2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Full Integrity and Freshness for Outsourced Storage

Hao Jin[1], Hong Jiang[2], Ke Zhou[1], Ronglei Wei[1], Dongliang Lei[1] and Ping Huang[3]
[1]School of Computer, Wuhan National Lab for Optoelectronics,
Huazhong University of Science and Technology, Wuhan, China
E-mail: khow.hust@gmail.com, {k.zhou, hust_wrl }@hust.edu.cn, dlleig@gmail.com

[2]Dept. of Computer Science and Engineering,
University of Nebraska-Lincoln, NE, USA
E-mail: jiang@cse.unl.edu

[3]Dept. of Electric and Computer Engineering, Virginia
Commonwealth University, VA, USA
E-mail: phuang@vcu.edu

***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 system 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 , proof of storage and fine-grained access control policies together to provide a secure storage service for outsourced data. Experimental evaluation of our prototype shows that the cryptographic cost and throughput is reasonable and acceptable.**

Keywords-full integrity; freshness; secure storage; proof of storage; cloud

I. INTRODUCTION

Outsourcing data to the cloud brings the advantages of high availability, 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 rely on these approaches to guarantee data's security are referred to as cryptographic file systems (CFS) [1].

However, providing only confidentiality and integrity for cloud data protection is not sufficient. This is because of the following limitation: CFS only ensures the integrity of retrieved data, for rarely or never accessed data, they do not work. Since signature or MAC based integrity protection mechanisms require users to download the data 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. And in a cloud environment, there may be a significant portion of data never or rarely accessed (e.g., archive data). In this context, relying on signature verification to guarantee the integrity of large-scale cloud data is a poor and inflexible solution. A better way should be able to differ between frequently retrieved data and rarely retrieved data, and provide an intelligent method to check their integrity.

We address this problem by combining CFS based method with proofs of storage (PoS) technique in a secure storage system design and providing flexible integrity check mechanisms for data with different access frequency. Proofs of storage [17,20,21] is a secure auditing mechanism to check the integrity of remotely stored data without the need of retrieving original data by users, which complements signature based mechanisms very well, especially for those rarely or never accessed data outsourced to the cloud. With such auditing mechanisms, users can periodically interact with the CSP through auditing protocols to check the correctness of their outsourced data.

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 address this problem in their storage system designs [6, 27]. But in cloud storage, owners and the cloud belong to different trust domains. Owners usually do not put full trust on the cloud. For example, the cloud may delete rarely or never accessed data to save storage, or launch a replay attack by sending stale data to users. Therefore, there must be some mechanism to check the freshness of retrieved data. Signature and PoS technique only guarantee the integrity of cloud data, but they cannot ensure the data freshness. We address this issue by combining key regression [34] with Merkle hash tree [10] to

978-1-4799-8006-2/15 $31.00 © 2015 IEEE
DOI 10.1109/CCGrid.2015.90

build a version authentication tree for a block group to guarantee the freshness of retrieved blocks.

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

- Confidentiality. Data are encrypted by the owner before outsourcing to the cloud, thus unauthorized users and 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 with instantaneous freshness check for retrieved data to resist potential replay attacks, which is especially important for data-based electronic business that rely on the real-time transmission of data.

II. Background and Related Work

A. 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 [32], several secure storage systems have been proposed recently, such as Cepheus [2], SNAD[4], Plutus [5], SiRiUS [6] and SUNDR [7, 8]. 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 [5] emphasizes on the secure sharing of encrypted data, where key rotation is used to facilitate the task of key management so that lazy revocation can be easily implemented for a block group. SiRiUS [8] 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 [7, 8] 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 large 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.

But a common trend can be found in these designs, that providing owners with direct control over permission authorization and revocation is of much significance. This is because it not only alleviates the cryptographic overhead on cloud servers, but also increases user's confidence in data security. After all, letting owners dominate the access control policy can more or less compensate for their lost physical control over their outsourced data.

CloudProof [27] 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 prove any misbehavior to a third party. CloudProof adopts verifiable attestations and chained hash to periodically audit access records of users and cloud servers to detect violations of security. But their freshness guarantee is actually a post-check measure, which is of limited meaning for data-based businesses decision. CS2 [26] is probably the most related previous study to our work, which 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.

B. Proofs of Storage

Proofs of Storage [21] can be traced back to remote integrity check schemes [14], 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: Proof of Retrievability (PoR) [17, 18] and Provable Data Possession (PDP) [20]. Roughly speaking, a PoR is a challenge-response protocol that enables a service provider to demonstrate to a client that his data is intact and 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 both PDPs and PoRs could be found in work [13, 15, 16, 22, 19, 21]. Our system adopts PDP for that PoRs require the use of erasure code to outsourced data, which prevents users from auditing frequently updated data. Generally, there are several trends in the development of integrity auditing schemes.

From accessing the entire file to accessing part of the file. Earlier schemes [14, 15, 16] aim at using the entire data file to generate deterministic proofs on data integrity. Such plain solutions bring expensive computation overhead at server side in proof computation phase, and they also lack of practicality when dealing with large-size data files. Hence, represented by "sampling" method in PoR and PDP, later schemes [17, 20, 13, 15, 16, 22, 19, 21] all provide probabilistic proofs computed by accessing part of the file, which greatly enhances the auditing efficiency.

From private auditing to public auditing. In private auditing schemes [14, 15], only data owner who possesses the private key can perform the auditing task, which may potentially overburden the owner. While public auditing schemes [20, 22, 23, 24, 21, 25] allow anyone who possesses the public key to perform the auditing, which makes it possible for the auditing task to be delegated to a third party, who performs the auditing for the owner and honestly reports the result to him.

From static data to dynamic data. Earlier schemes [17, 18, 20, 13, 15, 16, 22, 19, 21] intend to audit static archive data, where users usually read these data and seldom update them. So these schemes mainly focus on static data auditing and don't 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 with data dynamics support. To our knowledge, only schemes in [23, 24, 25, 35] provide address this problem, yet they all have flaws in providing both data dynamics and auditing efficiency, especially for large amount of data.

C. Notation and Preliminaries

Merkle hash tree. A Merkle Hash Tree (MHT) [10] 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 is used to authenticate the values and logical positions of data blocks. 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 $h(v_2)$, $h_b = h(h(v_1)||h(v_2))$, $h(v_5)$, $h_e = h(h(v_5)||h(v_6))$, $h_a = h(h_b||h_c)$, $h_d = h(h_e||h_f)$ and $h_r = h(h_a||h_d)$. 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.

Bilinear map. A bilinear map is a map $e: G_1 \times G_2 \longrightarrow G_T$, where G_1 and G_2 are two Gap Diffie-Hellman groups of prime order p, and G_T is another multiplicative cyclic group with the same order. A bilinear map has the following properties [33]: 1) Computable: there exists an efficiently computable algorithm for computing e; 2) Bilinear: for all $h_1, h_1 \in G$ and $a, b \in \mathbb{Z}_p$, $e(h_1{}^a, h_2{}^b) = e(h_1, h_2)^{ab}$; 3) Non-degenerate: $e(g_1, g_2) \neq 1$, where g_1 and g_2 are generators of G_1 and G_2.

Key regression. Key regression [34] is a secure updated version of key rotation appeared in [5], 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 [11] provides the ability of encrypting a message to a subset of users. Only users in the subset can decrypt the broadcast result. In cryptographic storage, rather than directly encrypting the data content , broadcast encryption can be used to store a key in a secure and efficient way so that only authorized users can recover the key (broadcast message), whereas revoked users cannot find sufficient information to recover the key.

III. Design and Implementation

A. Security Architecture and Threat Model

Our security architecture is illustrated in Fig. 2, there are three roles in our system: (1) Data owner, who has large amount of data to be stored in the cloud, and is responsible for granting and revoking users' access permissions; (2) Data users, who read or write the outsourced data according to their authorized rights granted by the owner; (3) Cloud, who has massive storage space and computation power that users do not possess, stores and manages owner's data.

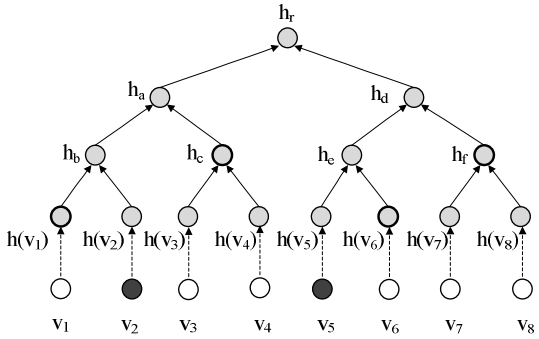

Fig. 1. Merkle hash tree for element authentication

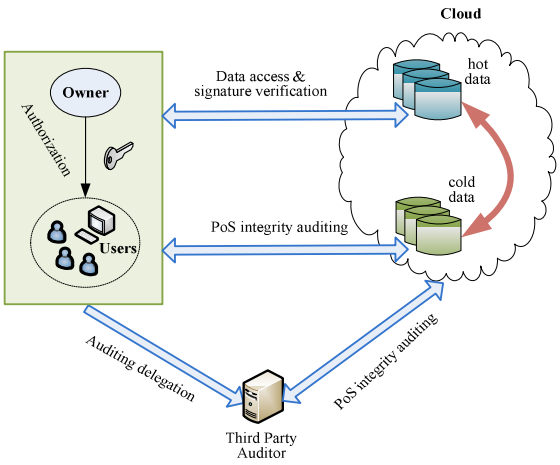

Fig. 2. Security architecture

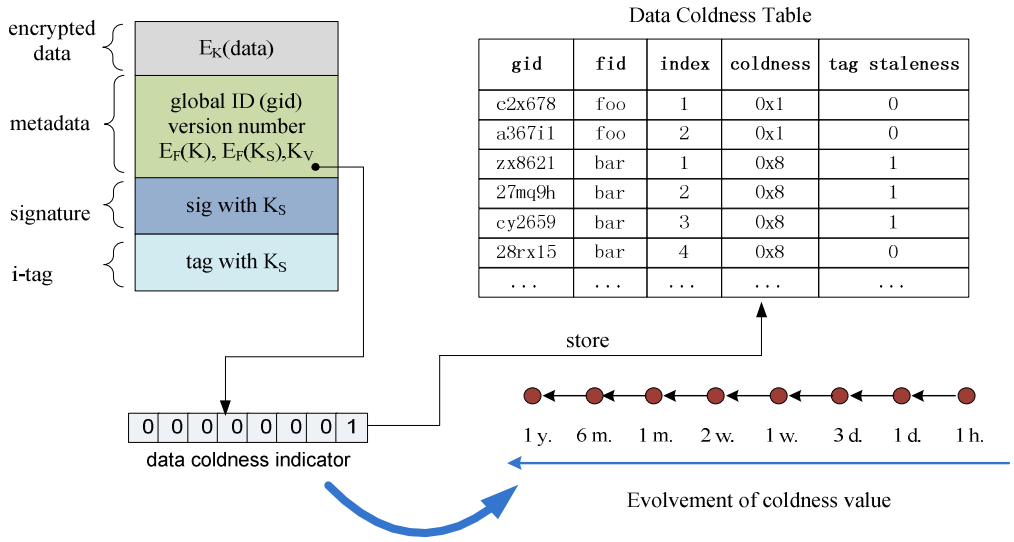

Fig. 3. Block structure and the coldness table

TABLE I NOTATIONS AND KEYS

Notation	Description
$E_K(D)$	encrypt a block with read key K
$E_F(K)$	Broadcast encryption of a key
$h()$	a collision-resistant hash function
gid	global id of a data block
sig	signature of a data block signed with KS
tag	a block's tag used for PoS auditing
coldness	the time interval that the data has not been accessed
K	Encryption key for a block group
K_S, K_V	Signing key and verification key for a block group
K_T^R, K_T^U	Root signing key and root verification key

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 or never accessed data. Additionally, we assume the existence of a public key infrastructure (PKI) to provide a strong binding with user id and generate public-private key pairs. Each entity can easily signs its message with its private key and others can verify the signature with his public key.

B. Data Organization and Access Control

Compared with centralized key management widely adopted in traditional storage systems, decentralized key management policies are receiving more and more attentions in recent secure storage designs. Represented by Cepheus[1], SNAD[4], Plutus [5] and SiRiUS [6], these systems usually provide a symmetric key for data encryption and an asymmetric key-pair for signing and verification. Roughly speaking, these systems differentiate read and write privilege by providing different keys for read and write access. Furthermore, these systems tend to use a lockbox structure to securely store the keys common to a group of users by encrypting these keys with each user's public key, and storing one lockbox per user as attached metadata. However, in such systems, revoking a user from a group needs to generate a new key and re-encrypt the lockbox of each user. Moreover, if lazy revocation policy is employed, then after several revocations, there may exist multiple versions of keys for a block group, and all these keys should be retained accessible, which makes the management of these keys a tedious task.

In our design, 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. Furthermore, we employ broadcast encryption [11] to accomplish the task of key storage and distribution. Since broadcast encryption let a broadcaster (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 can use his private key to decrypt the broadcast message $E_F(K_E)$ to get the encryption key K_E, so is with the signing key K_S. These broadcast messages are stored as part of the metadata attached with the encrypted block. As a result, the trivial task of real-time key distribution by an online owner or a key distribute centre is avoided. Notations and keys in our system are listed in Table 1.

We organize data in the following format: (*encrypted block, metadata, signature, tag*), as illustrated in Fig. 3. A block's metadata mainly includes a global id (*gid*) which uniquely identifies a data block globally, a version number indicating the update times, the signature verification key K_V and the broadcast message of read key K_E and write key K_S. 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 later PoS auditing.

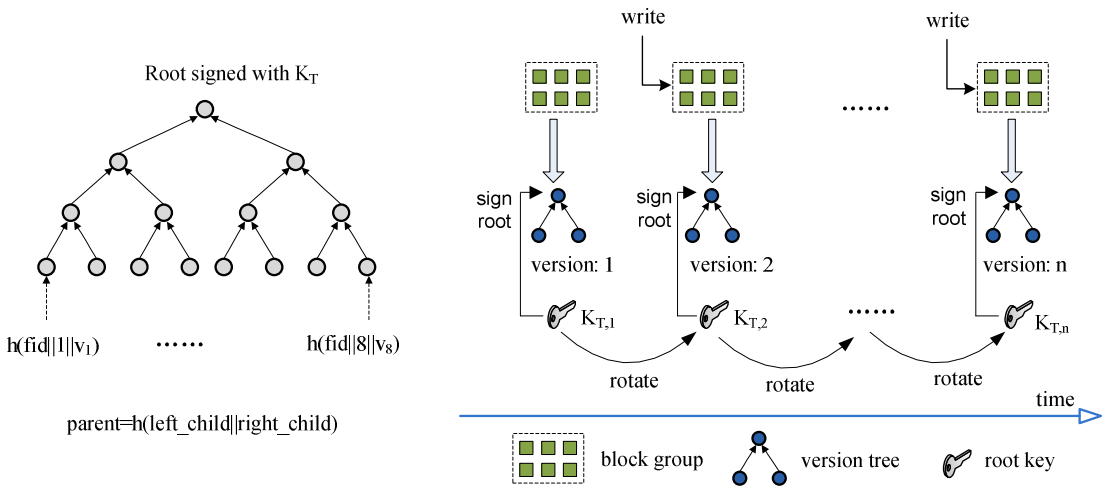

Fig. 4. Construction and update of a version authentication tree.

In our scheme, we employ the Merkle hash tree [10] 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 the block index in the file, v_i refers to a block's version number, as illustrated in Fig. 4. Further, we adopt key regression [34] technique to sign the tree root with the latest version of the root signing key K_T, which is used to sign the VAT root of a block group only. 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 will also 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.

Reading and writing a block. When a user u sends a request for a data block m_i to the cloud, the cloud replies with the encrypted block $E_K(m_i)$ and its metadata (global id, version number, broadcast encryption result of encryption key and signing key, verification key, signature) $\{gid_i, v_i, E_F(K), E_F(K_S), K_V, sig_i\}$, the signed VAT root $Sig_{K_T}(h(r))$ of the block's group, and the auxiliary authentication information Ω (the sibling nodes on the path from the requested node to the tree root).

On receiving the message from the cloud, user u firstly decrypts the broadcast message $E_F(K)$ and $E_F(K_S)$ to recover the encryption key K and the signing key K_S, verifies the block signature sig_i for integrity check. If succeeds, the user re-computes the VAT root using the value $fid||i||v_i$ of the requested block and AAI 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 for freshness check. If they are equal to each other, then the user can be sure that the retrieved data block from the cloud is latest.

Now the user decrypts the encrypted block $E_K(m_i)$ to read and modify the data block m_i according to his requirement. When the user finishes the update of block m_i, he also updates the block's metadata by incrementing the version number by one, re-encrypts the modified block m_i' and generates its new signature sig_i'. 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 an update message $\{E_K(m_i'), gid_i, v_i + 1, E_F(K), E_F(K_S), K_V, sig_i'\}$ and the new signed VAT root $Sig_{K_T'}(h(r'))$ to the cloud for storage.

The cloud verifies the integrity of the modified block, re-constructs the VAT using the new version number of the modified block, and verifies $Sig_{K_T'}(h(r'))$ from the user with the re-computed VAT root. If succeeds, the cloud stores the block (with its metadata and signature) and the new VAT root.

Note that the instantaneous freshness check relies on the latest root signing key K_T maintained by the client. According to [34], 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 get its public key for root verification. 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 for the stale data to pass the root authentication check at the client side.

C. PoS Auditing

In a PoS scheme, a data file is fragmented into multiple blocks of the same size, then for each block a tag is computed, e.g., $\sigma_i = (H(i) \cdot u^m)^x$. As both the block m_i and its index i are used to compute the tag, there exists a one-to-one correspondence between a block and its tag. Finally, a tag is signed with a user's private key x, hence it could not be forged without possession of 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, each $v_i \in \mathbb{Z}_p$ represents a random coefficient. On receiving $chal$, the CSP will compute 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 [28], the one-to-one correspondence between a block and its tag is kept in μ and σ. At verification, the auditor re-computes $H(i)$ of requested blocks, and verifies the validity of the proof by verifying the correspondence between μ and σ, namely, they must satisfy the bilinear pairing equation in [33].

Generally, embedding $H(i)$ in tag computation is to authenticate blocks lest a misbehaved server should use 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 embedding of block index, whenever a block is inserted or deleted, the indices of all following blocks will change, so tags of these blocks have to be re-computed, which is unacceptable for its high cost in computation, since tag computation need to treat the whole data block as an exponent. This problem makes data dynamics in PoS schemes a difficult task.

To our knowledge, only research in [23, 24, 25, 35] address this problem, but these schemes have their limitations. Dynamic PDP scheme proposed in [23] only supports limited operations and block insertion is not supported. Erway et al. [24] proposed to use the rank based skip list to uniquely differentiate among blocks their communication cost is linear to the number of blocks, and there's no explicit implementation of public verifiability in their scheme. Wang et al. [25] 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. [35] 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.

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 $\sigma_i = (H(gid) \cdot u^{m_i})^{\alpha}$, where gid is embedded in a block's metadata and its integrity is protected by the attached signature. The algorithms of our dynamic PoS scheme are described as follows.

Let G_1, G_2 and G_T be multiplicative cyclic groups of prime order p, and $e: G_1 \times G_2 \to G_T$ be a bilinear map. Let g be the generator of G_2 and $H(\cdot): \{0,1\}^* \to G_1$ be a secure map-to-point hash function which maps strings uniformly to group G_1. The data owner runs $KeyGen(\,)$ to generate the public-private key pair. By running $TagGen(\,)$ the data file F is pre-processed with erasure codes and fragmented, and tags for blocks are computed.

$KeyGen(1^k)$. The data owner chooses a random element $\alpha \to \mathbb{Z}_p$ and a random element $u \to G_1$, computes $v \to g^{\alpha}$ and $w \to u^{\alpha}$. The secret key is $sk = \alpha$ and the public key is $pk = (v, w, g, u)$.

$TagGen(sk, F)$. Given a data file $F = \{m_1, m_2, \dots, m_n\}$. 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 $I = \{s_1, s_2, \dots, s_c\} \subseteq \{1,2,\dots,n\}$, 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 $v_i \in \mathbb{Z}_p$ represents a random coefficient.

$ProofGen(chal, F, \Phi)$. On receiving the auditing request, the server computes its integrity proof as $\mu = \sum_{i \in I} v_i . m_i$ 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.

$ProofVerify(pk, chal, \pi)$. On receiving the integrity proof (μ, σ), the verifier should firstly search 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:

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

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

Data Dynamics. In our block organization, we allocate a globally unique identifier for each block, which is used to eliminate the passive effect of block index used in tag computation. Our design principle of dynamic PoS scheme is to let a block's index indicate the logical position in its file only, and to use a block's gid for its tag computation. Thus, block indices of all blocks in a file always appear as a consecutive sequence $1,2,\dots,n$, while the sequence of $gids$ 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 validity of integrity proof from the cloud.

1) Block 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, m'_k and σ'_k refer to the modified block and its new tag, and $Q = \{(i, v_i)\}_{i \in I \cap k \in I}$ 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.

2) Block Insertion. Block insertion will change the logic order of blocks after the insertion position, thus the *gid-index* mapping should be updated. When a user is to insert a new block m_k at position k, he firstly allocates an unused gid to the new block and computes its tag $\sigma_k = (H(gid) \cdot u^{m_k})^{\alpha}$. Then the user sends an update request $(O(I), k, m_k, \sigma_k)$ to the server, where $O(I)$ refers to insertion, m_k and σ_k denote the new block and its tag. Along with the update request is a small challenge set $\{(i, v_i)\}_{i \in I \cap k \in I}$ 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.

3) Block 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. Along with the update request is a small challenge set $Q = \{(i, v_i)\}_{i \in I \cap k \in I}$ with the deleted block m_k included (since the k-th position of the coldness table is now occupied by the subsequent record, we authenticate the deletion operation by authenticating the new record at the k-th position). Upon receiving the update request, the server deletes the specified block m_k and its tag 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 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., $c = 5$) to accelerate the 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 in the k-th record into gid'.
- Insertion. The user updates the coldness table by inserting the new record containing gid' at the k-th 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 k-th record and subtracting the index field of all following records by one.

For block insertion and deletion, the index field of all following 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.

For large-scale data, the coldness table would be huge. Since the *gid* values of challenged blocks are needed by auditors, its integrity and freshness will affect the correctness of auditing. For simplification of implementation and management, we use an extraction from the coldness table for PoS auditing. Specifically, we extract the $gid, fid, index$ fields from the coldness table to make a small *gid-index* mapping table for each block group, then recursively compute the Merkle hash root $MHT\{h(gid_i \parallel fid_i \parallel i)\}_{1 \le i \le n}$ and sign it with another signing key (different from the VAT root signing key), which is to be rotated to a new version upon each update of the *gid-index* table. The freshness check of *gid-index* table is similar to that of a VAT tree root. 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 is 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 has to be authenticated. We address this problem in our future work to provide more efficient PoS auditing in a secure storage system.

IV. 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. On the other hand, the cloud can deduce some information by recording the access patterns and analyzing the access frequency of some data. How to hide these information from data access is beyond our work.

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

Freshness. Each time a user get a block from the cloud, he would authenticate the VAT root using the latest version of root signing key. Due to the security of a RSA based key regression scheme [34], it is impossible for an adversary to guess the new version key without the secret master key. Therefore, if the cloud sends stale a data block and its VAT root (signed with an old root signing key) to the user, then it is impossible for the stale data to pass the root authentication check using the latest root signing key keeped by the client.

We now prove the security of our dynamic PoS auditing scheme by proving the following theorem.

Theorem 1. If the signature scheme used for file 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. We prove the theorem in the random oracle model by using a series of games defined in [22]. Game 0 is simply the challenge game, where the adversary can make store queries and undertake PoS protocol executions with the environment. Game 1 is the same as Game 0, except that the challenger keeps a list of all signed tags ever issued as part of a PoS protocol query. If the adversary ever submits a tag either in initiating a PoS protocol or as the challenge tag, the challenger will abort if it is a valid tag that has never been signed by the challenger. Game 2 is the same as Game 1, except that the challenger keeps a list of its responses to adversary's store queries.

Let $\pi = (\mu, \sigma)$ be the expected response form an honest prover, which satisfies $e(\sigma, g) = e((\prod_{i \in I} H(gid_i)^{v_i}) \cdot u^{\mu}, v)$. Assume the adversary's response is $\pi' = (\mu', \sigma')$, which satisfies $e(\sigma', g) = e((\prod_{i \in I} H(gid_i)^{v_i}) \cdot u^{\mu'}, v)$. 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 $(g, g^{\alpha}, h) \in G$, the simulator is to output h^{α}. The simulator sets $v = g^{\alpha}$, randomly picks $r_i, \beta, \gamma \in \mathbb{Z}_p^*$ and lets $u = g^{\beta}h^{\gamma}$. With similar method in [22], the simulator answers signature queries as $H(gid_i) = g^{r_i}/(g^{\beta \cdot m_i} \cdot h^{\gamma \cdot m_i})$. Finally, when the adversary outputs $\pi' = (\mu', \sigma')$. We obtain $e(\sigma'/\sigma, g) = e(u^{\Delta\mu}, v) = e((g^{\beta} \cdot h^{\gamma})^{\Delta\mu}, g^{\alpha})$. From this equation, we have

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

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

Game 3 is the same as Game 2, with one difference: if in any PoS instances the adversary succeeds (cause the challenge to abort) and the adversary's response (μ', σ') to query is not equal to the expected response (μ, σ). And in Game 2, we have proved that $\sigma' = \sigma$, so it is only $\mu' \neq \mu$. Define $\Delta\mu = \mu' - \mu$, the simulator answers the adversary's queries. Finally, the adversary outputs forged proof (μ', σ'). Now we have

$$e(\sigma', g) = e\left(\left(\prod_{i \in I} H(gid_i)^{v_i}\right) \cdot u^{\mu'}, v\right) = e(\sigma, g) = e((\textstyle\prod_{i \in I} H(gid_i)^{v_i}) \cdot u^{\mu}, v)$$

From this equation, we have $u^{\mu'} = u^{\mu}$ and then $1 = u^{\Delta\mu}$. So we have $\Delta\mu = 0 \ mod \ p$.

As analyzed above, there is only negligible difference between the adversary's success probabilities in these games. This completes the proof.

V. 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 2.4GHz, 4GB RAM and a 7200 RPM 2TB 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 hashing, 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 [3]. All experiment results are on the average of 12 trials, with the top and bottom results excluded.

A. Cryptographic Cost

TABLE II. Constant Cryptography Primitives

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

TABLE III. Involved Primitives in User Operations

cryptographic primitive	block create	block read	block write	revoke reader	revoke writer
signature verification		√	√		
broadcast decryption		√	√		
block decryption		√	√		
block encryption	√		√		
signature generation	√		√	√	√
tag generation	√		√		
broadcast encryption	√			√	√
AES key generation	√				
RSA key generation	√				
rotate encryption key				√	√
rotate signing key					√

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 block. Additionally, writing a block involves re-encrypting the modified block and generating a new signature for the block. Fig. 5 depicts the cryptographic cost of reading and writing a block, we can see that the growth of cryptographic overhead increases slowly when block size is less than 64KB, this is because the broadcast decryption overhead takes more than 95% of the total read cost. But when block size exceeds 64KB, the main influence of the read cost is AES encryption/decryption. (e.g., broadcast encryption cost takes 31.6% while AES decryption takes almost 62% at 1024KB).

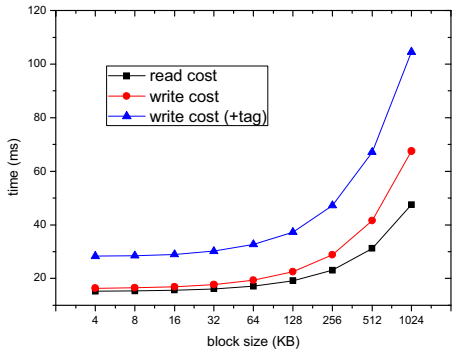

Fig. 5. Cryptographic cost of read and write. "+tag" means to immediately generate a block's tag after modification.

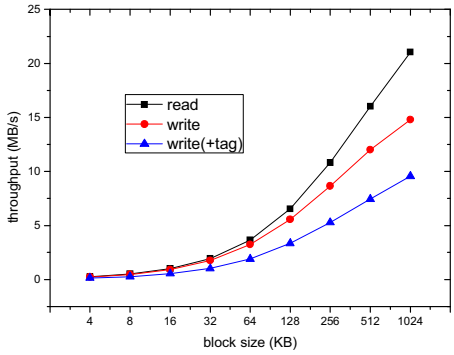

Fig. 6. Throughput of reading and writing a 100-MB file. "+tag" means to immediately generate a block's tag after modification.

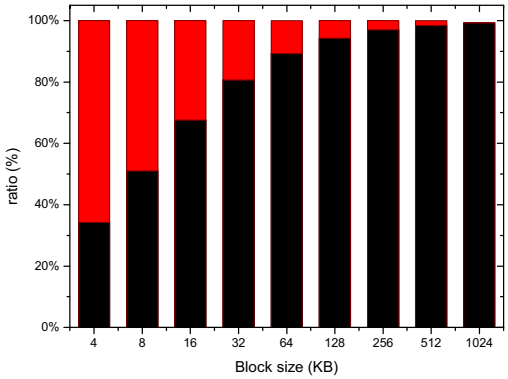

Fig. 7. Ratios of block data and metadata (including signature and tag) in different block size.

Table II depicts the cryptographic primitives which incur a constant overhead. Table III depicts the involved primitives of block operations and user revocations. Among them, we can see that the cost of rotating a RSA key is the most expensive, which explains why revoking a writer incurs much more overhead than revoking a reader. This is because the underlying RSA key rotation (revoking a writer) costs much more than the symmetric encryption key rotation (revoking a reader).

B. Throughput

To measure the throughputs, we choose a 100-MB file, fragment it into blocks with the block size varying from 4KB to 1MB. For each block, an encryption key and a RSA key pair are generated, and the metadata and signature are attached with the encrypted block. These block files are stored on different storage nodes. 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. From Fig. 6, we observe that the throughput of read and write are close at small block size, but the gap increases when block size exceeds 128KB. This is because the overhead of re-encryption and signature generation takes a greater portion in total cost at big block size (e.g., 7% and 30% for 4KB and 1024KB, 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 reading or writing a same-size file decreases.

C. Storage Overhead

From our block structure, a block's metadata consists of a gid, a version number, the broadcast message of K and K_s, K_v, the block signature and PoS tag. In implementation, we allocate 10 bytes for gid and 4 bytes forversion number. 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 in the number of users. Thus, the broadcast result consists of $x \times n + (2 \times n + 1) \times 16 + 32$ 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 a 100-user set authorized to access a block, the broadcast result is 3648 bytes. For small groups with less than 100 users, e.g., a group of 20 users, the broadcast result is about 768 bytes.

In this way, we can get the additional storage (metadata and signature) for a block is $10 + 4 + 3648 \times 2 + 128 + 128 + 128 = 7694$ bytes. Moreover, a block takes a record in the coldness table, which amounts to about 23 bytes for each record (4 bytes for fid and 8 bytes for index). Then we get the total storage cost for a single data block is about 7717 bytes, which takes a percentage of less than 3% when the block size exceeds 256 KB, as illustrated in Fig. 7. Generally, if we adopt big block size for data organization (e.g., greater than 256KB), then the ratio of additional storage will take a little portion in the total storage.

VI. Conclusion

The aim of this paper is to provide a secure storage system for the cloud, where confidentiality, full integrity and freshness guarantee are achieved. This paper presents the design, implementation and evaluation of such a secure storage system. We devise secure and flexible architecture and data organization structure for encrypted storage on untrusted cloud servers. Moreover, we seamlessly integrate traditional cryptographic storage design and efficient 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 are reasonable and acceptable.

Acknowledgment

Firstly, the authors would like to thank the anonymous referees of CCGrid 2015 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] K. Fu, "Group sharing and random access in cryptographic storage file system," Master's thesis, Dept. Computer Science and Eng., Massachusetts Inst. of Technology, 1999.

[2] M. Blaze, "A cryptographic file system for UNIX," Proc. 1st ACM Conf. Computer and Communications Security (CCS), 1993, pp. 9-16.

[3] K. Fu, M. F. Kaashoek, and D. Mazieres, "Fast and secure distributed read-only file system," ACM Trans. on Computer Systems, 1999.

[4] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C., Reed, "Strong security for network-attached storage," Proc. 1st USENIX Conf. File and Storage Technologies (FAST), 2002, pp. 1-13.
[5] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: scalable secure file sharing on untrusted storage," Proc. 2nd USENIX Conf. File and Storage Technologies (FAST), 2003, pp. 29-42.
[6] E. J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRius: securing remote untrusted storage," Proc. Network and Distributed System Security Symposium (NDSS), 2003, pp. 131-145.
[7] D. Mazieres, and D. Shasha, "Building secure file systems out of byzantine storage," Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC), 2002, pp. 108-117.
[8] J. Li, M. Krohn, D. Mazieres, and D. Shasha, "SUNDR: secure untrusted data repository," Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI), 2004.
[9] C. P. Wright, M. Martino, and E. Zadok, "NCryptfs: a secure and convenient cryptographic file system," Proc. USENIX Annual Technical Conf (ATC), 2003.
[10] R. C. Merkle, "Protocols for public key cryptosystems," Proc. IEEE Symp. Security and Privacy, 1980.
[11] D. Boneh, C. Gentry, and B. Waters, "Collusion resistant broadcast encryption with short ciphertexts and private keys," Proc. Advances in Cryptology-CRYPTO, 2005, pp. 258-275.
[12] R. Pletka and C. Cachin, Cryptographic security for a high-performance distributed file system. Proc. 24th IEEE Conf. on Mass Storage Systems and Technologies (MSST), 2007.
[13] E.C. Chang and J. Xu, "Remote integrity check with dishonest storage server," Proc. 13th European Symp. Research in Computer Security (ESORICS), 2008, pp. 223–237.
[14] Y. Deswarte, J. J. Quisquater, and A. Saïdane, "Remote integrity checking," Proc. 5th Working Conf. Integrity and Int'l Control in Information Systems VI, 2003, pp. 1-11.
[15] M. A. Shah, R. Swaminathan, and M. Baker, "Privacy-preserving audit and extraction of digital contents," IACR Cryptology ePrint Archive, Report 2008/186, 2008.
[16] F. Sebé, J. Domingo-Ferrer, A. Martinez-Balleste, Y. Deswarte, and J. J. Quisquater, "Efficient remote data possession checking in critical information infrastructures," IEEE Trans. on Knowl. and Data Eng., vol. 20, no. 8, 2008, pp. 1034-1038.
[17] A. Juels, and J. B. S. Kaliski, "PORs: proofs of retrievability for large files," Proc. 14th ACM Conf. Computer and Comm. Security (CCS), 2007, pp. 584-597.
[18] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: theory and implementation," Proc. ACM Workshop Cloud Computing Security (CCSW), 2009, pp. 43-54.
[19] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," Proc. Theory of Cryptography (TCC), 2009, pp. 109-127.
[20] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," Proc. 14th ACM Conf. Computer and Comm. Security (CCS), 2007, pp. 598-610.
[21] G. Ateniese, S. Kamara, and J. Katz. "Proofs of storage from homomorphic identication protocols," Proc. Advances in Cryptology-ASIACRYPT, 2009, pp. 319-333.
[22] H. Shacham, and B. Waters, "Compact proofs of retrievability," Proc. Advances in Cryptology - ASIACRYPT, 2008, pp. 90-107.
[23] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," Proc. Fourth Int'l Conf. Security and Privacy in Comm. Networks (SecureComm '08), 2008, pp. 1-10.
[24] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," Proc. 16th ACM Conf. Computer and Comm. Security (CCS), 2009, pp. 213-222.
[25] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," IEEE Trans. Parallel and Distributed Systems, vol. 22, 2011, pp. 847-859.
[26] S. Kamara, C. Papamanthou, and T. Roeder, "CS2: a searchable cryptographic cloud storage system," Technical Report MSR-TR-2011-58, Microsoft, 2011.
[27] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling security in cloud storage SLAs with CloudProof," Proc. USENIX Annual Technical Conf. (ATC), 2011.
[28] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," Proc. Advances in Cryptology-EUROCRYPT, 2003, pp. 416-432.
[29] E. Stefanov, M. V. Dijk, A. Oprea, a., and A. Juels, "Iris: a scalable cloud file system with efficient integrity checks," Proc. 28th Ann. Computer Security Applications Conf. 2012.
[30] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. "Athos: efficient authentication of outsourced file systems," Proc. Information Security Conference, 2008.
[31] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. "Venus: verification for untrusted cloud storage," Proc. Workshop on Cloud Computing Security, 2010.
[32] V. Kher and Y. Kim. "Securing distributed storage: challenges, techniques, and systems," Proc. ACM workshop on Storage security and survivability (StorageSS), 2005, 9-25.
[33] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," Proc. Advances in Cryptology-ASIACRYPT, 2001, pp. 514–532.
[34] K. Fu, S. Kamaram, and T. Kohno, "Key regression: enabling efficient key distribution for secure distributed storage," Proc. Network and Distributed Systems Security Symp. (NDSS), 2006.
[35] Y. Zhu, H. Wang, Z. Hu, G.J. Ahn, H. Hu, and S.S Yau, "Dynamic audit services for integrity verfication of outsourced storages in clouds," Proc. ACM Symp. Applied Computing (SAC), 2011, pp. 1550-1557.