information Appendix A. The supporting information is available online at info.scichina.com and link.springer.com. The supporting materials are published as submitted, without typesetting or editing. The responsibility for scientific accuracy and content remains entirely with the authors.

References

1 Zhang Y, Xu C, Liang X, et al. Efficient public verification of data integrity for cloud storage systems from indistinguishability obfuscation. IEEE Trans Inform Forensic Secur, 2017, 12: 676–688
2 Sookhak M, Gani A, Khan M K, et al. WITHDRAWN: dynamic remote data auditing for securing big data storage in cloud computing. Inf Sci, 2017, 380: 101–116
3 Garg N, Bawa S. RITS-MHT: relative indexed and time stamped Merkle hash tree based data auditing protocol for cloud computing. J Netw Comput Appl, 2017, 84: 1–13
4 Bonomi F, Mitzenmacher M, Panigrahy R, et al. An improved construction for counting bloom filters. In: Proceedings of the 14th Annual European Symposium on Algorithms, Zurich, 2006. 684–695
5 Wang Q, Wang C, Ren K, et al. Enabling public auditability and data dynamics for storage security in cloud computing. IEEE Trans Parall Distrib Syst, 2011, 22: 847–859