Data Auditing Method Based on Relative-Index Hash Tree in Cloud Environments (Postprint)

Authors: Li Mengting, Zhou Anning

Date: 2018-05-02T00:00:00+00:00

Abstract

To ensure that outsourced data in cloud environments remains free from tampering and to enhance the efficiency of data integrity auditing, this paper proposes a data auditing method based on Relative Index Merkle Hash Tree (RI-MHT). Firstly, each node of the classical MHT is modified to store two pieces of information: the hash value of the data block and the node' s relative index, thereby integrating the MHT with the node's relative index to reduce the computational cost of data block searching. Subsequently, by incorporating the last modification time of the data, the freshness of the data is guaranteed. Experimental results validate the effectiveness of the proposed method. Compared with other comparable methods, the proposed method demonstrates certain advantages in computational cost, communication cost, and storage cost, and can detect improper server operations with high probability.

Full Text

Preamble

Data Auditing Method Based on Relative Index Hash Tree in Cloud Environment

Li Mengting, Zhou Anning
(Guangdong University of Foreign Studies, Guangzhou 510006, China)

Abstract: To ensure that outsourced data in cloud environments remains tamper-free and to improve the efficiency of data integrity auditing, this paper proposes a data auditing method based on Relative Index-Merkle Hash Tree (RI-MHT). The approach first modifies each node of the classic MHT to store two pieces of information: the hash value of the data block and the node's relative index, integrating the MHT with relative indices to reduce the computational cost of data block searches. It then ensures data freshness by incorporating the last modification time of the data. Experimental results verify the effectiveness of

the proposed method. Compared with similar existing methods, this approach offers advantages in computational cost, communication cost, and storage cost, while detecting server misbehavior with high probability.

Key Words: cloud environment; data integrity audit; relative index; hash tree; computational cost

0 Introduction

Cloud computing has rapidly emerged in the market, fundamentally transforming how service providers interact with customers, distributors, and employees. Internet services such as Baidu and Alibaba have revolutionized communication, shopping, education, and numerous other activities. Cloud services include Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). As an infrastructure, the cloud provides remote storage locations for data, which are maintained by the cloud provider themselves. While cloud storage offers convenience, data security remains a critical vulnerability.

Several researchers have analyzed and developed solutions for data storage security. For instance, reference [**?**] proposed two data possession verification schemes: a sampling scheme and an efficient scheme. However, these offered weak data ownership guarantees and utilized RSA-based homomorphic tags, with the protocol supporting only static data and no dynamic operations. Reference [**?**] introduced two Proofs of Retrievability (PoR) schemes using homomorphic authenticators that support public auditing, based on publicly auditable BLS signatures. While efficient in terms of auditor time complexity, the communication complexity of this protocol increases linearly with the size of outsourced data blocks. Reference [**?**] proposed a mechanism for handling dynamic data that relies on rank-based authenticated skip lists; although this improved tag generation methods, its efficiency was not validated. Reference [**?**] utilized Merkle hash trees with auxiliary design modules to ensure devices proactively and periodically send their platform status to a management center, reducing storage overhead and computational costs and alleviating pressure on the management center, but this work was limited to static data. Reference [**?**] presented an improved protocol based on the correctness and completeness of server-held data evidence, which offered good security for encrypted outsourced databases but was also restricted to static data.

Existing protocols attempt to provide data integrity auditing for cloud systems but have not adequately addressed both public auditing and data dynamism. To this end, this paper proposes a data integrity auditing protocol featuring a Relative Index Merkle Hash Tree (RIMHT) that supports public auditing and dynamic data operations while ensuring data freshness. The protocol also considers privacy protection for outsourced data.

1 Proposed Data Integrity Auditing Model

The proposed data integrity auditing model is illustrated in Figure 1 [Figure 1: see original paper]. The numbers in the figure indicate the operation sequence during a data audit, while "*" marks activities that can be performed independently of the auditing process. The model involves three entities:

a) **Data Owner (DP):** Updates data through subsequent operations such as modification, insertion, and appending. The DP is an entity that relies on the cloud provider for data maintenance and is typically resource-constrained.

b) **Cloud Service Provider (CSP):** A storage server entity with sufficient computational resources and unlimited storage space. The CSP is primarily responsible for preserving and maintaining outsourced data but is generally considered an untrusted entity.

c) **Third-Party Auditor (TPA):** A professional entity that audits the DP' s data. The TPA is trusted by both the CSP and DP and can minimize the DP' s computational burden for data auditing.

While the TPA is a trusted entity and the CSP is untrusted, both may pose threats to the DP' s data:

a) **Data security threats from TPA:** The DP relies on the TPA to ensure data integrity, assuming it is an authentic and reliable independent entity. However, the TPA always has the potential to snoop on the DP' s data, potentially compromising data privacy in public auditing protocols. The protocol proposed in this paper does not address data privacy issues and assumes the TPA is loyal and reliable.

b) **Security threats from CSP:** The CSP may threaten the DP' s data in several ways: it might remove infrequently accessed data without notifying the DP to save server space, or it might introduce processing errors that cause irreparable damage to the DP' s data.

c) **External threats:** Legitimate users can access outsourced data through applications provided by the CSP, but weak applications may introduce risks. Login credentials of legitimate users might be used by strangers who could anonymously pollute or delete data. Additionally, former CSP administrators might invade cloud servers and compromise stored data. Therefore, ensuring data accessibility while preventing external attacks on outsourced data is crucial for CSPs when providing various cloud services.

2 Proposed Protocol

This section explains the RI-MHT protocol for performing data integrity auditing in cloud computing. It first describes modifications to the classic MHT [**?**] and then details the construction of the proposed protocol. The system work-

flow is shown in Figure 2 [Figure 2: see original paper], which follows the model in Figure 1 but expands the data flow.

2.1 Modification of MHT

To reduce the computational complexity of searching nodes during data integrity auditing in MHT, this paper modifies each node of the classic MHT to store two pieces of information: the hash value of a data block and the node' s relative index. The relative index associated with a node P specifies the number of leaf nodes belonging to P' s subtree.

In the modified MHT, the relative index of leaf nodes is set to 1. For example, if for a parent node P, L and R are the left and right children with hash values H_a and H_b and relative index field values r_a and r_b, respectively, then node P' s hash value is $H_P = H(H_a||H_b)$ and its relative index field value is $r_P = r_a + r_b$. A timestamp field is associated with the MHT' s root node. An example of the modified MHT is shown in Figure 3 [Figure 3: see original paper]. Here, $R_H = H(H_R||t_d)$, where t_d represents the date and time when the tree was created. Since any modification to data blocks in the tree updates t_d, R_H reflects the last modification time and date, thereby ensuring data freshness.

2.2 Definitions

This section provides definitions for the proposed algorithms:

a) **Keygen(λ):** Executed by the DP. Here, λ denotes the security parameter. The algorithm outputs a key pair (public key, private key).

b) **FileTagGen($fname, k, n, t_d$):** Executed by the DP to generate a tag for file F. The algorithm' s inputs include the outsourced file name, private key, number of data block partitions, and file preprocessing date and time. The file tag is represented as τ.

c) **BlockSigGen($k, H(d_i), t_d, u$):** Executed by the DP. Its inputs are the private key k, file block hash, file preprocessing date and time, and a random element u. The algorithm outputs $\psi = \{\sigma_i\}_{i=1}^{n}$, an ordered set of BLS signatures [**?**] for file blocks.

d) **Challenge (Challenge):** Executed by the TPA to generate a challenge message sent to the CSP after the DP delegates auditing authority.

e) **GenProof(F, ψ, C):** Executed by the CSP immediately upon receiving challenge message C from the TPA to generate a proof θ, which is passed to the TPA for verification. The algorithm' s inputs are file F, signature set ψ, and challenge message C.

f) **VerifyProof(C, θ_f, η):** Executed by the TPA to verify the proof θ_f received from the CSP. The algorithm' s inputs are challenge message C, message θ_f from the CSP, and public key η. The algorithm outputs $\{TRUE, FALSE\}$ depending on verification success or failure.

2.3 Construction Details

Let F denote the file to be outsourced, divided into n data blocks $\{d[1], d[2], ..., d[n]\}$. Let $e : G \times G \rightarrow G_T$ be a bilinear map where g is a generator and p is the prime order of group G. Let $H : \{0,1\}^* \rightarrow G$ be a cryptographic hash function. The proposed protocol is as follows:

a) Keygen(λ): First, the DP must generate a key set $(pubkey, seckey)$ using the key generation procedure shown in Algorithm 1. The algorithm selects a random element $k \in Z_p$ as a private key and generates g^k as the public key.

Algorithm 1: Key Generation Algorithm
Input: Security parameter λ
Output: Publish system parameters (g, η), retain private key k.

1. Initialize pairing: $pbc\_demo\_pairing\_init(pairing, count, param)$
2. Initialize generator $g \in G$: $element\_init\_G(g, pairing)$
3. Initialize public key $\eta \in G$: $element\_init\_G(\eta, pairing)$
4. Initialize private key $k \in Z_p$: $element\_init\_Zr(k, pairing)$
5. Generate system parameters and private key: $k \leftarrow element\_random(k)$
6. Generate public key: $\eta \leftarrow element\_pow\_Zn(\eta, g, k)$

b) FileTagGen: To generate file F' s tag, the DP first generates a random element $u \in G$. The DP then generates the system date and time, denoted as t_d. The system date and time are concatenated into the file tag to ensure data file freshness. Let τ be file F' s tag. The concatenated string hash h is stored locally for future file tag verification. The pseudocode for generating the file tag is shown in Algorithm 2.

Algorithm 2: File Tag Generation
Procedure GENERATE_FILE_TAG
Input: File name $fname$, private key k, number of file blocks n, date and time t_d
Output: Publish file tag (τ), retain file tag hash (h).

1. Initialize pairing: $pbc\_demo\_pairing\_int(pairing, count, param)$
2. Initialize random elements $u \in G$ and $h \in G$:
   $element\_init\_G(u, pairing)$
   $element\_init\_G(h, pairing)$
3. Read system date and time: $t_d \leftarrow ctime(t)$
4. Concatenate $fname, n, u, t_d$: $file\_tag\_concatenation \leftarrow fname||n||u||t_d$
5. Convert $file\_tag\_concatenation$ to random element: $h \leftarrow element\_from\_hash(file\_tag\_concatenati$
6. Generate file tag: $\tau \leftarrow element\_pow\_Zn(\tau, file\_tag\_concatenation, k)$

c) BlockSigGen: After generating the file tag τ, the DP generates BLS signatures σ_i for each i-th file block $d[i]$ as $\sigma_i = (H(d[i]) \cdot u^{b_i})^k$, where $H(d[i])$ is the file block hash value obtained using a cryptographic hash function. The proposed protocol uses the SHA256 hash algorithm. The signature set is $\psi = \{\sigma_i\}_{i=1}^n$. After generating block signatures, the DP constructs an RI-MHT using

$\{H(d[i])\}_{i=1}^{n}$ as leaf nodes and generates a root node R_H.

When implementing R_H, the DP concatenates it with system date and time to ensure data freshness: $R_H = H(H_R||t_d)$. The DP then signs the root using k as $sig_{R_H} = (R_H)^k$. The DP sends information $\{F, \psi, \tau, \theta, \rho\}$ to the CSP and stores the same information locally before deleting it. The pseudocode for the MHT creation algorithm is shown in Algorithm 3.

Algorithm 3: MHT Creation Algorithm
Procedure MERKLE_HASH_TREE_CREATION
Input: Structure element mt, block hash header file $merkle\_tree\_h$
Output: Root node ($roothash$)

1. Set $leaf\_start$ to number of blocks
2. Initialize number of nodes in tree: $mt \rightarrow n \leftarrow (leaf\_start + mt \rightarrow data\_blocks - 1)$
3. Allocate memory for mt in $mt \rightarrow nodes$
4. Initialize $leaf\_start$ counter
5. While counter $\leq mt \rightarrow n$:
   Set $mt \rightarrow node[count].index = 1$
   Set $mt \rightarrow node[count].hash = H(d[count - leaf\_start])$
   Set $mt \rightarrow node[count].left = NULL$
   Set $mt \rightarrow node[count].right = NULL$
6. End while
7. Initialize counter to $leaf\_start - 1$
8. While counter > 0:
   Set $mt \rightarrow node[count].hash = mt \rightarrow node[2{*}count].hash + mt \rightarrow node[2{*}count + 1].hash$
   Set $mt \rightarrow node[count].index = mt \rightarrow node[2 * count].index + mt \rightarrow node[2 * count + 1].index$
   Set $mt \rightarrow node[count].left = mt \rightarrow node[2 * count].left$
   Set $mt \rightarrow node[count].right = mt \rightarrow node[2 * count].right$
9. End while
10. Return address $mt \rightarrow node$

d) Challenge: The DP may check the integrity of outsourced data F. To do so, the DP delegates the auditing task to the TPA by sharing file details, root signature, and private key over a secure channel. To initiate auditing, the TPA selects a random subset Q containing k elements $Q = \{i_1, i_2, ..., i_k\} \subset [1, n]$ and sends challenge $C = \{(i, b_i)\}_{i \in Q}$ to the CSP.

e) GenProof: Upon receiving challenge message C from the TPA, the CSP immediately uses a search algorithm to locate the i-th node C_i in the hash tree. During node search, the CSP stores information about each sibling node on the search path for the i-th node. This information serves as auxiliary information (AI) for the node being searched. For example, if the i-th node is node 5 in Figure 3, its AI is $\{A_H, (H(d[6]), 1), (C_H, 2)\}$, where L denotes left sibling node and R denotes right sibling node. If the i-th node does not exist in the MHT,

the CSP notifies the TPA of the block-not-found message and stops the auditing process. If the block exists, the CSP continues the proof generation procedure and produces a complete ownership proof.

f) VerifyProof: Before beginning verification, the TPA first verifies the file tag τ as follows:
(a) Recover locally stored file tag hash h
(b) Recover file tag τ from the server
(c) Check if $e(\tau, g) = e(h, \eta)$ holds
(d) If verification passes, output TRUE and recover u and t_d
(e) If verification fails, stop the procedure

If the file tag τ authenticity verification fails, the TPA stops further auditing and reports integrity audit failure to the DP. If file tag authentication passes, the TPA recovers u and t_d fields to use them for authenticating root R_H in θ_f received from the CSP. The pseudocode for integrity verification is shown in Algorithm 4.

Algorithm 4: Integrity Verification
Procedure INTEGRITY_VERIFICATION
Input: θ_f, η, C
Output: TRUE if file blocks pass authentication, else FALSE

1. Initialize pairing: $pbc\_demo\_pairing\_init(pairing, count, param)$
2. Initialize random elements $temp1, temp2, temp3, H(d[i]), Power[i] \in G$ and $b[i] \in Z_p$
3. Set $temp1$ to identity element: $element\_set1(temp1)$
4. Read $H(d[i])$ and μ_i from server; read $b[i], u$ from C
5. Compute $Power[i] = H(d[i])^{b_i}$ and $Power1[i] = u^{b_i}$
6. Generate $\mu_i = H(d[i])^{b_i} \cdot u^{b_i}$
7. Compute $temp1 = \prod_{i \in Q} \mu_i$
8. Apply pairing on $temp1$: $pairing\_apply(temp2, temp1, \eta, pairing)$
9. Apply pairing on ρ: $pairing\_apply(temp3, \rho, g, pairing)$
10. Compare $temp2$ and $temp3$: $result = element\_cmp(temp2, temp3)$
11. If result is 0, output TRUE; else output FALSE

3 Performance Analysis

This section analyzes the performance of RI-MHT, including how computational complexity varies with the number of corrupted blocks. It presents computational costs, storage costs, and communication overhead for different algorithms, along with comparisons to other data integrity auditing protocols.

3.1 Computational Cost Analysis

Different entities in the auditing model incur different computational costs. For the TPA, computational cost refers to resources required to perform one integrity audit. For the CSP, it corresponds to the time needed to process challenge

messages and generate ownership proofs. Computational costs for dynamic data operations differ from regular integrity auditing. During updates, the TPA must generate tags for fresh blocks, while the CSP must regenerate the root hash and update file tags—all of which impose computational burdens on the respective entities.

Table 1 defines the notation used for analysis. Table 2 shows the computational costs for each algorithm in RI-MHT, where n represents the total number of data blocks and t represents the number of challenged data blocks. Table 3 compares computational costs across three protocols.

The results demonstrate that the proposed protocol is more efficient than the other two. Reference [**?**] requires longer computation time for searching data blocks, which grows linearly with block count (i.e., $O(n)$). In the proposed protocol, using relative index fields reduces node search complexity to $O(\log n)$. The search algorithm' s computational complexity is similar to searching for an element in a binary search tree (BST) with $(2n-1)$ nodes, whereas reference [**?**] resembles linear search complexity. Figure 4 [Figure 4: see original paper] shows the proposed protocol' s performance under frequent data modifications, updating a 1 GB file with modified block counts varying from 100 to 1000. As the number of modified blocks increases, reference [**?**]' s computational cost grows at a higher rate than the proposed protocol, while reference [**?**] does not support dynamic data operations.

Table 1: Notation and Descriptions - $H(G)$: Hashing a value into group G - $Add(G)$: Addition operation in group G - $Mul(G)$: Multiplication operation in group G - $Exp(G)$: Exponentiation operation in group G - $Pair(G)$: Pairing operation in group G

Table 2: Computational Costs in RI-MHT - KeyGen: $tAdd(G) + (t-1)Mul(G)$ - FileTagGen: $Exp(G)$ - BlockSigGen: $Exp(G)+nH(G)$ - GenProof: $(n+1)Exp(G)+tExp(G)$ - VerifyProof: $tMul(G)+tEXP(G)$

Table 3: Computational Cost Comparison | Protocol | Time Complexity | |——-|———-| | Reference [**?**] | $tEXP(G) + (t-1)Mul(G)$, $2Pair + (t+2)Exp(G)+(t+1)Mul(G)$, $O(n)$ | | Reference [**?**] | $2tExp(G)+(2t-2)Mul(G)$, $3Pair+(t+1)Exp(G)+tMul$ | | RI-MHT | $tExp(G)+(t-1)Mul(G)$, $4Pair+tExp(G)+tMul(G)$, $O(\log n)$ |

3.2 Storage Cost Analysis

This section analyzes storage and communication costs for RI-MHT. The DP stores the key pair (k, η) and file tag hash h. The delegated TPA stores the public key η and file tag τ. The CSP stores the DP' s data file F, file block tags ψ, root signature ρ, and file tag τ. In storage overhead, metadata size generated during file preprocessing significantly impacts total cost.

In RSA-based protocols, metadata size equals the RSA modulus size. Considering a 2048-bit RSA modulus and 6168-bit prime public key, the protocol yields

a data block size that should remain $||G|| \geq 2^{283}$. If a data block $||d_i|| < 2^{256}$, the protocol may become insecure. Thus, a 64 MB data file must be divided into at least 87,056 data blocks, with each block' s metadata being 2048 bits, resulting in total metadata size of approximately 21.2 MB. RI-MHT uses BLS signature-based homomorphic tags, which are shorter than RSA-based homomorphic tags. Storage costs for RI-MHT are shown in Table 4 , where n is the total number of data blocks, k is the security parameter, l is the file tag size, and $|G|$ is the size of an element in G. Table 4 shows that storage overhead from metadata is constant at the DP and TPA but linear at the CSP. The storage costs for TPA are identical across protocols; the main differences lie with the data owner (DP) and cloud provider (CSP). Thus, RI-MHT offers advantages in overall storage cost, primarily due to smaller generated metadata.

Table 4: Storage Costs in RI-MHT (bits) | Entity | Reference [**?**] | Reference [**?**] | RI-MHT | |—-|————|————|—-| | DP | $k + 2|G| + l$ | $k + 3|G| + 2l$ | $k + 3|G| + l$ | | TPA | $k + |G| + l$ | $k + |G| + l$ | $k + |G| + l$ | | CSP | $n + l + (n + 1)|G|$ | $n + 2l + (n + 1)|G|$ | $n + 2l + (n + 1)|G|$ |

3.3 Communication Cost Analysis

Communication cost constitutes the data transfer volume between CSP and TPA during one auditing instance. In RI-MHT, communication cost primarily comes from the TPA sending challenge messages to the CSP and the CSP sending proof messages to the TPA. Communication overhead from DP delegating auditing tasks to the TPA is constant. Communication overhead among entities in RI-MHT is shown in Table 5 , where $|Z_p|$ is the size of an element in Z_p, t is the number of challenged data blocks, l is the file tag size, and $|G|$ is the size of an element in G.

Table 5: Communication Costs in RI-MHT | Phase | Cost (bits) | |—-|———-| | Delegation (DP→TPA) | $2|G| + |Z_p| + l$ | | Challenge (TPA→CSP) | $t|Z_p| + |G|$ | | Proof (CSP→TPA) | $t|G| + |G| + l$ |

Since the DP shares keys, file tags, and root signatures with the TPA when delegating auditing tasks, this communication cost occurs only once per audit, equal to the size of keys, root signature, and file signature. The TPA sends challenge messages containing subset Q with k elements and random element u to the CSP. Related studies [**?**, **?**] show that Q can be represented as $||Q|| = \frac{\log ||F|| - 3\log(err)}{283}$ bytes, where $||F||$ refers to data file size and err is error probability. Generally, for a file size $||F|| = 1024$ TB, Q size can be expressed as 283 bytes. Thus, in the challenge phase, the maximum communication cost for a file of size 1024 TB is $||Q||_{max} = 1024 \times 283$ bytes.

3.4 Comprehensive Evaluation

The experimental system used a 2.20 GHz Intel Core(TM)2 dual-core processor with 2 GB RAM. The PBC library [**?**] version 0.5.14 was used for pairing operations, and OpenSSL version 1.0.2e was used for hash operations. The protocol

implementation used a 160-bit group order elliptic curve on an Ubuntu 10.04 operating system. For experimental demonstration, file sizes of 10–100 KB were used. Figure 4 shows the TPA' s computational cost when auditing integrity for different numbers of file blocks, demonstrating that RI-MHT imposes the smallest computational burden on the TPA.

To study update operation impact, reference [**?**]' s protocol first searches for file block positions in the MHT, with complexity growing linearly with the number of blocks in the tree, imposing additional computational overhead for search operations. In RI-MHT, node search complexity is reduced logarithmically. As shown in Figure 5 [Figure 5: see original paper], computational costs for references [**?**] and [**?**] increase with file size, while the proposed method imposes minimal computational overhead on the TPA. Therefore, when the DP outsources large data files to the cloud or frequently updates outsourced files, the proposed RI-MHT can significantly reduce computational overhead.

4 Conclusion

This paper proposes a cloud data integrity auditing protocol based on MHT construction and BLS signatures. To reduce search complexity, node relative indices are integrated with hash values in the hash tree. The protocol concatenates a timestamp field to the root hash to ensure data freshness, guaranteeing that users always access the most recent copy of data files. Consequently, the proposed protocol supports efficient public data auditing while ensuring data freshness. Experimental results and security proofs demonstrate that RI-MHT achieves high security and efficiency.

References

[1] Zhang Jianxun, Gu Zhimin, Zheng Chao. Survey on cloud computing research progress [J]. Application Research of Computers, 2010, 27(2): 429-433.
[2] Mell P, Grance T. The NIST definition of cloud computing [J]. Communications of the ACM, 2009, 53(6): 50-50.
[3] Yao C, Xu L, Huang X, et al. A secure remote data integrity checking cloud storage system from threshold encryption [J]. Journal of Ambient Intelligence & Humanized Computing, 2014, 5(6): 857-865.
[4] Xu Ke, Liu Xuchong, Fu Zhen' ai, et al. Optimization of network information encryption RSA algorithm for operation speed and confidentiality [J]. Bulletin of Science and Technology, 2015, 34(7): 144-147.
[5] Shacham H, Waters B. Compact proofs of retrievability [J]. Journal of Cryptology, 2013, 26(3): 442-483.
[6] Erway C C, Papamanthou C, Tamassia R. Dynamic provable data possession [J]. ACM Trans on Information & System Security, 2015, 17(4): 15-26.
[7] Xie Fei. Trusted cloud computing information security proof method based on Merkle hash tree [J]. Laser Journal, 2016, 37(11): 122-127.
[8] Wang J, Chen X, Huang X, et al. Verifiable auditing for outsourced database

in cloud computing [J]. IEEE Trans on Computers, 2015, 64(11): 3293-3303.
[9] Li Ling. Research on several issues of data security in cloud computing services [D]. Hefei: University of Science and Technology of China, 2013.
[10] Gong Junqing, Qian Haifeng. Efficient sanitizable digital signature scheme with full confidentiality [J]. Application Research of Computers, 2011, 28(1): 312-317.
[11] Wang Qiufen, Liang Daolei. A greedy algorithm for constructing optimal binary search trees [J]. Computer Applications and Software, 2013, 30(7): 57-61.
[12] Etemad M M. Generic efficient dynamic proofs of retrievability [C]// Proc of ACM on Cloud Computing Security Workshop. London, UK: IEEE press, 2016: 85-96.
[13] Liu Aifen. Efficient dynamic ciphertext search method in cloud environment [D]. Shenyang: Northeastern University, 2013.
[14] Shelat A, Shen C H. Two-output secure computation with malicious adversaries [M]// Advances in Cryptology -EUROCRYPT 2011. Springer Berlin Heidelberg, 2011: 386-405.

Note: Figure translations are in progress. See original paper for figures.