2019 2nd International Conference on Computer Science and Advanced Materials (CSAM 2019)

Research on Cloud Computing Data Storage Security Based on Hash Authentication Tree

Liu Hongqing

Hunan Vocational College of Modern Logistics, Changsha, Hunan, 410131, China

Email:158140027@qq.com

Keywords: Cloud Calculation, Data Storage, Chameleon Authentication Tree, Third Part Audit, Quasi Random Function

Abstract: In order to improve the security of data storage in cloud calculation, a chameleon Hash authentication tree optimization audit method for data storage security in cloud calculation is proposed. First, an optimized public audit agreement is proposed. By storing homomorphic linear validator for user data on TPA sites, the response size of cloud storage server (CSS) is optimized. At the same time, the quasi-random function is used to optimize the query request to CSS; secondly, the chameleon hash and an improved chameleon authentication tree are used to perform efficient dynamic data updating on client data (cloud calculation) to support block-level updating and fine-grained updating; finally, through thorough security and performance analysis, it is clearly verified that the proposed method is safe and efficient.

1. Introduction

Cloud computing is an Internet-based computing model that is one of the important directions in the development of current computing technology. It is considered to be the next-generation architecture of the IT industry with many advantages, such as on-demand self-service, extensive network access, quick response, location-independent resource pool and etc. [1~2]. Data is moved to the cloud. Computing resources and services are used in the pay-for-use mode and the platform as a service (PaaS) form. Users can easily access and use resources without considering the complex hardware management. This is much less costly than their own building of IT infrastructure [3].

2. Problem description

2.1 System model

The network architecture for cloud data storage is shown in Fig. 1. It has three different network entities that can be identified as follows:

(1) Client/Cloud-User: The entity can be a single consumer or organization, has a large number of data files stored in the cloud, and relies on the cloud for data maintenance and computing.

(2) Cloud Storage Server (CSS): An entity that has significant storage space, computing power and resources to maintain customer data. It is managed by a cloud service provider (CSP). Here, we will not make a distinction between CSS and CSP.

2.2 Safety threat

We considered three types of data/metadata attacks in this paper. They are:

(1) Replace Attacks: Malicious CSS may omit questionable data block (m_i) or its metadata (σ_i), and replace with another pair of valid and undamaged data block (m_i) and metadata (σ_i) for passing the audit.

(2) Forge Attacks: Malicious CSS or auditor (TPA) cannot forge metadata (HLAS), which may lead to unreasonable and unsatisfactory data auditing.

(3) Replay Attacks: Malicious CSS uses the evidences generated by data not updated or previous data and previous evidences or other information. It does not query the real data of the client.

2.3 Design objectives

Our design objectives are as follows:

(1) Optimized public auditability of cloud data storage (OPACDS): it permits that TPA may be authorized to verify the correctness of client data stored on the cloud. Such process requires no retrieval of it. No additional online client/cloud-user will be generated.

(2) Support for dynamic data update operations: it permits that cloud-user/client executes block level and fine grit updating on its files with a method as efficient as possible.

2.4 Chameleon authentication tree

Chameleon authentication tree is a generalized Merkle Hash Tree. The right child of each node is equal to the Chameleon Hash value of its child node. The left child of each node is equal to the simple Hash value stitched in series of the child. Fig. 1 shows an improved version of the Chameleon authentication tree structure (mCAT) proposed in this paper.

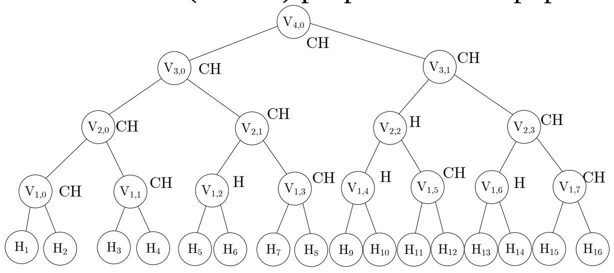

Fig. 1 An improved version of Chameleon authentication tree structure

The improved version of the Chameleon authentication tree (mCAT) structure as shown in Fig. 1 will be detailed in the next section.

3. mCAT-based public auditing and dynamic data update

To ensure the integrity of the data, an optimized public auditing and dynamic data update scheme is proposed. It consists of three phases:

(1) Setup phase: This phase includes key generation, file preprocessing of production block metadata (HLAS), mCAT structure generation of files and authorization to TPA.

(2) Dynamic data update: At this stage, the client executes the stored data updating of block level and fine grit by using MCAT algorithm in the cloud. It then computes the new HLA of the modified block and stores the same on the TPA site.

(3) Third-party auditing: At this stage, an authorized third-party auditor (TPA) sends a query request to the CSS. CSS returns an integrity certificate that corresponds to the set of challenged blocks and is returned to the TPA. TPA verifies the integrity of the challenged block set thereafter.

3.1 Setup phase

It is supposed that G is a p-order prime set consisting of Z_p. Where, p is a big prime. $H:(0,1)^* \to G$ is an anti-collision hash function. $h()$ is a cryptographic hash function. $ch()$ is a Chameleon hash function, and $e:G\times G \to G_T$ is a bilinear map. The client first negotiates these parameters with other parties (CSS and TPA) and then runs the following algorithms:

$KeyGen(1^\lambda)$: The client generates the value $\alpha \in Z_p$ and the generator $g \in G$. α becomes a part of the key. (g, g^α) becomes a part of the public key. The client uses $chGen(1^\lambda)$ to generate a key pair (csk, cpk) for Chameleon Hash function $ch()$. Where, csk is the key. cpk is the public key of this Chameleon Hash function. Finally, the client's key is: (α, csk), and the public key is: (g, g^{α}, cpk).

Generate HLA: The client divides the file F into n blocks of equal size. Thus, there is $F = (m_1, m_2, m_3, \cdots, m_n)$, and there is $m_i \in Z_p$. It generates a random number $u \leftarrow G$. Then, for each data block, the client generates a homomorphic linear authentication (HLA): $\sigma_i \leftarrow (H(m_i) \cdot u^{m_i})\alpha$. This results in n group(s) of HLAs, the set of which can be expressed as $\phi = \{\sigma_i\}, 1 \le i \le n$. The client then creates a mCAT whose leaf nodes are hashed file blocks $(h(m_i), 1 \le i \le n)$.

Authorize TPA: The customer selects a TPA and asks for its ID. The client generates an authorization message k_{auth} and generates a signature $sig_{auth} = (k_{auth} \| ID)^{\alpha}$. It sends the signature sig_{auth}. The audit delegate queries to the TPA.

Finally, the customer stores $\{F, T, k_{auth}\}$ of CSS and $\{\phi, u\}$ of TPA. $\{F, T, k_{auth}, \phi\}$ is then deleted from its storage memory. This ensures that the mCAT algorithm uses only nodes that are not defined for use.

3.2 Dynamic data update phase

At this stage, the client/cloud-user interacts with CSS to execute updates of their data. The update phase supports the following update requests: PM (partial modification of the data block) and R (replacement of the data block). Considering Fig. 2, the dynamic data update process of PM is as follows:

(1) The client writes the request: $BlockRequest = \{i\}$, where i is the index of the block to be modified, and sends it to CSS.

(2) CSS receives the BlockRequest and executes the following operations: CSS locates m_i, and computes the mCAT authentication path for m_i:

$$Response = \{m_i, h(m_{adj}), aPath, sig_c(aPath), v_{h,0}\} \quad (1)$$

Where, m_{adj} is the brother node of m_i. $aPath$ is the verification path of the data block m_i. $sig_c(aPath) = (aPath)^{\alpha}$ is the customer signature on $aPath$. $v_{h,0}$ is the leftmost node, which can be moved from $h(m_i)$ along the path to the root node. Finally, it sends a response to the client/cloud-user.

(3) After receiving the response, the client executes the operation VerifyAuth: First, the client verifies the aPath signature. That is to check:

$$e(sig_c(aPath), g) == e(aPath, g^{\alpha}) \quad (2)$$

If the test fails, the client interrupts the process; otherwise, the client computes $h(m_i)$. The client then uses $h(m_{adj})$ and $h(m_i)$ to compute $v'_{h,0}$. Finally, the client checks if the following condition is met: $v_{h,0} == v'_{h,0}$.

If the condition $v_{h,0} == v'_{h,0}$ is still not met, the client interrupts the process; otherwise, the client computes m'_i and $h(m'_i)$, and uses them to generate a collision R value (using $col()$ function) for the first node ($v'_{h,i}$ in the CAT). Such value can be computed using the Chameleon Hash function. It depends on the path from the node $h(m'_i)$ to the remaining nodes $v_{h,0}$ ($h \ge h'$). The client then uses r' to update the verification path $aPath \to aPath'$, and computes the signature of the path $aPath'$, i.e. $sig_c(aPath') = (aPath')^{\alpha}$. The client forms an update request:

$$UpdateRequest = \{PM, i, o, data, aPath', sig_c(aPath')\} \quad (3)$$

The update request is sent to CSS. CSS starts with the offset o in m_i. The data block m_i will be updated to m_i' with the data. At the same time, it uses the new random number in $aPath'$ to update the authentication path in mCAT, and then replaces the old $sig_c(aPath)$ with $sig_c(aPath')$. A "update successfully" response is then sent to the client.

(4) After receiving the “update successfully” response from CSS, the client computes a new HLA: $\sigma_i' \leftarrow (H(m_i') \cdot u^{m_i'})^{\alpha}$ for the modified data block m_i', and sends it to TPA. TPA replaces the old $HLA(\sigma_i)$ with the new $HLA(\sigma_i')$. The client deletes m_i' from memory thereafter.

3.3 Third party auditing phase

At this stage, the authorized TPA (sig_{auth} to be sent by the client) works with CSS to determine the integrity of the client’s data. It mainly executes three operations:

(1) Challenge request: This operation is executed by TPA. It uses client pair $\{\varphi, u, sig_{auth}\}$ for sharing during the installation phase. TPA takes its ID and encrypts it with the public key of CSS: $\{ID\}_{PKCSS}$. TPA now uses quasi-random function $f_s()$ to generate c different block indices. It is supposed that $I \subseteq [1, n]$ is the set of c block indices generated by quasi-random function $f_s()$. TPA now selects c random coefficients as $\{v_i \in Z_p\}_{i \in I}$. Finally, it forms a challenge request $ChallengeReq = \left\{\{ID\}_{PKCSS}, \{i, v_i\}_{i \in I}, sig_{auth}\right\}$, and sends it to CSS.

(2) Integrity proof: After receiving the challenge information $ChallengeReq = \left\{u, \{ID\}_{PKCSS}, \{i, v_i\}_{i \in I}, sig_{auth}\right\}$, CSS verifies sig_{auth}. To this end, CSS first decrypt ID with its private key SK_{CSS} and k_{auth} (i.e. $k_{auth} \| ID$). This is then verified by the following equation:

$$e(sig_{auth}, g) = e(k_{auth} \| ID, g^{\alpha}) \quad (4)$$

If $signature(sig_{auth})$ verification fails, it will reject $ChallengeReq$. Otherwise, CSS computes $\mu = \sum_{i \in I} v_i m_i \in Z_p$ and $\xi = \prod_{i \in I} H(m_i)^{v_i}$, where $i \in I$. Finally, it sends the evidence $IntegrityProof = \{\mu, \xi\}$ to TPA.

(3) Integrity verification. After receiving the $IntegrityProof = \{\mu, \xi\}$ information from CSS, TPA executes the operation $VerifyIntegrity$. It contains: TPA uses $\{i, v_i\}_{i \in I}$ and stores $\sigma_i (i \in I)$ of the file block, computes $\sigma = \prod_{i \in I} \sigma_i^{v_i} \in G$, and verifies whether the following equation is satisfied:

$$e(\sigma, g) = e(\xi . u^{\mu}, g^{\alpha}) \quad (5)$$

If this is not the case, it means that the client data stored on the CSS violates the integrity requirement. TPA is responsible for reporting the same to the client.

4. Conclusion

This paper optimized the public auditing of client data in the cloud environment by migrating HLA from the CSS site to the TPA site. We also proposed a dynamic data update protocol using the improved Chameleon Authentication Tree (mCAT). Here, we demonstrate the security of our optimized audit protocol, showing that it can withstand replay, replace and forge attacks. At the same time, the validity of the algorithm and the validity of the integrity verification are analyzed and proved by theoretical analysis.

References

[1] Xu Fenggang; Xu Junkui; Pan Qing; an improved algorithm of extensible Hash method [J]; computer engineering and application; 2004

[2] Chen Tao; Shawn; Liu Fang; Fu Changsheng; Data Layout Algorithms Based on Clustering and Consistent Hash [J]; Journal of Software; 12 issues 2010