Future Generation Computer Systems 80 (2018) 640–652

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Full integrity and freshness for cloud data

Hao Jin [a], Ke Zhou [a,*], Hong Jiang [b], Dongliang Lei [a], Ronglei Wei [a], Chunhua Li [a]

[a] *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*
[b] *Department of Computer Science and Engineering, University of Nebraska-Lincoln, NE, USA*

HIGHLIGHTS

- We design a secure storage system for cloud data to ensure the security of hot data and cold data.
- We integrate proofs of storage technique and signature based method in a secure storage system to provide full integrity for outsourced data.
- We achieve data dynamics support in our integrity auditing scheme.
- We provide instantaneous freshness check for retrieved data to resist potential replay attacks.

ARTICLE INFO

Article history:
Received 28 August 2015
Received in revised form
18 May 2016
Accepted 17 June 2016
Available online 25 June 2016

Keywords:
Confidentiality
Full integrity
Data freshness
Proofs of storage
Integrity auditing

ABSTRACT

Data outsourcing relieves cloud users of the heavy burden of infrastructure management and maintenance. However, the handover of data control to untrusted cloud servers significantly complicates the security issues. Conventional signature verification widely adopted in cryptographic storage systems only guarantees the integrity of retrieved data, for those rarely or never accessed data, it does not work. This paper integrates proof of storage technique with data dynamics support into cryptographic storage design to provide full integrity for outsourced data. Besides, we provide instantaneously freshness check for retrieved data to defend against potential replay attacks. We achieve these goals by designing flexible block structures and combining broadcast encryption, key regression, Merkle hash tree and proof of storage together to provide a secure storage service for outsourced data. Experimental evaluation of our prototype shows that the cryptographic cost and throughput are reasonable and acceptable.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Outsourcing data to the cloud brings the advantages of on-demand service, universal access, usage-based pricing, reduced costs of infrastructure construction and maintenance. However, data outsourcing deprives users of their direct control over data, which makes them vulnerable to many security threats. Despite of the more powerful machines and stronger security mechanisms provided by the cloud service provider (CSP), data in the cloud still face network attacks, hardware failures and administrative errors. Amazon's data corruption on its Simple Storage Service (S3) and Google Doc's information leakage indicate that serious concerns for cloud data security still remain.

Confidentiality and integrity are usually considered as two basic important properties in secure storage systems. Confidentiality requires that unauthorized users cannot learn any information about owners' data, which is usually achieved by encrypting data using symmetric keys. Integrity requires that any illegitimate modification of data can be detected, which is usually achieved by generating signatures or message authentication codes (MAC) on the data for later verification. Storage systems that rely on these approaches to guarantee data's security are called cryptographic file systems (CFS) [1].

However, CFS only ensures the integrity of retrieved data, for rarely or never accessed data, they do not work. Since signature or MAC based mechanisms require users to download the data locally to re-compute a signature or MAC value for comparison, for large-scale data, it is impossible for users to retrieve all the data to check its integrity due to the high cost in bandwidth. Furthermore, in the cloud environment, there may be a significant portion of data never or rarely accessed, e.g., archive data for long-term backup. In this context, relying on signature verification to guarantee the integrity of large-scale cloud data is an inflexible solution. A better way should be able to differentiate between frequently retrieved data (hot data) and rarely retrieved data (cold data), and provide a flexible method to check their integrity.

* Corresponding author.
E-mail address: khow.hust@gmail.com (H. Jin).

http://dx.doi.org/10.1016/j.future.2016.06.013
0167-739X/© 2016 Elsevier B.V. All rights reserved.

Proofs of storage (PoS) [2–5] is a secure auditing mechanism to check the integrity of remotely stored data without downloading them locally, which complements signature based mechanisms very well. With such auditing mechanisms, users can periodically interact with the CSP through auditing protocols to check the correctness of their outsourced data, especially those rarely accessed data. In our design, we combine CFS based method with proofs of storage technique seamlessly in a secure storage system to provide full integrity for both hot data and cold data, namely, we provide different integrity check mechanisms for data with different access frequencies. Moreover, one critical and subtle related property, data freshness, which ensures the retrieved data reflect its latest update, is often ignored. Only limited research works [6,7] address this problem in their storage system designs. To prevent the cloud from launching a replay attack by sending stale data to users, there must be some mechanism to check the freshness of retrieved data. Unfortunately, signature verification and PoS auditing are insufficient to address this problem due to the following reasons: (1) Signature verification usually re-computes the signature from the downloaded data and compares it with the one attached with the data, but a stale version of data with its corresponding signature can also pass such a verification. (2) PoS technique verifies the integrity of remotely stored data by verifying the integrity proof computed by the cloud, but the fact that data is intact in the cloud does not necessarily mean that the cloud will send the latest data to a user upon a request. We address this issue by combining key regression [8] technique with Merkle hash tree [9] to build a version authentication tree for each block group, and rely on the user to verify the tree root signed with a rotatable key to guarantee the freshness of retrieved data blocks.

In cloud storage, data owners and the cloud belong to different trust domains. Owners usually do not put full trust on the cloud. Therefore, just letting the cloud manage the security mechanisms is not a feasible way. From recent studies on secure storage systems [10,6,11,7], a user-centric trend can be found: that providing data owners with direct control over the access control policy is of much significance for protecting data stored on untrusted cloud servers, which can more or less compensate for their lost control over the data. Our design also follows such a user-centric principle. Specifically, we employ broadcast encryption [12] and key regression [8] to secure the key storage and update, where it is the owner's responsibility to manage permission authorization and revocation, to update the encryption keys and signing keys.

Generally, our work address the problem of confidentiality, full integrity and freshness in a secure storage system design to provide strong protection for data outsourced to the cloud. And we achieve following security goals.

Confidentiality. Data are encrypted by the owner before being outsourced to the cloud, thus unauthorized users or even the cloud cannot learn data content without corresponding keys. Keys are securely stored in its metadata using broadcast encryption, so that only authorized users can decrypt the broadcast message to recover keys.

Full integrity. We efficiently integrate signature verification and proofs of storage technique into a secure storage system design to provide full integrity for outsourced data. For frequently retrieved data, integrity could be guaranteed by verifying the validity of attached signatures; and for rarely retrieved data, PoS auditing could guarantee its integrity. Moreover, we adapt existing PoS schemes to provide efficient support for data dynamic operations, which is a challenging task in the research of integrity auditing.

Freshness. We devise new authentication structure to provide instantaneous freshness check for retrieved data to resist potential replay attacks from the CSP, which is especially important for companies that rely on the real-time transmission of data to make business decisions.

2. Background and related work

2.1. Secure storage systems

Traditional cryptographic file systems [1] generally aim to provide confidentiality and integrity protection for local storage systems, where data access events always occur in the same trust domain. Based on studies of CFS [13,14], several secure storage systems have been proposed recently, such as Cepheus [15], SNAD [16], Plutus [10], SiRiUS [6] and SUNDR [17,11]. These systems are designed to secure data storage on untrusted servers. They provide read and write differentiation by choosing different keys for encryption and signing, and a lockbox like structure is adopted to secure the storage of keys.

Among these systems, Plutus [10] emphasizes on the secure sharing of encrypted data, where key rotation is used to facilitate key management so that lazy revocation can be easily implemented for a block group. SiRiUS [6] is designed on top of existing network file systems, which uses a hash tree built from blocks' metadata files to guarantee the freshness of a directory's metadata. However, the key box structure is not scalable because its number is linear to the number of users, and SiRiUS only guarantees metadata freshness. SUNDR [17,11] is the first secure system to provide well-defined consistency (fork consistency) semantics for untrusted storage. All files of a user with write privilege are aggregated into a hash value called i-handle using hash trees, and a version vector ties all i-handles together. Consistency check is conducted by checking the partial order between version vectors before and after a write. However, such a delicate and complicated structure involves all users and files. As a result, it is not scalable for cloud storage with thousands of users accessing millions of files, because the version entries would grow too large. Furthermore, SUNDR cannot guarantee data freshness and is not likely to be easily extended to support it.

CloudProof [7] is a cryptographic cloud storage system designed to meet Service Level Agreements, where users can detect violations of integrity, freshness and write-serializability through auditing, and any misbehavior can be proved to a third party. CloudProof adopts verifiable attestations and chained hash to periodically audit access records of users and CSPs to detect violations of security. But their freshness guarantee is actually a post-check measure, which is of limited meaning for data-based business decisions. CS2 [18] combines symmetric searchable encryption (SSE), search authenticators and PoS technique to provide verifiable search on encrypted data and global integrity—a similar concept to full integrity in our work, but their work mainly focus on the search of encrypted data, while our work focus on the secure storage and auditing of cloud data. Other cloud based but less related secure storage systems can be found in [19–21]. Due to space limitations, we do not introduce these designs here.

2.2. Proofs of storage

Proofs of Storage [5] can be traced back to remote integrity check schemes [22–24], which provide users the ability of verifying the integrity of their outsourced data without downloading them locally. There are mainly two kinds of variants around this issue: Proofs of Retrievability (PoR) [2,25] and Provable Data Possession (PDP) [3,5]. Roughly speaking, a PoR is a challenge–response protocol that enables a service provider to demonstrate to a client that his data is intact, or retrievable if any data corruption is detected in auditing. A PDP is a similar auditing protocol with weaker guarantee that only detects data corruption in outsourced data. Extensions and improvements to PDP and PoR could be found in work [26–28,4]. Our system adopts PDP in that PoR requires the use of erasure code on outsourced data, which makes the update

of auditing data troublesome and inefficient. Generally, there are several trends in the development of auditing schemes.

From deterministic proof to probabilistic proof. Earlier schemes [22,24,28,27] usually generate a deterministic proof on data integrity by letting the storage service provider access the entire data file, e.g., treat the whole file as an exponent to compute the proof. However, such plain solutions bring expensive computation overhead at the server side, and they lack practicality when dealing with large-size data files. In contrast, represented by "sampling" method in PoR and PDP models, later schemes [2,26,3,4] all provide probabilistic proofs by letting the CSP access part of the file, which greatly enhances auditing efficiency over earlier schemes.

From private verifiability to public verifiability. Private verifiability [22,24] refers to let the data owner who possesses the private key to perform the integrity auditing only, which may potentially overburden the owner. While schemes with public verifiability [3,4,29] allow anyone who has the public key to perform the auditing, which makes it possible for auditing tasks to be delegated to a third party auditor (TPA). A TPA is a professional institute with computation and storage capacity that users do not have, who performs the integrity auditing for owners and honestly reports the result to them.

From static data to dynamic data. Earlier schemes [2–4] intend to audit static archive or backup data, where users usually read these data and seldom update them. So these schemes mainly focus on static data auditing and do not support data dynamic operations. But from a general perspective, data update is a very common requirement for cloud applications. So it is imperative for auditing schemes to provide data dynamics support. To our knowledge, only schemes in [29–33] address this problem, yet they are insufficient in providing both data dynamics and auditing efficiency simultaneously, especially for large-size data files.

Among these trends, enabling an auditing scheme with data dynamics support may be the most challenging. Dynamic PDP scheme proposed in [29] only supports limited operations and block insertion is not supported. Erway et al. [30] use the rank based skip list to uniquely differentiate among blocks, their communication cost is linear to the number of challenged blocks, and there is no explicit implementation of public verifiability in their scheme. Wang et al. [31] propose to replace $H(i)$ with $H(m_i)$ in tag computation, but it requires all data blocks to be different, which is not appropriate since the probability of block resemblance increases when the block size decreases. Zhu et al. [32] use index-hash table to construct their dynamic auditing scheme based on zero-knowledge proof, which is similar to our PoS construction in terms of index differentiation and avoidance of tag re-computation. Zheng et al. [33] construct a dynamic auditing scheme with fairness guarantee to prevent a dishonest client accusing an honest CSP. Their scheme only realizes private auditing and is difficult to be extended to support public auditing.

2.3. Notation and preliminaries

Merkle hash tree. A Merkle Hash Tree (MHT) [9] is a well-studied authentication structure used to prove that a set of elements are unaltered. It is usually constructed as a binary tree where the leaf nodes store the hashes of data elements and the non-leaf nodes store the hashes of its two children. Generally, MHT can be used to authenticate the values and logical positions of data blocks in storage systems. Fig. 1 depicts an example, where the verifier with a root h_r requests for elements $\{v_2, v_5\}$. The prover provides the auxiliary authentication information (AAI) $\Omega = \{h(v_1), h_c, h(v_6), h_f\}$ and sends it to the verifier. The verifier now can verify v_2 and v_5 by computing $h(v_2), h_b = h(h(v_1) \parallel h(v_2)), h(v_5), h_e = h(h(v_5) \parallel h(v_6)), h_a = h(h_b \parallel h_c), h_d = h(h_e \parallel h_f)$ and $h_r = h(h_a \parallel h_d)$. Then the verifier can check if the computed root h_r is the same as the one he holds for authentication. In this paper, we further employ MHT to construct a version authentication tree for a block group, and combine it with key regression technique to verify the freshness of retrieved data blocks.

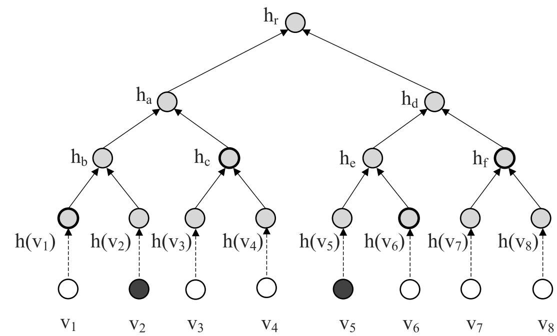


Fig. 1. Merkle hash tree for element authentication.

Key regression. Key regression [8] is an updated version of key rotation technique appeared in [10], which allows a sequence of keys to be generated from an initial key and a secret master key. Only the possessor of the secret key can rotate the key forward to a new version in the key sequence, and users knowing the current key can produce all earlier versions of the key in the sequence. Combined with lazy revocation policy in cryptographic storage, key regression technique can efficiently reduce the number of keys to be retained after multiple revocations and provide a scalable model for access control.

Broadcast encryption. Broadcast encryption was firstly introduced in [34] and improved in [35,36,12], which let an owner encrypt a piece of data to a subset of users. Only users in the subset can decrypt the broadcast encrypted message to recover the data. In cryptographic storage, rather than directly encrypting the data content, broadcast encryption can be used to securely store a key so that only authorized users can recover the key by decrypting the broadcast message, whereas revoked users cannot find sufficient information to do so.

Bilinear map. A bilinear map is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are two Gap Diffie–Hellman (GDH) groups of prime order p, and $\mathbb{G}_T$ is another multiplicative cyclic group with the same order. A bilinear map has the following properties [37]: (i) Computable: there exists an efficiently computable algorithm for computing e; (ii) Bilinear: for all $h_1 \in \mathbb{G}_1, h_2 \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, $e(h_1^a, h_2^b) = e(h_1, h_2)^{ab}$; (iii) Non-degenerate: $e(g_1, g_2) \neq 1$, where g_1 and g_2 are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$.

3. Design and implementation

3.1. Security architecture and threat model

The security architecture is illustrated in Fig. 2, there are four roles in our system: (i) Data owner, who has large amount of data to be stored in the cloud, and is responsible for granting and revoking users' access permissions; (ii) Data users, who read or write the outsourced data according to their access rights granted by the owner; (iii) Cloud Service Provider (CSP), who has massive storage space and computation power that owners do not possess, stores and manages owner's data; (iv) Third party auditor (TPA), who is a professional institute with expertise and capabilities for integrity auditing, and is trusted and payed by the owner to check the integrity of owners' data.

Our design mainly focuses on the secure storage, integrity auditing and freshness check of client's data against a semi-trusted

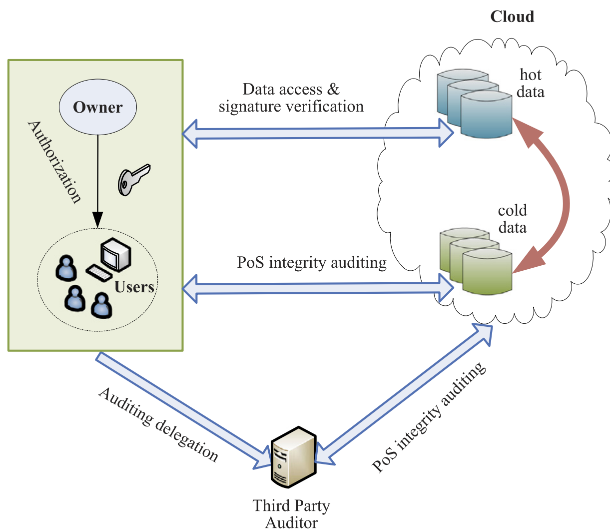


Fig. 2. The security architecture.

CSP. So in our threat model, we assume the owner and users are honest, they store the data on CSP's servers and may update their data now and then. On the other hand, we assume the CSP is not fully trusted, it behaves properly as prescribed contracts most of the time, but it also has the motive to disclose, modify or replay owner's data. Both the owner and users may undertake the task of PoS auditing for rarely accessed data, or delegate auditing tasks to the TPA. We consider the TPA honest-but-curious, that is, it behaves honestly most of the time but it is also curious about the content of the auditing data, thus the privacy protection of the auditing data should be considered.

Finally, we assume the existence of a public key infrastructure (PKI) to generate public–private key pairs for each party and provide a strong binding to each party's id. Each party can easily signs its message with its private key and others can verify the signature with his public key.

3.2. Design overview

Compared with centralized key management policies widely adopted in traditional cryptographic storage systems, decentralized and user-oriented key management policies are receiving more and more attentions in recent secure storage designs, such as Cepheus [15], SNAD [16], Plutus [10] and SiRiUS [6]. In these systems, differentiation of read and write access to encrypted data are usually achieved by providing different keys for read and write access. Specifically, a symmetric key is used for data encryption and decryption, which is necessary for read access. And an asymmetric key-pair is used for signing and verification, which is to guarantee data integrity and is necessary for write access. Furthermore, these systems tend to use a lockbox structure to securely store the keys shared among a group of users by encrypting these keys with each user's public key, and storing one lockbox per user as attached metadata. As a result, the trivial task of real-time key distribution by an online owner or a key distribute center is avoided. However, in such designs, the number of lockbox is linear to the number of users. Revoking a user from a group needs to generate a new key, re-encrypt the lockbox for each user and re-encrypt the data with the new key, which bring about huge computation overhead and may limit its usage in cloud storage.

In our construction, besides providing read and write differentiation for encrypted data, we employ broadcast encryption [12] to secure the key storage and distribution, and use key regression [8] to reduce the number of keys to be maintained. Specifically, for each block group, there is an encryption key K_E and an asymmetric key pair (K_S, K_V) for signing and verification. A user with read privilege should have K_E and K_V, and a user with write privilege should have K_E, K_V and K_S. In this sense, the write privilege implies the read privilege. Table 1 lists the notations in our design. Note in the following sections, we will use the terms "read key" and "encryption key" interchangeably, so are the terms "write key" and "signing key".

Table 1
Notations and keys.

Notation	Description
K_E	Encryption key for a block group
K_S, K_V	Signing and verification key for a block group
K_T^R, K_T^U	Root signing and verification key for a block group
K_{cloud}^R, K_{cloud}^U	CSP's private–public key pair
K_{owner}^R, K_{owner}^U	Data owner's private–public key pair
G_R	Reader set of a block group
G_W	Writer set of a block group
$E(D, K_E)$	Encrypt data with the read key K_E
$E_B(K, G)$	Broadcast encryption of key K to a user set G
$h()$	A collision-resistant hash function
$MHT\{h(\cdot)\}_{1\le i\le n}$	Hash tree root computation for a block group
VAT	Version authentication tree for a block group
GAT	Gid authentication tree for a block group
gid	Global id of a data block
sig	Signature of a data block signed with K_S
tag	A block's tag used for PoS auditing
coldness	Time interval that a block has not been accessed

Since broadcast encryption let a broadcaster (data owner) encrypt a message to a subset of users and only qualified users in the subset could decrypt the broadcast message to recover the content, we then use it to encrypt the read key and write key to a subset of readers and writers. Thus, only users with read privilege (in the read set G_R) can use their private keys to decrypt the broadcast message $E_B(K_E, G_R)$ to get the encryption key K_E, and only users with write privilege (in the write set G_W) can decrypt the broadcast message $E_B(K_S, G_W)$ to get the signing key K_S. These broadcast messages are stored in the admin block of the block group. Additionally, key regression technique enables the admin block store only the latest version of the read key K_E and the write key K_S. Due to the forward secrecy property of key regression, all earlier versions of the key can be recovered by rotating the current key backward, and only the owner who has the secret master key can rotate a read key or write key forward to a new version. Combining broadcast encryption with key regression in cryptographic storage has also appeared in [7,38].

As depicted in Fig. 3, a block group consists of one admin group and multiple data blocks. We organize data blocks in the following format: (*encrypted block, metadata, signature, tag*). A block's metadata mainly includes a global id (gid) which uniquely identifies a data block, a version number indicating the update times, read key version and write key version. Finally, a signature computed from the concatenation of the encrypted block and its metadata is attached, along with a tag computed from the encrypted data block only, which is used for PoS auditing. The admin block consists of a version number, the broadcast encryption messages of latest read key K_E and write key K_S, the signature verification key K_V in plaintext, and the newest version number of K_E and K_S, finally a signature signed with the owner's private key is attached.

After outsourcing data to the cloud, users have to rely on the real-time data transmission by the cloud to retrieve their data, which brings up another problem: how to ensure the freshness of retrieved data? On this issue, signature or PoS technique is insufficient, because they can only guarantee the integrity of retrieved or remotely stored data, but cannot guarantee that instantly transmitted data to reflect their latest update.

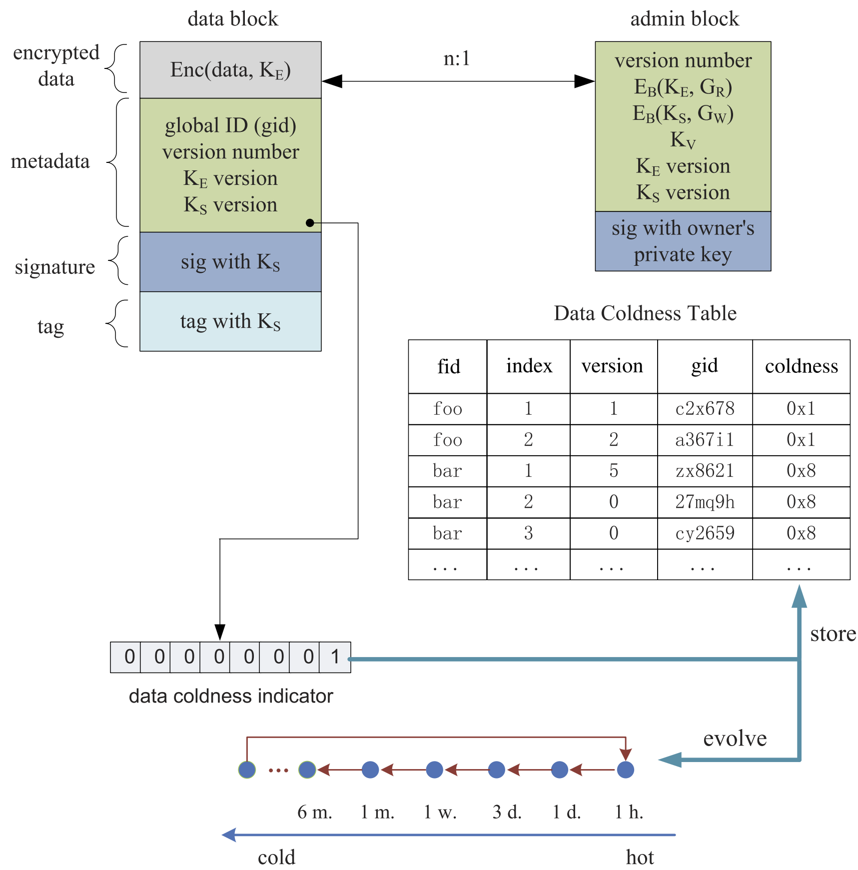


Fig. 3. Data organization of a block group.

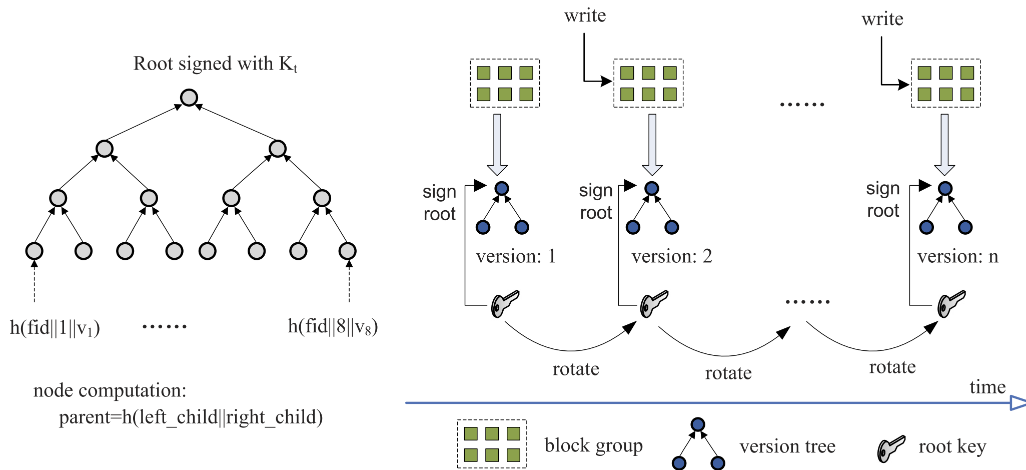


Fig. 4. Version authentication tree and key regression.

To address this problem, we employ the Merkle hash tree [9] to construct a version authentication tree (VAT) for each block group. To efficiently verify the freshness of a retrieved data block, we do not store block hashes in a VAT's leaf nodes. Instead, we store the values of $h(fid\|i\|v_i)$, where *fid* denotes the file name and i refers to a block's index in the file, v_i refers to a block's version number, as illustrated in Fig. 4. Then the VAT root can be computed as $r = MHT\{h(fid_i\|i\|v_i)\}_{1\le i\le n}$, where n denotes the number of blocks in the group. Furthermore, we adopt key regression [8] technique to sign the tree root with the latest version of a root signing key K_T, which is used to sign the VAT root r of a block group. Whenever a data block in a group is updated, the version number of this block will be incremented by one, so the version authentication tree of the group also needs to be updated. Meanwhile, the root signing key should be rotated forward to a new version K_T' and used to sign the new VAT root r'.

3.3. Revocation

For group based cryptographic storage, blocks in a group are usually encrypted with the same key, which can efficiently reduce the number of encryption keys. On the other hand, users are also grouped according to their access privileges, users with similar privileges to the same group of blocks are always organized into

a user set. Generally, when a user is revoked from a user set, it should be guaranteed that the revoked user cannot access the data anymore. Around this issue, there are two methods. Immediate revocation implies a revoked user will not be able to access the data right after the revocation, which involves the instant re-encryption of the data. While lazy revocation [10] allows a revoked user still accessing the unmodified version of the data and delays the re-encryption till the next write operation. The philosophy behind lazy revocation is that it will not cause significant loss to security for a revoked user being able to read data blocks he has already read before, but the updated version of those blocks should be prohibited to be read by the revoked user. Because lazy revocation can significantly save computation cost in cryptographic storage, it is widely adopted in current CFS designs [10,39,7,40,38].

However, lazy revocation policy can cause the number of keys to increase after multiple revocations, namely, the blocks in a group will be encrypted with different versions of keys. If revocations occur frequently, there will be a significant number of keys to be retained, which may violate the design intention of reducing keys number by grouping blocks together. Besides, how to securely store and manage these keys will also be a tedious task.

In our design, we combine broadcast encryption with key regression to accomplish the task of key generation, update and storage. For each block group, the newest version of encryption key and signing key is broadcast encrypted and stored in the admin block of the group, only authorized readers and writers can decrypt the broadcast message to get the corresponding key. If a block is encrypted or signed with an old version of key, then the user can rotate the newest encryption key or signing key backward to get the correct version of key, the rotation times are decided by the current key version in the block's metadata and the newest key version in the admin block.

In encrypted storage, writing a data block needs to decrypt the block using the read key, then the modification can be performed on the plaintext content of the block. Additionally, each write to a data block incurs the re-encryption and re-signing of the block. The advantage of lazy revocation is that it pushes the burden of block re-encryption and re-signing to writers and does not incur additional overhead. Furthermore, it should be emphasized that lazy revocation in current researches [10,7,38] mainly refers to the revocation of the read permission. Actually, revocation of different privileges implies different semantics. Specifically, revoking a user from a group can be differentiated into three situations. Let $B = \{m_0, m_1, m_2, \ldots, m_n\}$ denote a block group and m_0 refers to the admin block. $G = \{u_1, u_2, \ldots, u_m\}$ denote the user set. According to our design, we further divide the user set into a reader set G_R and a writer set G_W, then we have $G = G_R \supseteq G_W$, and $u_i \in G_W$ implies $u_i \in G_R$.

- Revoke the read privilege from a reader $u_i \in G_R \cap u_i \notin G_W$, namely, after the revocation, we have $u_i \notin G_R \cap u_i \notin G_W$, which implies re-encrypting all data blocks in the group with a new version of encryption key.
- Revoke the write privilege from a writer $u_i \in G_W \cap u_i \in G_R$ but keep its read privilege, namely, after the revocation, we have $u_i \notin G_W \cap u_i \in G_R$, which implies re-generating signatures of all data blocks in the group with a new version of signing key.
- Revoke both the read and write privileges from a writer $u_i \in G_W \cap u_i \in G_R$, namely, after the revocation, we have $u_i \notin G_W \cap u_i \notin G_R$, which implies re-encrypting all data blocks in the group and re-generating their signatures with a new version of encryption key and signing key respectively.

It should be emphasized that lazy revocation usually means revoking the read privilege. To revoke the write privilege, the signing key should be immediately rotated forward to a new version and used to generate new signatures for all data blocks in the group. Otherwise, a revoked writer may maliciously tamper the data block and meanwhile still generates a valid signature for the block. After revocation, the new version of encryption key or signing key should be broadcast encrypted to the new user set with the revoked user excluded.

In implementation, we adopt the Boneh–Gentry–Waters broadcast encryption scheme [12] that supports incremental sharing. Adding or revoking a user from the current user set only involves multiplying or dividing a group element from the broadcast message, which is especially efficient for cryptographic storage. In the section of performance evaluation, we can see that a broadcast decryption is a CPU-intensive operation, which takes about 15 ms around, and the cost will decrease given a more efficient implementation.

3.4. Access protocols

When a user sends a request to the cloud for a data block m_i, the cloud replies with the encrypted block (with its metadata and signature) and the corresponding admin block m_0. Meanwhile, the cloud also sends the signed VAT root $Sig_{K_T}(h(r))$ of the block group that the retrieved block belongs to, and the auxiliary authentication information Ω that refers to the sibling nodes on the path from the requested block to the tree root.

On receiving the message from the cloud, the user firstly verifies signatures of m_i and m_0 for integrity check. If succeeds, the user re-computes the VAT root using the value $fid\|i\|v_i$ of m_i and the auxiliary authentication information Ω, signs the root with the latest root signing key K_T he holds, and compares it to $Sig_{K_T}(h(r))$ from the cloud to perform the freshness check. If they are equal, the user can be sure that the retrieved data block does reflect its latest update.

When the above two steps succeed, the user decrypts the broadcast message $E_B(K_E, G_R)$ in the admin block to recover the read key K_E. If the version numbers of K_E in m_i and m_0 do not match, then K_E should be rotated forward to get the correct version and use it to decrypt the encrypted data block $E(m_i, K_E)$ to get its plaintext content. For block update, the user also needs to decrypt the broadcast message $E_B(K_S, G_W)$ to recover the signing key K_S, so that he can generate the new signature for the update block.

When the user finishes the update of block m_i, he also updates the block's metadata by incrementing the block version number by one, re-encrypts the modified block m_i' using the latest K_E and generates the block's new signature using the latest K_S. Then the user re-computes the new VAT root r', rotates the root signing key K_T to a new version K_T' and uses it to sign r' to get $Sig_{K_T'}(h(r'))$. Finally, the user sends the updated block (with its updated metadata and signature) and the new signed VAT root to the cloud for storage.

Upon receiving the update message from the client, the cloud verifies the integrity of the modified block, re-constructs the VAT root using the new version number of the modified block, and verifies the updated VAT root signature $Sig_{K_T'}(h(r'))$ from the client with the re-computed one. If succeeds, the cloud stores the updated block (with its updated metadata and signature) and the new VAT root, and sends an acknowledgment $Sig_{K^R_{cloud}}(m_i')$ to the client to inform the client that the update is successfully finished.

The instantaneous freshness check in our design relies on the latest root signing key K_T maintained by the client. According to [8], only the owner who holds the secret master key can rotate the root signing key to a new version. For each update of a data block, the private key of the rotated root signing key is maintained by the owner only, the CSP just has its public key for verification of the updated root. In this sense, even the cloud launch a replay attack by sending a stale version of block and VAT root (corresponding to an older version of the root signing key) to the user, it is impossible

Table 2
Involved cryptographic primitives in user operations.

Involved primitives	Block create	Block read	Block write	Revoke a reader	Revoke a writer
Signature verification		√	√		
Broadcast decryption		√	√		
Block decryption		√	√		
Block encryption	√		√		
Signature generation	√		√	√ (admin block)	√ (admin block)
Tag generation	√		√		
Broadcast encryption	√			√ (admin block)	√ (admin block)
AES and RSA key generation	√				
Rotate encryption key				√	√
Rotate root signing key					√

for the stale data to pass the root signature verification at the client side. Table 2 depicts the involved primitives of block operations and user revocations, where signature generation and broadcast encryption involved in user revocations occur in the admin block of a block group.

3.5. Dynamic integrity auditing for cold data

In a PoS scheme, data is fragmented into multiple blocks of the same size, then for each block a tag is computed, e.g., $\sigma_i = (H(i) \cdot u^{m_i})^x$. Because a tag is signed with owner's private key x, it could not be forged without the private key. An auditor needs to send a challenge request $chal = \{(i, v_i)\}_{s_1 \le i \le s_c}$ to the CSP to initiate an auditing, where i refers to the index of a requested block, $v_i \in \mathbb{Z}_p$ denotes a random coefficient corresponding to i. On receiving $chal$, the CSP computes the integrity proof (μ, σ), where μ is computed from requested blocks and σ is computed from their tags. Due to the aggregative property of homomorphic verifiable tags [41], the one-to-one correspondence between a block and its tag is kept in μ and σ. The auditor verifies the proof by verifying whether μ and σ satisfy some mathematical relation (e.g., bilinear pairing in [4]).

Generally, embedding $H(i)$ in tag computation is to authenticate blocks lest a misbehaved server uses other non-requested blocks to compute the integrity proof, such attacks do exist if we simplify the tag computation as $\sigma_i = (u^{m_i})^x$. However, due to the usage of block index, whenever a block is inserted or deleted, indices of all following blocks change. As a result, tags of these blocks have to be re-computed, which is unacceptable for its high computation cost. This problem makes data dynamics support in PoS schemes a difficult task. To our knowledge, researches in [29–33,42] address this problem, but they all have their limitations, as explained in the background section.

In our construction, we combine data dynamics support with our data organization to achieve more efficient and scalable auditing. Specifically, a tag is computed as $\sigma_i = (H(gid_i) \cdot u^{m_i})^\alpha$, where gid_i is a globally unique id for a block, it is embedded in a block's metadata and its integrity is protected by the attached signature. Because tags are generated and updated by the data owner, it is the owner's responsibility to guarantee the uniqueness of each gid. In implementation, a simple method is to maintain a monotonous increasing counter (increment by one for each tag generation or update) to generate a unique gid for each tag computation. The algorithms of our dynamic PoS scheme are described as follows, which are based on the public scheme in [4].

Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be multiplicative cyclic groups of prime order p, g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear map, and $H(\cdot) : \{0, 1\}^* \to \mathbb{G}_1$ be a secure public map-to-point hash function, which maps a string $\{0, 1\}^*$ uniformly into an element of $\mathbb{G}_1$.

KeyGen(1^k). According to the security parameter k, the data owner randomly chooses $\alpha \leftarrow \mathbb{Z}_p$ and $u \leftarrow \mathbb{G}_1$, computes $v \leftarrow g^\alpha$ and $w \leftarrow u^\alpha$. The secret key is $sk = \alpha$ and the public key is $pk = (v, w, g, u)$.

TagGen(sk, F). Given a data file $F = \{m_i\}_{1 \le i \le n}$. For each block m_i, the owner computes its tag as $\sigma_i = (H(gid_i) \cdot u^{m_i})^\alpha$. Denote the tag set by $\Phi = \{\sigma_i\}_{1 \le i \le n}$.

The auditor randomly picks a c-element subset $I = \{s_1, s_2, \ldots, s_c\} \subseteq \{1, 2, \ldots, n\}$, where elements in I refer to block indices of requested blocks. The auditor sends a challenge request $chal = \{(i, v_i)\}_{i \in I}$ to the server where each $v_i \in \mathbb{Z}_p$ denotes a random coefficient corresponding to i.

ProofGen$(chal, F, \Phi)$. On receiving the auditing challenge $\{(i, v_i)\}_{i \in I}$, the server computes its integrity proof as $\mu = \sum_{i \in I} v_i \cdot m_i$ and $\sigma = \prod_{i \in I} \sigma_i^{v_i}$, where μ denotes the linear combination of requested blocks and σ denotes the aggregated signature of corresponding tags. Finally, the server sends the proof $\pi = (\mu, \sigma)$ to the auditor for verification.

ProofVerify$(pk, chal, \pi)$. On receiving the integrity proof $\pi = (\mu, \sigma)$, the verifier firstly searches the gid_i for each requested block m_i in the challenge set. Then the verifier could check the correctness of the integrity proof by verifying whether the following equation holds or not:

$$e(\sigma, g) \stackrel{?}{=} e\left(\left(\prod_{i \in I} H(gid_i)^{v_i}\right) \cdot u^\mu, v\right). \quad (1)$$

If the equation holds, output *TRUE*, otherwise *FALSE*.

Data dynamics. Our design principle of data dynamics support is to let a block's index indicate the logical position in its file, and to let a block's *gid* be used for its tag computation. Thus, indices of all blocks in a file always appear as a consecutive sequence $1, 2, \ldots, n$, while the sequence of *gid* of all blocks may not necessarily be consecutive. The coldness table keeps a mapping between *gids* and block indices. During auditing, it is necessary for the auditor to obtain corresponding *gids* for requested blocks to verify the proof from the cloud.

- **Modification**. When a user changes a block m_k into m'_k, the user allocates an unused gid' for m'_k and computes its new tag as $\sigma'_k = (H(gid') \cdot u^{m'_k})^\alpha$. Then the user sends an update request $\{O(M), k, m'_k, \sigma'_k, Q\}$ to the server, where $O(M)$ refers to modification and k specifies the operation position, $Q = \{(i, v_i)\}_{i \in I \cap k \in I}$ is a small challenge set with the modified block m'_k included. Upon receiving the request, the cloud verifies the correctness of m'_k and σ'_k by verifying $e(\sigma'_k, g) \stackrel{?}{=} e(H(gid') \cdot u^{m'_k}, v)$. If succeeds, the cloud replaces m_k and σ_k with m'_k and σ'_k, computes the integrity proof π according to Q, and sends π to the user for verification.
- **Insertion**. When a user inserts a new block m_k at position k, he firstly allocates an unused *gid* for the new block and computes its tag $\sigma_k = (H(gid) \cdot u^{m_k})^\alpha$. Then the user sends an update request $\{O(I), k, m_k, \sigma_k, Q\}$ to the server, where $O(I)$ refers to insertion and k specifies the insertion position. Along with the

update request is a small challenge set $Q = \{(i, v_i)\}_{i \in I \cap k \in I}$ with the new block m_k included. Upon receiving the update request, the server verifies the correctness of m_k and σ_k by checking $e(\sigma_k, g) \stackrel{?}{=} e(H(gid) \cdot u^{m_k}, v)$. If succeeds, the CSP inserts m_k and σ_k into F and Φ, then computes the integrity proof π according to Q, and sends π to the user for verification.

- **Deletion**. When a user deletes a block, the user sends an update request $\{O(D), k, Q\}$ to the server, where $O(D)$ refers to deletion and k specifies the deletion position, $Q = \{(i, v_i)\}_{i \in I \cap k \in I}$ is a small challenge set with the deleted block m_k included (since the kth position is now occupied by the subsequent block, we authenticate the deletion operation by authenticating the new block at the kth position). Upon receiving the update request, the server deletes m_k and σ_k from F and Φ, then computes the integrity proof π according to Q, and sends π to the user for verification.

These provable data dynamics (block modification, insertion and deletion) protocols all have the default integrity verification protocol included in order to authenticate the update operation at the server side. Therefore, we can choose the number c of challenged blocks to be a small number (e,g., $c = 10$) to accelerate the speed of proof computation and verification.

Update of the coldness table. On receiving the integrity proof π from the cloud, the user verifies its validity according to *ProofVerify*. If succeeds, the user updates the coldness table according to the operation type.

- **Modification**. The user updates the coldness table by modifying the *gid* field of the kth record into gid'.
- **Insertion**. The user updates the coldness table by inserting the new record containing gid' at the kth position in the table and incrementing the index field of all following records by one.
- **Deletion** The user updates the coldness table by deleting the kth record and subtracting the index field of all following records by one.

For block insertion and deletion, the index field of all subsequent records (after the operation position) should be updated. And the update should be limited in the range of the file that the operating block belongs to, namely, these records should have the same *fid* as the block to be inserted or deleted.

Discussion Note that, while privacy protection is beyond the scope of this paper, our scheme can directly adopt the random mask technique proposed in [43,44] to prevent information leakage toward the third-party auditor. Otherwise, if the TPA can collect enough proofs on the same set of challenged blocks, then he can derive the content of these blocks by solving a system of linear equations. Further, our scheme can adopt the ring signatures in [45,46] to protect the identity-privacy of signers for data shared among a group of users.

3.6. Tag update policy

Recall in our block organization, we store the gid and version number of each block into the coldness table. For large-scale data, the coldness table can be very large. For ease of management and update, we do not store gid and version of all blocks in a single big table. Instead, we construct a small table for each block group.

Since the *gid* values of challenged blocks in an auditing are needed by the auditor, its integrity and freshness affect the correctness of the auditing result. For simplification, we extract the *gid*, *fid* and index fields from the coldness table to build a gid authentication tree (GAT) for each block group, then recursively compute the Merkle hash root $MHT\{h(fid_i \| i \| gid_i)\}_{1 \le i \le n}$ and sign it with another root signing key K_G (different from the VAT root signing key K_T), which is to be rotated to a new version upon each update of the gid authentication tree. The freshness check process of the GAT root is similar to that of a VAT root.

Now we consider the situation of tag update, where block modification, insertion and deletion are included. For block insertion, the signature and tag of the new block should be computed and attached with the encrypted block. For block deletion, the block and its tag are deleted from the group. In both cases, the corresponding leaf node in the authentication tree (VAT and GAT) should also be inserted or deleted. Consequently, the tree roots need to be re-computed, and the coldness table should also be updated, as described in the previous section. For block modification, we divide the tag update policy into two situations.

Lazy update. Intuitively, when the block is updated, its tag should also be updated. However, the tag computation involves two exponentiation, whose cost is expensive. On the other hand, for a frequently accessed and updated block, it is not so urgent to re-compute its tag at each update. As an alternative, we can delay the tag re-computation until the block becomes cold, i.e., when its coldness value exceeds a pre-defined time threshold. We call such a tag update policy as lazy update. Under such situation, we have to build two Merkle hash tree (VAT and GAT) for version authentication and gid authentication respectively, because the update of a VAT root is instantly performed at each update of the block group, while the update of a GAT root can be delayed to save computation overhead, which may lead to the inconsistency between a block and its tag.

In implementation, our system let a process monitor the coldness values of recently updated blocks, when their coldness values exceed a certain threshold (e.g. three months), the monitor process will notify the owner to update tags of these blocks. This is to ensure the correctness of later PoS auditing, since auditing is mainly performed to check the integrity of cold data.

Active update. In contrast to lazy update, active update policy requires the tag of an updated block to be immediately updated after block modification. Under such an update policy, the version authentication tree and the gid authentication tree can be merged into one single authentication tree, where the leaf nodes store the values of $h(fid_i \| i \| v_i \; \| \; gid_i)_{1 \le i \le n}$ instead. Thus, for each update, the authentication tree root should be re-computed and signed, and the coldness table should also be updated correspondingly. As a consequence of active update, the total cryptographic cost of block write will increase a significant portion and lead to a low throughput, as we will see it in the section of performance evaluation.

Note that the sampling strategy of PoS auditing scheme requires the challenged blocks to be randomly chosen. At worst, the challenged blocks are chosen from many big block groups (e.g., a group contains thousands of data blocks) with each group a block being chosen. In this case, the root verification of gid values will be costly, since for each challenged block, the gid values of the whole group (that the challenged block belongs to) have to be authenticated.

4. Security analysis

In this section, we analyze the security fulfillment of our design goals, including confidentiality, integrity and freshness. And prove the security of our dynamic PoS scheme.

Confidentiality. In our block organization, a data block is encrypted using its read key, which is stored in the broadcast encryption message. Only authorized user can decrypt it to recover the read key for block decryption. Without the decryption key, even the cloud cannot learn the content of the block. The confidentiality guarantee is dependent on the security of the AES algorithm and the broadcast encryption scheme. On the other hand, the cloud can deduce some information by recording the

Table 3
Constant cryptography primitives.

Primitive	Cost (ms)
Encryption key rotation (128 bit AES)	1.8
RSA key pair rotation (1024 bit)	112
RSA key pair generation (1024 bit)	31
Broadcast encryption (64 users)	22
Broadcast decryption (64 users)	15

access patterns and access frequency of some data. How to hide these information is beyond our work.

Integrity. Each data block is attached with a signature generated using the write key. The cloud verifies a block's integrity for each write and the user verifies a block's signature for each read. Due to the security of the employed signature scheme, an unauthorized user cannot forge a valid signature of a block without the corresponding signing key.

Freshness. Each time a user reads a block from the cloud, he would authenticate the VAT root using the latest root signing key. Due to the security of the RSA based key regression scheme [8], it is impossible for an adversary to guess the new version key without the secret master key. Therefore, if the cloud sends a stale data block and its corresponding VAT root (signed with an old root signing key) to the user, it is impossible for the stale data to pass the root authentication check using the latest root signing key kept by the client.

We now prove the security of our dynamic PoS auditing scheme by proving the following theorem on the basis of the Computational Diffie–Hellman assumption.

Definition 1 (*CDH Assumption*)**.** The Computational Diffie–Hellman assumption is that, given $g, g^a, h \in \mathbb{G}$ for unknown $a \in \mathbb{Z}_p$, it is computationally infeasible to output g^h.

Theorem 1. *If the signature scheme used for tags is existentially unforgeable and the Computational Diffie–Hellman problem is hard in bilinear groups, then no adversary against the soundness of our auditing scheme could cause the verifier to accept with non-negligible probability, except by responding with correct proof.*

Proof. Firstly, the freshness check of the GAT root guarantees gids of all challenged blocks (in a PoS auditing) reflect their latest update.

Secondly, since each gid is a globally unique identifier, the mapping from block indices $i \in \{1, 2, \ldots, n\}$ to its gids $gid_i \in \mathbb{N}$ can be regarded as an injective function. Thus, the hash query of gid in our scheme is similar to the hash query of block index in [4]. We now prove the theorem in the random oracle model by using the same series of games defined in [4].

Suppose $Q = (i, v_i)_{i \in I}$ is the query that causes the adversary wins the game. Let $\pi = \{\mu, \sigma\}$ be the expected response from an honest prover, which satisfies $e(\sigma, g) = e\big(\big(\prod_{i\in I} H(gid_i)^{v_i}\big) \cdot u^{\mu}, v\big)$. Assume the adversary's response is $\pi' = \{\mu', \sigma'\}$, which satisfies $e(\sigma', g) = e\big(\big(\prod_{i\in I} H(gid_i)^{v_i}\big) \cdot u^{\mu'}, v\big)$. Obviously, we have $\mu \neq \mu'$, otherwise we will have $\sigma = \sigma'$, which contradicts our assumption.

Define $\Delta\mu = \mu' - \mu$, now we can construct a simulator to solve the Computational Diffie–Hellman problem. Given $(g, g^{\alpha}, h) \in \mathbb{G}$, the simulator is to output h^{α}. The simulator sets $v = g^{\alpha}$, randomly picks $r_i, \beta, \gamma \in \mathbb{Z}_p^*$ and sets $u = g^{\beta} \cdot h^{\gamma}$. Then the simulator answers hash queries as $H(gid_i) = g^{r_i}/(g^{\beta \cdot m_i} \cdot h^{\gamma \cdot m_i})$, and answers signing queries as $\sigma_i = (g^{r_i})^{\alpha}$. Finally, the adversary outputs $\pi' = (\mu', \sigma')$. From above two equations, we have

$$e(\sigma'/\sigma, g) = e(u^{\Delta\mu}, v) = e((g^{\beta} \cdot h^{\gamma})^{\Delta\mu}, v).$$

From this equation, we can obtain

$$e(\sigma' \cdot \sigma^{-1} \cdot (g^{\alpha})^{-\beta \cdot \Delta\mu}, g) = e((h^{\gamma})^{\Delta\mu}, g^{\alpha}) = e((h^{\alpha})^{\gamma\Delta\mu}, g).$$

Further, we get $h^{\alpha} = (\sigma' \cdot \sigma^{-1} \cdot (g^{\alpha})^{-\beta\Delta\mu})^{1/(\gamma\Delta\mu)}$. Since γ is randomly chosen from $\mathbb{Z}_p^*$ by the challenger, and is hidden from the adversary, the probability of $\gamma\Delta\mu = 0 \mod p$ will be $1/p$, which is negligible. □

5. Performance evaluation

In this section, we measure the performance of our construction. We developed a prototype on a CentOS 6.3 system using C/C++ language. The system is equipped with a 4-Core Intel Xeon E5620 processor running at 2.4 GHz, 4 GB RAM and a 7200 RPM 1 TB drive. Our implementation uses the Pairing-Based Cryptographic (PBC) library at version 0.5.11, OpenSSL library at version 1.0.0. We choose AES-128 for block encryption and decryption, SHA-1 for hash, RSA-1024 for signing and verification. For broadcast encryption, we use PBC_bce library 0.0.1 which is developed on the basis of Boneh–Gentry–Waters broadcast encryption scheme. All experiment results are on the average of 12 trials, with the top and bottom results excluded.

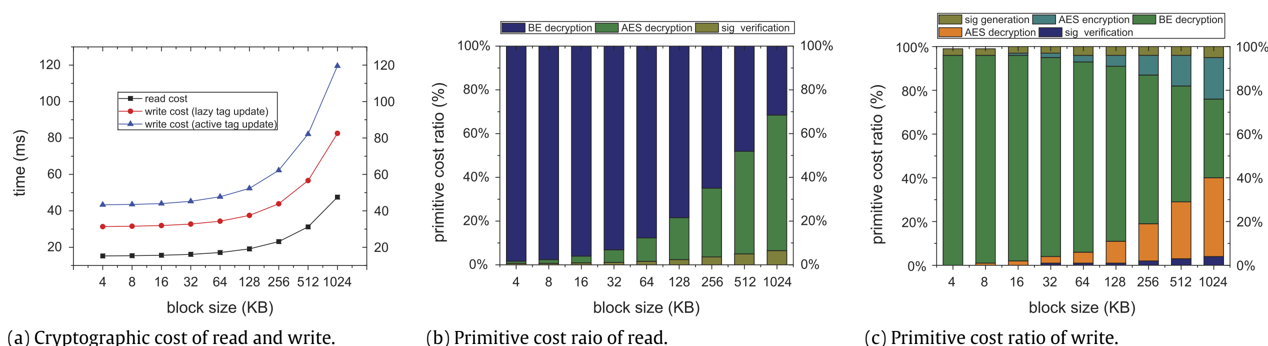
5.1. Cryptographic cost

In our design, reading a block involves verifying the block's signature, performing a broadcast decryption to get the encryption key, and decrypting the encrypted data block. Additionally, writing a block involves re-encrypting the modified block, performing a broadcast decryption to get the signing key and generating a new signature for the updated block.

Among these primitives, some incur a constant overhead that is independent of the block size, such as key rotation, key generation and broadcast encryption/decryption. From Table 3, we can see that the cost of rotating a RSA key is the most expensive and costs much more overhead than the symmetric key rotation, which explains why revoking a writer incurs much more overhead than revoking a reader. Other primitives' overheads are dependent on the block size, e.g., block encryption/decryption and signature generation/verification.

Fig. 5(a) depicts the total cryptographic cost of reading and writing a data block, we can see that the growth of overhead increases slowly when block size is less than 64 kB, this is because the broadcast decryption overhead takes more than 95% in the total cost. But when block size exceeds 128 kB, the three curves begin to increase sharply, which is due to the growth of AES encryption/decryption overhead. For example, when the block size is 1024 kB, for block read, broadcast encryption cost takes 32% and AES decryption cost takes 62% in the total cost; for block write, broadcast encryption cost takes 36%, while cost of AES encryption and decryption takes about 55%. Figs. 5(b) and 5(c) show the cost ratio among involved primitives of block read and write. For both operations, broadcast decryption cost dominates the total cost when block size is less than 64 kB, and its influence decreases fast when block size exceeds 128 kB. Meanwhile, cost of AES encryption/decryption becomes a dominant factor of the total cost.

The gap between read and write is due to the broadcast decryption of the signing key. And the gap between two write curves is due to the cost of computing the tag right after block update, according to lazy update policy and active update policy respectively. Recall that a tag is computed as $\sigma_i = (H(gid_i) \cdot u^{m_i})^{\alpha}$, whose cost mainly includes reading the block, exponentiation and signing with the private key α. Table 4 shows the cost of tag computation at different block size.



(a) Cryptographic cost of read and write.
(b) Primitive cost raio of read.
(c) Primitive cost ratio of write.

Fig. 5. Cryptographic cost and primitive ratios of block read and write.

Table 4
Cost of tag computation.

Block size (kB)	≤32	64	128	256	512	1024
Time cost (ms)	12.2	13.4	14.8	18.4	25.5	37

5.2. Throughput

To measure the throughput of read and write, we choose a 100-MB data file, fragment it into multiple blocks according to different size (varying from 4 to 1024 kB). We organize these block files into multiple groups. For each block size, we read and write all the content (100-MB) from these block files. The results are generated by dividing the file size by the effective duration, with the round-trip time excluded.

For each block group, an encryption key and a RSA key pair are generated, and these keys are broadcast encrypted and stored in the admin block of the group. For read operation, the user needs to perform a broadcast decryption to recover the read key. While for write operation, the user needs to perform two broadcast decryption to recover both the read key and the write key. As the overhead of broadcast decryption is constant, its influence on read and write throughput is dependent on how many blocks of the same group will be accessed. At worst, the user reads or writes only one block from each group, which leads to the minimal throughput. Because in this case, the number of broadcast decryption is linear to the number of retrieved blocks, since different groups use different read keys and write keys.

To elaborate the possible factors influencing the throughput results, we firstly run the experiment by organizing each block into a unique group to get the worst-case throughput. Then we choose some fragmentation sizes (4, 16, 64, 256 kB) and vary the number of blocks in a group to test the real running throughput.

Fig. 6 depicts the worst-case throughput, we observe that read and write throughput are close at small block size, but the gap increases when block size exceeds 128 kB. This is because the overhead of AES encryption/decryption takes a greater portion in total cost at big block size (e.g., 29% and 46% for read and write at 1024 kB, respectively). Generally, throughput increases as block size increases. This is because for the same file, the growth rate of cryptographic cost in read and write is less than the reduction rate of block numbers when block size increases, thus the total cost for read and write decreases.

Fig. 7 illustrates the throughput of different fragmentation for the same 100-MB file when varying the number of blocks in a group: 10, 50, 100 and 200. Generally speaking, one hand, both the read and write (lazy update) curves increase when the block numbers in a group increase. This is because the growth of blocks in a group leads to the broadcast decryption overhead be shared by more blocks (blocks in a group are encrypted with the same

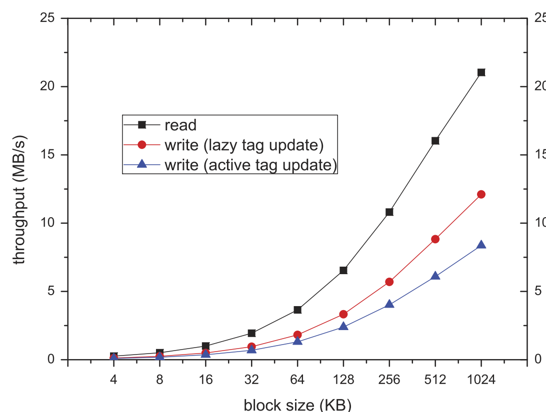


Fig. 6. Throughput of reading and writing a 100-MB file.

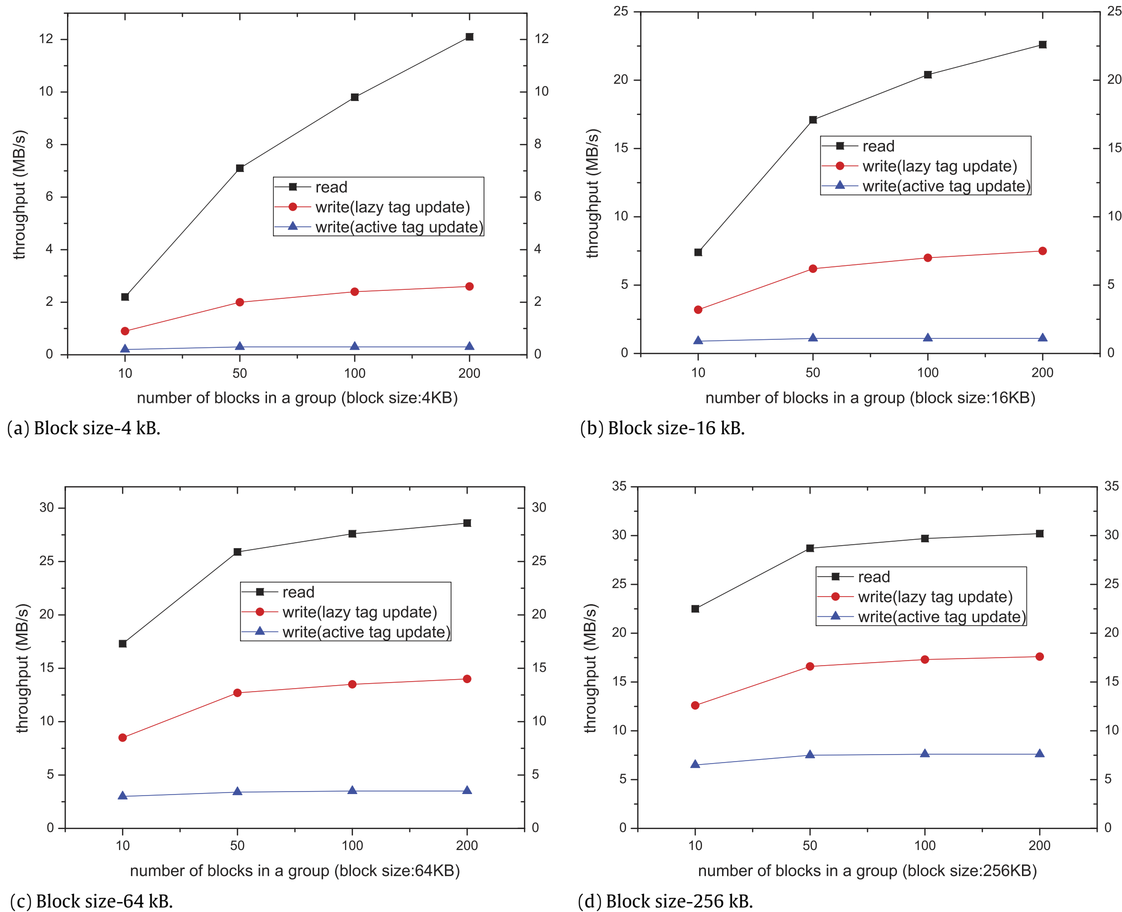
read key), so the cryptographic cost of accessing a single block decreases. The write (active update) curve remains stable and shows a low throughput, which can be explained by the immediate tag re-computation after block update, whose cost is dominant among all involved primitives (e.g., more than 15 ms when block exceeds 128 kB). On the other hand, when the number of blocks in a group is fixed, the throughput increases when we choose bigger fragmentation, e.g., for groups containing 100 blocks, 4 kB fragmentation leads to 9.8 MB/s for read and 2.4 MB/s for write, while 256-kB fragmentation can achieve 29.7 MB/s for read and 17.3 MB/s for write. Additionally, there is a significant gap between read and write curves and the gap increases when the number of blocks in a group grows. The reason is that for a single block access, the cost of broadcast decryption decreases since it is now shared by more blocks. As a result, block re-encryption and signature generation in a write operation take a larger portion in the total overhead.

Combined with the cryptographic cost analysis, we can see that two factors having the greatest influence on the throughput: the block fragmentation size and the number of blocks in a group. For example, when choosing 256-kB fragmentation and organizing 200 blocks in a group, the throughput can achieve 30.2 MB/s for read and 17.6 MB/s for write.

5.3. Storage overhead

From our block structure, a block's metadata consists of a *gid*, a version number, version of K_E and K_S, the block signature and the PoS tag. The admin block of a group consists of a version number, the broadcast message of K_E and K_S, K_V in plain-text, newest version number of K_E and K_S, and an attached a signature.

In implementation, we allocate 12 bytes for *gid* and 4 bytes for version number. Both the signature and tag are 128 bytes. In this



(a) Block size-4 kB.

(b) Block size-16 kB.

(c) Block size-64 kB.

(d) Block size-256 kB.

Fig. 7. Real throughput when choosing different number of blocks in a group.

way, we can get the additional storage (metadata, signature and tag) for a block is about 280 bytes. Moreover, a block takes a record in the coldness table, which amounts to about 26 bytes for each record (4 bytes for *fid* and 8 bytes for index). Thus, we can get the additional storage for a block in total is about 306 bytes, which takes a percentage of less than 1.8% when the block size exceeds 16 kB.

The Boneh–Gentry–Waters broadcast encryption scheme provides a fixed size cipher-text consisting of only two group elements, and the public key information of a broadcast instance and the description of a user set are linear to the number of users. Thus, the broadcast result consists of $x \times n + (2n + 1) \times 16 + 32$ bytes, where x is the bytes a user ID takes and n is the number of users (in PBC library, each group element takes 16 bytes). Assuming 4-byte user IDs and the number of authorized users in a user set being 100, the broadcast result is about 3648 bytes. For small groups with less than 100 users, e.g., a group of 20 users, the broadcast result is much less (about 768 bytes).

Generally, if we choose big block size for fragmentation and organize more blocks in a group, then the ratio of additional storage will take a less portion in the total storage cost.

6. Conclusion

The aim of this paper is to provide a secure storage system for cloud data, where confidentiality, full integrity and instantaneous freshness check are achieved. This paper presents the design, implementation and evaluation of such a secure storage system. We devise secure and flexible data structure for encrypted storage on untrusted cloud servers, and provide corresponding access control, key management, permission revocation policies to enhance security and scalability. Moreover, we seamlessly integrate traditional signature based integrity protection with PoS auditing mechanism to provide full integrity for outsourced data, including frequently retrieved data and rarely retrieved data. We believe our work complements current research on building a secure cloud storage system. Our experimental evaluation shows that the cryptographic cost incurred by security primitives and throughput of read and write operations are reasonable and acceptable.

Acknowledgments

Firstly, the authors would like to thank the anonymous referees for their reviews and suggestions to improve this paper. Secondly, the work is supported in part by the National Basic Research Program (973 Program) of China under Grant No. 2011CB302305, and the National Natural Science Foundation of China under Grant No. 61232004. This work is also sponsored in part by the National High Technology Research and Development Program (863 Program) of China under Grant No. 2013AA013203.

References

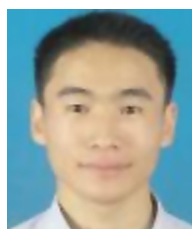
[1] M. Blaze, A cryptographic file system for unix, in: Proc. 1st ACM conf. Computer and Comm. Security, 1993, pp. 9–16.
[2] A. Juels, B.S. Kaliski Jr., Pors: Proofs of retrievability for large files, in: Proc. 14th ACM Conf. Computer and Comm. Security, CCS'07, 2007, pp. 584–597.
[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, Provable data possession at untrusted stores, in: Proc. 14th ACM Conf. Computer and Comm. Security, CCS'07, 2007, pp. 598–609.

[4] H. Shacham, B. Waters, Compact proofs of retrievability, in: Proc. 14th Int'l Conf. Theory and Application of Cryptology and Information Security: Advances in Cryptology, ASIACRYPT'08, 2008, pp. 90–107.
[5] G. Ateniese, S. Kamara, J. Katz, Proofs of storage from homomorphic identification protocols, in: Proc. 15th Int'l Conf. Theory and Application of Cryptology and Information Security: Advances in Cryptology, ASIACRYPT'09, 2009, pp. 319–333.
[6] E.-J. Goh, H. Shacham, N. Modadugu, D. Boneh, Sirius: Securing remote untrusted storage., in: Proc. 7th Network and Distributed System Security Symp., NDSS'03, 2003, pp. 131–145.
[7] R.A. Popa, J.R. Lorch, D. Molnar, H.J. Wang, L. Zhuang, Enabling security in cloud storage slas with cloudproof, in: USENIX Ann. Technical Conf., ATC'11, vol. 242, 2011.
[8] K. Fu, S. Kamara, T. Kohno, Key regression: Enabling efficient key distribution for secure distributed storage, 2006, pp. 110–149.
[9] R.C. Merkle, Protocols for public key cryptosystems, in: Proc. IEEE Symp. Security and Privacy, 1980, pp. 122–133.
[10] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, Plutus: Scalable secure file sharing on untrusted storage, in: Proc. 2nd USENIX Conf. File and Storage Technologies, FAST'03, 2003, pp. 29–42.
[11] J. Li, M.N. Krohn, D. Mazières, D. Shasha, Secure untrusted data repository (sundr), in: Proc. 6th USENIX Symp. Operating Systems Design and Implementation, OSDI'04, 2004, pp. 121–136.
[12] D. Boneh, C. Gentry, B. Waters, Collusion resistant broadcast encryption with short ciphertexts and private keys, in: Advances in Cryptology, CRYPTO'05, 2005, pp. 258–275.
[13] P. Stanton, Securing data in storage: A review of current research, ArXiv Preprint cs/0409034.
[14] V. Kher, Y. Kim, Securing distributed storage: challenges, techniques, and systems, in: Proc. 2005 ACM workshop on Storage Security and Survivability, 2005, pp. 9–25.
[15] K.E. Fu, Group sharing and random access in cryptographic storage file systems (Ph.D. thesis), Massachusetts Institute of Technology, 1999.
[16] E.L. Miller, D.D. Long, W.E. Freeman, B. Reed, Strong security for network-attached storage., in: Proc. 1st USENIX Conf. File and Storage Technologies, FAST'02, 2002, pp. 1–13.
[17] D. Mazieres, D. Shasha, Building secure file systems out of byzantine storage, in: Proc. 21st Ann. Symp. Principles of Distributed Computing, PODC'02, 2002, pp. 108–117.
[18] S. Kamara, C. Papamanthou, T. Roeder, Cs2: A searchable cryptographic cloud storage system, Microsoft Research, TechReport MSR-TR-2011-58, 2011.
[19] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, M. Walfish, Depot: Cloud storage with minimal trust, ACM Trans. Comput. Syst. 29 (4) (2011) 12–28.
[20] E. Stefanov, M. van Dijk, A. Juels, A. Oprea, Iris: A scalable cloud file system with efficient integrity checks, in: Proc. 28th Ann. Computer Security Applications Conf., 2012, pp. 229–238.
[21] A. Bessani, M. Correia, B. Quaresma, F. André, P. Sousa, Depsky: dependable and secure storage in a cloud-of-clouds, ACM Trans. Storage 9 (4) (2013) 12–27.
[22] Y. Deswarte, J.-J. Quisquater, A. Saïdane, Remote integrity checking, in: Proc. 5th Working Conf. Integrity and Int'l Control in Information Systems, 2004, pp. 1–11.
[23] A. Oprea, M.K. Reiter, K. Yang, Space-efficient block storage integrity, in: Proc. 9th Network and Distributed System Security Symp., NDSS'05, 2005.
[24] D.L. Gazzoni Filho, P.S.L.M. Barreto, Demonstrating data possession and uncheatable data transfer, IACR Cryptology ePrint Archive, Report 2006/150, 2006.
[25] K.D. Bowers, A. Juels, A. Oprea, Proofs of retrievability: Theory and implementation, in: Proc. ACM Cloud Computing Security Workshop, CCSW'09, 2009, pp. 43–54.
[26] E.-C. Chang, J. Xu, Remote integrity check with dishonest storage server, in: Proc. 13th European Conf. Research in Computer Security, ESORICS'08, 2008, pp. 223–237.
[27] M.A. Shah, R. Swaminathan, M. Baker, Privacy-preserving audit and extraction of digital contents, IACR Cryptology ePrint Archive, Report 2008/186, 2008.
[28] F. Sebé, J. Domingo-Ferrer, A. Martinez-Balleste, Y. Deswarte, J.-J. Quisquater, Efficient remote data possession checking in critical information infrastructures, IEEE Trans. Knowl. Data Eng. 20 (8) (2008) 1034–1038.
[29] G. Ateniese, R. Di Pietro, L.V. Mancini, G. Tsudik, Scalable and efficient provable data possession, in: Proc. 4th Intl Conf. Security and Privacy in Comm. Networks, SecureComm'08, 2008, pp. 1–10.
[30] C. Erway, A. Küpçü, C. Papamanthou, R. Tamassia, Dynamic provable data possession, in: Proc. 16th ACM Conf. Computer and Comm. Security, CCS'09, 2009, pp. 213–222.
[31] Q. Wang, C. Wang, J. Li, K. Ren, W. Lou, Enabling public verifiability and data dynamics for storage security in cloud computing, in: Proc. 14th European Conf. Research in Computer Security, ESORICS'08, 2009, pp. 355–370.
[32] Y. Zhu, H. Wang, Z. Hu, G.-J. Ahn, H. Hu, S.S. Yau, Dynamic audit services for integrity verification of outsourced storages in clouds, in: Proc. ACM Symp. Applied Computing, SAC'11, 2011, pp. 1550–1557.
[33] Q. Zheng, S. Xu, Fair and dynamic proofs of retrievability, in: Proc. 1st ACM Conf. Data and Application Security and Privacy, CODASPY'11, 2011, pp. 237–248.
[34] A. Fiat, M. Naor, Broadcast encryption, in: Advances in Cryptology, CRYPTO'93, 1994, pp. 480–491.
[35] J.A. Garay, J. Staddon, A. Wool, Long-lived broadcast encryption, in: Advances in Cryptology, CRYPTO'00, 2000, pp. 333–352.
[36] D. Halevy, A. Shamir, The lsd broadcast encryption scheme, in: Advances in Cryptology, CRYPTO'02, 2002, pp. 47–60.
[37] D. Boneh, B. Lynn, H. Shacham, Short signatures from the weil pairing, in: Advances in Cryptology, ASIACRYPT'01, 2001, pp. 514–532.
[38] A. Kumbhare, Y. Simmhan, V. Prasanna, Cryptonite: A secure and performant data repository on public clouds, in: Proc. IEEE 5th Int'l Conf. Cloud Computing, 2012, pp. 510–517.
[39] M. Backes, C. Cachin, A. Oprea, Lazy revocation in cryptographic file systems, in: Proc. 3rd Int'l Security in Storage Workshop, 2005, pp. 11–27.
[40] S. Zarandioon, D.D. Yao, V. Ganapathy, K2c: Cryptographic cloud storage with lazy revocation and anonymous access, in: Security and Privacy in Communication Networks, 2012, pp. 59–76.
[41] D. Boneh, C. Gentry, B. Lynn, H. Shacham, Aggregate and verifiably encrypted signatures from bilinear maps, in: Advances in Cryptology, EUROCRYPT'03, 2003, pp. 416–432.
[42] C.C. Erway, A. Küpçü, C. Papamanthou, R. Tamassia, Dynamic provable data possession, ACM Trans. Inf. Syst. Secur. 17 (4) (2015) 15–29.
[43] C. Wang, Q. Wang, K. Ren, W. Lou, Privacy-preserving public auditing for data storage security in cloud computing, in: Proc. IEEE INFOCOM, 2010, pp. 1–9.
[44] C. Wang, S.S. Chow, Q. Wang, K. Ren, W. Lou, Privacy-preserving public auditing for secure cloud storage, IEEE Trans. Comput. 62 (2) (2013) 362–375.
[45] B. Wang, B. Li, H. Li, Oruta: Privacy-preserving public auditing for shared data in the cloud, in: Proc. 5th Int'l Conf. Cloud Computing, 2012, pp. 295–302.
[46] B. Wang, B. Li, H. Li, Oruta: Privacy-preserving public auditing for shared data in the cloud, IEEE Trans. Cloud Computing 2 (1) (2014) 43–56.

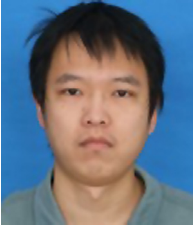
Hao Jin received the B.Sc. degree in Computer Software and Theory in 2005 from Huazhong University of Science and Technology (HUST), Wuhan, China. He is currently a Ph.D. student majoring in Computer System and Architecture at Huazhong University of Science and Technology. His research interests focus on security and privacy issues in cloud computing, storage security and applied cryptography.

Ke Zhou received his Ph.D. degree from the College of Computer Science and Technology, Huazhong University of Science and Technology (HUST) in 2003. Currently, he is a Professor of the College of Computer Science and Technology at HUST. His main research interests include computer architecture, network storage systems, parallel I/O, storage security and theory of network data behavior.

Hong Jiang received the B.Sc. degree in Computer Engineering in 1982 from Huazhong University of Science and Technology, Wuhan, China; the M.A.Sc. degree in Computer Engineering in 1987 from the University of Toronto, Toronto, Canada; and the Ph.D. degree in Computer Science in 1991 from the Texas A&M University, College Station, Texas, USA. Since August 1991 he has been at the University of Nebraska-Lincoln, Lincoln, Nebraska, USA, where he is Willa Cather Professor of Computer Science and Engineering. At UNL, he has graduated 13 Ph.D. students who upon their graduations either landed academic tenure-track positions in Ph.D.-granting US institutions or were employed by major US IT corporations. He has also supervised more than 10 post-doctoral fellows and visiting researchers at UNL. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, performance evaluation. He served as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems, 2008–2013. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACM-TACO, JPDC, ISCA, MICRO, USENIX ATC, FAST, LISA, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by NSF, DOD and the State of Nebraska. Dr. Jiang is a Senior Member of IEEE, and Member of ACM.



Dongliang Lei received his B.Sc. degree in Computer Science and Technology from Hunan Normal University in 2007. Dongliang is currently a Ph.D. Candidate in Computer Science of Huazhong University of Science and Technology. His research interests include high-available storage system and storage security.



Ronglei Wei is a Ph.D. candidate in Wuhan National Laboratory for Optoelectronics at Huazhong University of Science and Technology (HUST). He received the B.Sc. degree from College of Computer Science and Technology from HUST in 2013. His current interests include cloud storage, storage security and privacy.

Chunhua Li is an Associate Professor of Computer Science at the Huazhong University of Science and Technology (HUST). She obtained the Ph.D. degree in Computer Science in 2006 at HUST. Her research interests mainly include cloud storage system and its security, multimedia security, and privacy protection for big data. She has published more than twenty papers in international journals and conferences, including ACM Multimedia, MSST, ICC, SYSTOR, ICME , NAS, ICTAI, etc. She is a member of CCF.