This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

Dynamic and Public Auditing with Fair Arbitration for Cloud Data

Hao Jin, Hong Jiang, *Senior Member, IEEE,* and Ke Zhou

Abstract—Cloud users no longer physically possess their data, so how to ensure the integrity of their outsourced data becomes a challenging task. Recently proposed schemes such as "provable data possession" and "proofs of retrievability" are designed to address this problem, but they are designed to audit static archive data and therefore lack of data dynamics support. Moreover, threat models in these schemes usually assume an honest data owner and focus on detecting a dishonest cloud service provider despite the fact that clients may also misbehave. This paper proposes a public auditing scheme with data dynamics support and fairness arbitration of potential disputes. In particular, we design an index switcher to eliminate the limitation of index usage in tag computation in current schemes and achieve efficient handling of data dynamics. To address the fairness problem so that no party can misbehave without being detected, we further extend existing threat models and adopt signature exchange idea to design fair arbitration protocols, so that any possible dispute can be fairly settled. The security analysis shows our scheme is provably secure, and the performance evaluation demonstrates the overhead of data dynamics and dispute arbitration are reasonable.

Index Terms—Integrity auditing, public verifiability, dynamic update, arbitration, fairness.

1 Introduction

DATA oursourcing is a key application of cloud computing, which relieves cloud users of the heavy burden of data management and infrastructure maintenance, and provides fast data access independent of physical locations. However, outsourcing data to the cloud brings about many new security threats. Firstly, despite the powerful machines and strong security mechanisms provided by cloud service providers (CSP), remote data still face network attacks, hardware failures and administrative errors. Secondly, CSP may reclaim storage of rarely or never accessed data, or even hide data loss accidents for reputation reasons. As users no longer physically possess their data and consequently lose direct control over the data, direct employment of traditional cryptographic primitives like hash or encryption to ensure remote data's integrity may lead to many security loopholes. In particular, downloading all the data to check its integrity is not viable due to the expensive communication overhead, especially for large-size data files. In this sense, message authentication code (MAC) or signature based mechanisms, while widely used in secure storage systems, are not suitable for integrity check of outsourced data, because they can only verify the integrity of retrieved data and do not work for rarely accessed data (e.g., archive data). So how to ensure the correctness of outsourced data without possessing the original data becomes a challenging task in cloud computing, which, if not effectively handled, will impede the wide deployment of cloud services.

Data auditing schemes can enable cloud users to check the integrity of their remotely stored data without downloading them locally, which is termed as blockless verification. With auditing schemes, users can periodically interact with the CSP through auditing protocols to check the correctness of their outsourced data by verifying the integrity proof computed by the CSP, which offers stronger confidence in data security because user's own conclusion that data is intact is much more convincing than that from service providers. Generally speaking, there are several trends in the development of auditing schemes.

First of all, earlier auditing schemes usually require the CSP to generate a deterministic proof by accessing the whole data file to perform integrity check, e.g., schemes in [1], [2] use the entire file to perform modular exponentiations. Such plain solutions incur expensive computation overhead at the server side, hence they lack efficiency and practicality when dealing with large-size data. Represented by the "sampling" method in "Proofs of Retrievability" (PoR) [3] model and "Provable Data Possession" (PDP) [4] model, later schemes [5], [6] tend to provide a probabilistic proof by accessing part of the file, which obviously enhances the auditing efficiency over earlier schemes.

Secondly, some auditing schemes [3], [7] provide private verifiability that require only the data owner who has the private key to perform the auditing task, which may potentially overburden the owner due to its limited computation capability. Ateniese el al. [4] were the first to propose to enable public verifiability in auditing schemes. In contrast, public auditing schemes [5], [6] allow anyone who has the public key to perform the auditing, which makes it possible for the auditing task to be delegated to an external third party auditor (TPA). A TPA can perform the integrity check on behalf of the data owner and honestly report the auditing result to him [8].

Thirdly, PDP [4] and PoR [3] intend to audit static data that are seldom updated, so these schemes do not provide data dynamics support. But from a general perspective, data

- *Hao Jin and Ke Zhou are with Wuhan National Lab for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: khow.hust@gmail.com and k.zhou@hust.edu.cn.*
- *Hong Jiang is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, NE, USA. E-mail: jiang@cse.unl.edu.*

Manuscript received April 19, 2005; revised September 17, 2014.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

update is a very common requirement for cloud applications. If auditing schemes could only deal with static data, their practicability and scalability will be limited. On the other hand, direct extensions of these static data oriented schemes to support dynamic update may cause other security threats, as explained in [6]. To our knowledge, only schemes in [6], [9], [10] provide built-in support for fully data dynamic operations (i.e., modification, insertion and deletion), but they are insufficient in providing data dynamics support, public verifiability and auditing efficiency simultaneously, as will be analyzed in the section of related work.

From these trends, it can be seen that providing probabilistic proof, public verifiability and data dynamics support are three most crucial characteristics in auditing schemes. Among them, providing data dynamics support is the most challenging. This is because most existing auditing schemes intend to embed a block's index i into its tag computation, e.g., $H(i||v)$ in [4] or $H(name||i)$ in [5], which serves to authenticate challenged blocks. However, if we insert or delete a block, block indices of all subsequent blocks will change, then tags of these blocks have to be re-computed. This is unacceptable because of its high computation overhead.

We address this problem by differentiating between tag index (used for tag computation) and block index (indicate block position), and rely an index switcher to keep a mapping between them. Upon each update operation, we allocate a new tag index for the operating block and update the mapping between tag indices and block indices. Such a layer of indirection between block indices and tag indices enforces block authentication and avoids tag re-computation of blocks after the operation position simultaneously. As a result, the efficiency of handling data dynamics is greatly enhanced.

Furthermore and important, in a public auditing scenario, a data owner always delegates his auditing tasks to a TPA who is trusted by the owner but not necessarily by the cloud. Current research usually assumes an honest data owner in their security models, which has an inborn inclination toward cloud users. However, the fact is, not only the cloud, but also cloud users, have the motive to engage in deceitful behaviors. For example, a malicious data owner may intentionally claim data corruption against an honest cloud for a money compensation, and a dishonest CSP may delete rarely accessed data to save storage. Therefore, it is of critical importance for an auditing scheme to provide fairness guarantee to settle potential disputes between the two parties. Zheng et al. [11] proposed a fair PoR scheme to prevent a dishonest client from accusing an honest CSP, but their scheme only realizes private auditing. Kupccu [12] proposed general arbitration protocols with automated payments using fair signature exchange protocols [13]. Our work also adopts the idea of signature exchange to ensure the metadata correctness and protocol fairness, and we concentrate on combining efficient data dynamics support and fair dispute arbitration into a single auditing scheme.

To address the fairness problem in auditing, we introduce a third-party arbitrator(TPAR) into our threat model, which is a professional institute for conflicts arbitration and is trusted and payed by both data owners and the CSP. Since a TPA can be viewed as a delegator of the data owner and is not necessarily trusted by the CSP, we differentiate between the roles of auditor and arbitrator. Moreover, we adopt the idea of signature exchange to ensure metadata correctness and provide dispute arbitration, where any conflict about auditing or data update can be fairly arbitrated.

Generally, this paper proposes a new auditing scheme to address the problems of data dynamics support, public verifiability and dispute arbitration simultaneously. Our contributions mainly lie in:

- We solve the data dynamics problem in auditing by introducing an index switcher to keep a mapping between block indices and tag indices, and eliminate the passive effect of block indices in tag computation without incurring much overhead.
- We extend the threat model in current research to provide dispute arbitration, which is of great significance and practicality for cloud data auditing, since most existing schemes generally assume an honest data owner in their threat models.
- We provide fairness guarantee and dispute arbitration in our scheme, which ensures that both the data owner and the cloud can not misbehave in the auditing process or else it is easy for a third-party arbitrator to find out the cheating party.

The rest of the paper is organized as follows. Section 2 introduces the system model, threat model and our design goals. In Section 3 and 4, we elaborate on our dynamic auditing scheme and arbitration protocols. Further, we present the security analysis and performance evaluation in Sections 5 and 6, respectively. Section 7 surveys the related work. Finally, Section 8 concludes the paper.

2 PROBLEM STATEMENT

2.1 System Model

As illustrated in Fig. 1, the system model involves four different entities: the data owner/cloud user, who has a large amount of data to be stored in the cloud, and will dynamically update his data (e.g., insert, delete or modify a data block) in the future; the cloud service provider (CSP), who has massive storage space and computing power that users do not possess, stores and manages user's data and related metadata (i.e., the tag set and the index switcher); the third party auditor (TPAU) is similar to the role of TPA in existing schemes, who is a public verifier with expertise and capabilities for auditing, and is trusted and payed by the data owner (but not necessarily trusted by the cloud) to assess the integrity of the owner's remotely stored data; the third party arbitrator (TPAR), who is a professional institute for conflict arbitration and trusted by both the owner and the CSP, which is different to the role of TPAU.

Cloud users rely on the CSP for data storage and maintenance, and they may access and update their data. To alleviate their burden, cloud users can delegate auditing tasks to the TPAU, who periodically performs the auditing and honestly reports the result to users. Additionally, cloud users may perform auditing tasks themselves if necessary. For potential disputes between the auditor and the CSP, the TPAR can fairly settle the disputes on proof verification or data update . Note in following sections, we may use the terms "TPAU" and "auditor" interchangeably, so are the terms "TPAR" and "arbitrator".

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

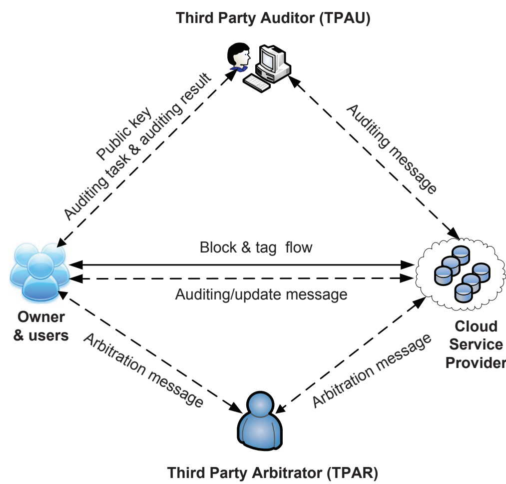

Fig. 1. The system and threat model.

2.2 Threat Model

Threat models in existing public auditing schemes [4], [5], [6], [14] mainly focus on the delegation of auditing tasks to a third party auditor (TPA) so that the overhead on clients can be offloaded as much as possible. However, such models have not seriously considered the fairness problem as they usually assume an honest owner against an untrusted CSP. Since the TPA acts on behalf of the owner, then to what extent could the CSP trust the auditing result? What if the owner and TPA collude together against an honest CSP for a financial compensation? In this sense, such models reduce the practicality and applicability of auditing schemes.

In a cloud scenario, both owners and CSP have the motive to cheat. The CSP makes profit by selling its storage capacity to cloud users, so he has the motive to reclaim sold storage by deleting rarely or never accessed data, and even hides data loss accidents to maintain a reputation. Here, we assume the CSP is semi-trusted, namely, the CSP behaves properly as prescribed contract most of the time, but he may try to pass the integrity check without possessing correct data. On the other hand, the owner also has the motive to falsely accuse an honest CSP, e.g., a malicious owner intentionally claims data corruption despite the fact to the contrary so that he can get a compensation from the CSP.

Therefore, disputes between the two parties are unavoidable to a certain degree. So an arbitrator for dispute settlement is indispensable for a fair auditing scheme. We extend the threat model in existing public schemes by differentiating between the auditor (TPAU) and the arbitrator (TPAR) and putting different trust assumptions on them. Because the TPAU is mainly a delegated party to check client's data integrity, and the potential dispute may occur between the TPAU and the CSP, so the arbitrator should be an unbiased third party who is different to the TPAU.

As for the TPAR, we consider it honest-but-curious. It will behave 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. Note that, while privacy protection is beyond the scope of this paper, our scheme can adopt the random mask technique proposed in [14], [15] for privacy preservation of auditing data, or the ring signatures in [16] to protect the identity-privacy of signers for data shared among a group of users.

2.3 Design Goals

Our design goals can be summarized as follows:

1) Public verifiability for data storage correctness: to allow anyone who has the public key to verify the correctness of users' remotely stored data;
2) Dynamic operation support: to allow cloud users to perform full block-level operations (modification, insertion and deletion) on their outsourced data while guarantee the same level of data correctness, and the scheme should be as efficient as possible;
3) Fair dispute arbitration: to allow a third party arbitrator to fairly settle any dispute about proof verification and dynamic update, and find out the cheating party.

3 Our Dynamic Auditing Scheme

This section presents our dynamic auditing scheme with dispute arbitration. After introducing notations and preliminaries, we firstly describe the idea of index switcher which keeps a mapping between block indices and tag indices. Then, we present our main scheme and show how to achieve data dynamics support using our index switcher. Finally, we briefly discuss the efficiency of index switcher update caused by dynamic operations.

3.1 Notation and Preliminaries

- F is the data file to be outsourced to the cloud, which is defined as a sequence of blocks of the same size $F = \{m_1, m_2, ..., m_n\}$, where each $m_i \in \mathbb{Z}_p$ for some large prime p.
- $H(\cdot) : \{0,1\}^* \to \mathbb{G}$ is a collision-resistant map-to-point hash function who maps a string with arbitrary length into a point in $\mathbb{G}$, where $\mathbb{G}$ is a cyclic group.
- $h(\cdot) : \{0,1\}^* \to \{0,1\}^l$ is a collision-resistant cryptographic hash function.

$BilinearMap$. A bilinear map is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, 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 [22]: (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 $\mathbb{G}_2$.

3.2 Definitions

Our dynamic auditing scheme with public verifiability and dispute arbitration consists of the following algorithms.

$KeyGen(1^k)$. This algorithm is run by the client, which takes as input security parameter 1^k and generates a public-private key pair (pk, sk).

$TagGen(sk, F, \Omega)$. This algorithm is run by the client, which takes as input a secret key sk and user's file $F \in$

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

$\{0,1\}^k$ as a collection of data blocks $\{m_i\}_{1\leq i\leq n}$, outputs a tag set $\Phi = \{\sigma_i\}_{1\leq i\leq n}$, where σ_i is the tag for block m_i. And a signature on the metadata Ω signed with owner's private key is produced.

$Commitment(pk, F, \Phi, \Omega)$. This algorithm is run by the cloud, which takes as input the whole block set F and the tag set Φ, generates an integrity proof and verifies its validity with pk. This process ensures that the tag set Φ received by the CSP is correctly computed from the block set F. If succeeds, a signature on the metadata Ω signed with CSP's private key is produced.

$ProofGen(chal, F, \Phi)$. This algorithm is run by the cloud, which takes as input a challenge request $chal$, the data file F and the tag set Φ, outputs an integrity proof π.

$ProofVerify(pk, chal, \pi)$. This algorithm is run by the auditor, which takes as input the public key pk, the challenge $chal$ and the integrity proof π, outputs $TRUE$ if the proof is verified as valid, which means that the file is stored intact on the server, or $FALSE$ otherwise.

$Update(F, \Phi, up_req)$. This algorithm is run by the cloud, which takes as input the data file F, the tag set Φ and an update record up_req from the client. It outputs the updated file F' and tag set Φ', and the update proof P of the dynamic operation.

$UpdateVerify(pk, up_req, P)$. This algorithm is run by the auditor, which takes as input the public key pk, the update request up_req and the update proof P from the cloud. If succeeds, it outputs $TRUE$, or $FALSE$ otherwise.

In our design, we do not have any additional requirement on the data to be stored on cloud servers. And we regard erasure codes as orthogonal to our auditing scheme because the owner can encode their data before outsourcing them to the cloud. Additionally, the absence of the $Commitment$ algorithm in existing schemes [4], [5] is because these schemes assume clients to be honest and they mainly focus on detecting a dishonest CSP. While in our threat model, we assume that both clients and CSP have the motive to cheat, thus ensuring the correctness of initial blocks and tags is indispensable for later dispute arbitration.

3.3 Index Switcher

In recent schemes [4], [5], the data file is first fragmented into multiple blocks of the same size, then for each block a tag is computed (e.g., $\sigma_i = (H(i) \cdot u^{m_i})^x$). As both the block m_i and its index i are used to compute its tag σ_i, there exists a one-to-one correspondence between m_i and σ_i. Additionally, a tag is signed with a user's private key x, it cannot be forged due to the unforgeability of secure signature schemes. To initiate an auditing, the auditor generates a challenge against a set of randomly selected blocks. The CSP computes the proof $\pi = (\mu, \sigma)$, where μ is computed from requested blocks and σ is computed from their tags. Due to the aggregative property of homomorphic verifiable tags [17], the one-to-one correspondence between a data block and its tag is also kept in μ and σ. The verification algorithm verifies the validity of the proof by verifying the correspondence between μ and σ, namely, they must satisfy some mathematic equation (e.g., bilinear pairing in [5]).

If we simplify the tag computation as $\sigma_i = (u^{m_i})^x$, then the server can cheat the auditor by using other non-designated blocks to compute the proof. Since the index is not embedded in a block's tag, the verification algorithm only verifies the correspondence between $(u^{m_i})^x$ and m_i. Therefore any blocks and their tags can pass the check as long as they are intact. In this sense, embedding $H(i)$ into tag computation serves to authenticate blocks. However, due to the index embedding, when a block is inserted or deleted, the indices of all following blocks change and their tags have to be re-computed, which is unacceptable because of its high computation cost.

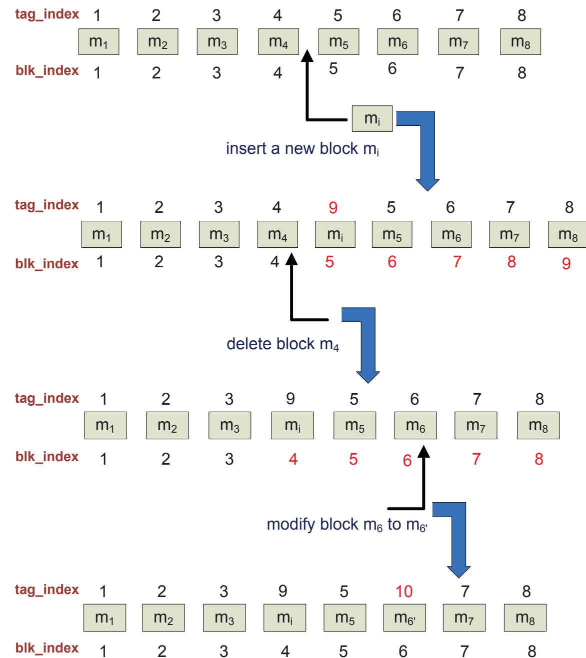

Fig. 2. An index switcher for handling data dynamics.

We address this problem by introducing an index switcher to maintain a mapping between block indices and tag indices. In our construction, tag indices are used in tag computation only, while block indices are used to indicate the logical positions of data blocks. This layer of indirection between block indices and tag indices can enforce block authentication as long as the index switcher be updated for each update. As a result, the expensive tag re-computation of following blocks (after the operation position) caused by dynamic update operations can be avoided, and the efficiency of handling data dynamics is greatly enhanced.

An index switcher is actually a table used to keep a mapping between block indices and tag indices. As illustrated in Fig.2, initially, tag indices and block indices appear as the same sequence $1, 2, \ldots, 8$. After insertion of a new block m_i at position 4, the block index of m_i is 5 while the tag index of m_i is 9 (5 is the block index of m_5 before insertion). After deletion of block m_4, the block indices of following blocks $(m_i, m_5, m_6, m_7, m_8)$ are incremented by one to keep the block index sequence consecutive, while tag indices of these following blocks do not change. After modification of m_6, its tag index is changed to 10. Here, we also allocate a new tag index for block modification, which is to resist potential replay attacks by a malicious CSP. For example, if a block's tag index does not change after modification, then the CSP can discard the update by using the old pair of (m_k, σ_k) during a later auditing, and the proof can still pass

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

the verification.

Generally, block update will make the tag index sequence unordered and inconsecutive, while the block index sequence always remains consecutive. In implementation, a global monotonously increasing counter can be used to generate a new tag index for each inserted or modified block. Moreover, block insertion and deletion will cause index pairs of all subsequent blocks (after the operation position) to be shifted backward or forward a position in the index switcher, and the block indices of these blocks to be incremented or decreased by one. The cost of index pair shifting is negligible as compared with the cost of tag recomputation of following blocks, e.g., a 10-GB data file using 2-KB fragmentation generates 5×2^{20} index pairs, shifting all these pairs through insertion or deletion cost less than 50 miliseconds.

Finally, since the index switcher is needed by the auditor during an auditing, its correctness affects the correctness of the auditing result. To guarantee the correctness of the index switcher and further the fairness of dispute arbitration, signatures on the updated index switcher have to be exchanged upon each dynamic operation, as we will further discuss it in the next section.

3.4 Dynamic Auditing Scheme

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 $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear map, and $H(\cdot) : \{0,1\}^* \rightarrow \mathbb{G}_1$ be a secure public map-to-point hash function, which maps a string $\{0,1\}^*$ uniformly into an element of $\mathbb{G}_1$. Let $Sig_{sk}(seq, \Omega) \leftarrow (h(seq||\Omega))^{sk}$ denote a signature on the concatenation of a sequence number seq and the index switcher Ω using the private key sk. Let sk_c and sk_s denote the private key of the client and the CSP, respectively. Then the scheme can be described as follows.

KeyGen. 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. Given a data file $F = \{m_1, m_2, \ldots, m_n\}$. For each block m_i, the owner computes its tag as $\sigma_i = (H(t_i) \cdot u^{m_i})^{\alpha}$, where t_i denotes the tag index of the block. Denote the tag set by $\Phi = \{\sigma_i\}_{1 \leq i \leq n}$. Initially, tag indices and block indices are the same sequence $1, 2, \ldots, n$, so tag computation can be simplified as $\sigma_i = (H(i) \cdot u^{m_i})^{\alpha}$, and the TPAR can easily construct his version of the index switcher. Then, the owner computes his signature on the index switcher $Sig_c = Sig_{sk_c}(seq_0, \Omega_0)$, where seq_0 is initialized to 0 and $\Omega_0 = \{(i, t_i = i)\}_{1 \leq i \leq n}$. Finally, the owner sends $\{F, \Phi, Sig_c\}$ to the CSP for storage and sends pk to the TPAR. The owner deletes its local copy of $\{F, \Phi\}$ and keeps the index switcher Ω.

Commitment. This process is to prevent a malicious owner from generating incorrect tags at the initial stage so that he can falsely accuse the cloud at a later time. The cloud generates deterministic proof from all received blocks and tags according to algorithm **ProofGen** and verify its validity with algorithm **ProofVerify**. If the verification succeeds, the cloud can be sure that all tags are correctly computed from received blocks, and then he sends his signature on the index switcher $Sig_s = Sig_{sk_s}(seq_0, \Omega_0)$ to the client for storage, where $seq_0 = 0$ and $\Omega_0 = \{(i, t_i = i)\}_{1 \leq i \leq n}$. The client also verifies the correctness of Sig_s, if succeeds, he keeps it; otherwise he contacts the TPAR for arbitration.

ProofGen. To audit the integrity of outsourced data, a public 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 the 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 used to compute the proof.

On receiving the auditing challenge $\{(i, v_i)\}_{i \in I}$, the server computes the 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. On receiving the integrity proof $\pi = (\mu, \sigma)$, the verifier firstly turns each block index $i \in I$ into its tag index t_i using the index switcher and gets $chal' = \{(t_i, v_i)\}_{i \in I}$. Then the verifier checks the correctness of the proof by verifying whether the following equation holds:

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

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

Update. A user updates the k-th block by performing one of the following operations.

- **Modification.** The user modifies the k-th block m_k into m'_k. He allocates an unused tag index t'_k for the modified block m'_k and computes its new tag as $\sigma'_k = (H(t'_k) \cdot u^{m'_k})^{\alpha}$. Then the user updates the index switcher to Ω' and sends an update request $up_req = \{seq, O(M), k, t'_k, m'_k, \sigma'_k, Q', Sig_{sk_c}(seq, \Omega')\}$ to the server, where m'_k and σ'_k refer to the modified block and its new tag, $O(M)$ denotes modification, k and t'_k denote the block index and its new tag index, Ω' is the updated index switcher, and $Q' = \{(i, v_i)\}_{i \in I \cap k \in I}$ is a small challenge set with the modified block m'_k included.
- **Insertion.** The user inserts a new block at the k-th position. He allocates an unused tag index t'_k to the new block m'_k and computes its tag as $\sigma'_k = (H(t'_k) \cdot u^{m'_k})^{\alpha}$. Then the user updates the index switcher to Ω' and sends an update request $up_req = \{seq, O(I), k, t'_k, m'_k, \sigma'_k, Q', Sig_{sk_c}(seq, \Omega')\}$ to the server, where m'_k and σ'_k refers to the block to be inserted and its tag, $O(I)$ denotes insertion, k and t'_k denote the insertion position and the new block's tag index, Ω' is the updated index switcher, and $Q' = \{(i, v_i)\}_{i \in I \cap k \in I}$ is a small challenge set with the new block m'_k included.
- **Deletion.** The user deletes the block at the k-th position. He updates the index switcher to Ω' and sends an update request $up_req = \{seq, O(D), k, Q', Sig_{sk_c}(seq, \Omega')\}$ to the server, where $O(D)$ denotes deletion and k specifies the deletion position, Ω' is the updated index switcher, and $Q' = \{(i, v_i)\}_{i \in I \cap k \in I}$ is a small challenge set with the new block at the k-th position included (since the k-th position is now occupied by the next

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

block, we authenticate the deletion operation by authenticating the new block at the k-th position).

Upon receiving up_req, for block modification and insertion, the CSP verifies the correctness of m'_k and σ'_k by verifying $e(\sigma'_k, g) \overset{?}{=} e(H(t'_k) \cdot u^{m'_k}, v)$. If succeeds, the CSP then verifies the correctness of $Sig_{sk_c}(seq, \Omega')$ from the client by updating his index switcher with the operation information in the update record. If both verification succeed, for modification, the CSP replaces the old block m_k and its tag σ_k with the new block m'_k and its tag σ'_k; for insertion, the CSP adds m'_k and σ'_k into F and Φ respectively. Block deletion only involves the signature verification, if succeeds, the CSP deletes the specified block m_k and its tag σ_k from F and Φ. Finally, the CSP computes $Sig_{sk_s}(seq, \Omega')$ using his private key, computes the integrity proof $\pi' = (\mu', \sigma')$ according to the small challenge set Q', and sends $(\mu', \sigma', Sig_{sk_s}(seq, \Omega'))$ to the user for update verification.

UpdateVerify. On receiving the update request $(\mu', \sigma', Sig_{sk_s}(seq, \Omega'))$, the user verifies the correctness of (μ', σ') according to equation (1). Then he verifies the validity of the signature $Sig_{sk_s}(seq, \Omega')$ using the CSP's public key. If both verification succeed, the user can be assured that the CSP has indeed updated the block and its tag, then he stores the CSP's signature.

Update of the index switcher is as follows. For modification, the user modifies the tag index of the k-th index pair into t'_k. For insertion, the user inserts the new index pair (k, t_k) at the k-th position, shifts all following pairs backward a position and increments by one the block indices of these pairs. For deletion, the user deletes the k-th index pair, shifts all following pairs forward a position and decreases by one the block indices of these pairs.

The protocols of data update (block modification, insertion and deletion) all have the default integrity auditing included in order to authenticate the update operation at the server side (compared with the integrity check purpose of the auditing scheme). Thus, we can choose the number of challenged blocks to be a small number (e,g., $c = 5$) to accelerate the speed of proof generation and verification. Furthermore, the signature exchange in the **Update** and **UpdateVerify** is to reach an agreement on the updated metadata, i.e., the index switcher. The exchanged signature signed by one party and kept by the other party is necessary for later arbitration, where it is to reveal the metadata confirmation by both parties in the previous round of update, as we will see it in the next section.

Discussion We assume a single-writer and many-readers scenario in our scheme, that is, only the data owner can dynamically update his outsourced data, while users just have the read privilege. In this sense, multiple update operations are executed in a serial manner, including the access of the index switcher. However, if parallelization technique is used to optimize the tag generation and proof verification at the client side, then the access of the index switcher may be a bottleneck of performance. A possible way to solve this problem is to replicate [18], [19] the index switcher across multiple users. Since the correctness of the index switcher affects the auditing result, strong consistency [20] among these replicas should be guaranteed. We leave the implementation of such an optimized scheme as an important future extension, which can be combined into the design of a secure cloud storage system to cope with large scale data more robustly and efficiently.

4 Dispute Arbitration

4.1 Overview

As we have pointed out before, in the cloud environment, both clients and CSPs have the motive to cheat. In our scheme, the index switcher is used by the auditor to obtain tag indices for requested blocks at proof verification phase, thus the verification result relies on the correctness of the index switcher. However, the generation and update of index switcher are performed by the data owner only, it will potentially give a dishonest owner the opportunity of falsely accusing an honest CSP. In this sense, we must provide some mechanism to ensure the correctness of the index switcher and further the fairness of possible arbitration, so that no party can frame the other party without being detected.

A straightforward way is to let the arbitrator(TPAR) keep a copy of the index switcher. Since the change of the index switcher is caused by dynamic operations, the client can send necessary update information (i.e., operation type, operation position, new tag index) to the TPAR for each update operation. With these information, the arbitrator could re-construct the latest version of the index switcher, whose correctness decides the validity of later arbitration. However, such a solution costs $O(n)$ storage at the arbitrator side and needs the arbitrator to be involved in each update operation. Ideally, we want the TPAR only undertake the role of an arbitrator who involves only at dispute settlement, and maintains a constant storage for state information, i.e., public keys of the client and the CSP.

As an alternative, we employ the signature exchange idea in [12] to ensure the correctness of the index switcher. Specifically, we rely on both parties exchanging their signatures on the latest index switcher at each dynamic operation. To resist replay attacks, a sequence number indicating the update times is embedded in the signature.

A basic fact is that when the client initially uploads his data to the cloud, the cloud needs to run the **Commitment** to check the validity of outsourced blocks and their tags, and afterwards their signatures on the initial index switcher are exchanged. If this initial signature exchange fails, the client would not assume his data being successfully uploaded. On the other hand, the initial tag index sequence is the same as the block index sequence, that is, the index switcher can be denoted as $\{(i, i)\}_{1 \leq i \leq n}$. Hence, this step of signature exchange, according to our design, can be easily completed since the initial content of the index switcher is public to both parties, which is a basis for later signature exchanges. In this sense, our arbitration does not need the existence of a fair signature exchange protocol in [12]. Moreover, because the change of the index switcher is caused by data update operations, the CSP can re-construct the latest index switcher as long as necessary update information (i.e., op, k, t'_k in each update record) are sent to the CSP upon each update, which enables the CSP to check the client's signature and generate his own signature on the updated index switcher.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

Now, upon each data dynamic operation, besides verifying the updated blocks and tags, the CSP also checks client's signature on the updated index switcher. If succeeds, the CSP sends his signature on the updated index switcher to the client for storage. Then for each successful update, each party holds the other party's signature on the updated index switcher. Such a signature exchange implies an agreement has been reached on the new metadata by the two parties, which is necessary for later dispute resolution. Moreover, the signature is generated on the concatenation of a sequence number seq and the index switcher Ω, where seq is a monotonically increasing integer that is incremented by one each time.

Generally, a dispute may be caused by the disagreement on the proof (including an updated block m'_k and its tag σ'_k in an update request), or on the exchanged signature on the index switcher. According to the time a dispute occurs, we divide the arbitration occasion into three cases.

- Case 1: The dispute occurs when an auditor claims a failure of proof verification during an auditing.
- Case 2: The dispute occurs when the CSP receives an invalid update request up_req from the client.
- Case 3: The dispute occurs when the client receives an invalid response to up_req from the CSP.

Case 1 only involves the disagreement of proof verification, it occurs after a previous successful update where an agreement on the index switcher has been made. While case 2 and case 3 occur before the completion of the current round of update and signature exchange, so the TPAR should be engaged in the protocol to arbitrate on the dispute and help to finish the update and signature exchange.

4.2 Arbitration on Integrity Proof

Let $Sig_c = Sig_{sk_c}(seq, \Omega)$ and $Sig_s = Sig_{sk_s}(seq, \Omega)$ denote client and CSP's signatures on the index switcher in the last successful update, where seq refers to the latest sequence number. When a successful signature exchange completes, the client has the signature Sig_s of the server, and the server has the signature Sig_c of the client. During the arbitration, seq_c and seq_s denote the sequence number sent by the client and the CSP, Ω_c and Ω_s denote the index switcher sent by them, respectively. We assume the public key of each party is in some trusted PKI, hence it can be easily obtained by the other party (including the TPAR). And throughout our protocols, we assume the messages transmitted among three parties are in an authenticated secure channel.

We first describe the arbitration protocol of case 1, where the dispute only involves proof disagreement. When the client finds a failure of proof verification during an auditing, he contacts the TPAR to launch an arbitration. Since verifying proof validity needs to access the index switcher to get tag indices of challenged blocks, and verifying signatures also needs the index switcher, it is necessary for each party to send the TPAR the latest index switcher he has kept, along with the signature (on the index switcher) signed by the other party. The arbitration protocol proceeds as follows.

1) The TPAR requests $\{seq_c, \Omega_c, Sig_s\}$ from the client. Then he checks the signature Sig_s of the CSP. If it is invalid, the TPAR may punish the client for misbehaving; otherwise the TPAR proceeds.
2) The TPAR requests $\{seq_s, \Omega_s, Sig_c\}$ from the CSP. Then he checks the signature Sig_c of the client. If the signature does not verify correctly, the TPAR may punish the CSP for misbehaving; otherwise the TPAR proceeds.
3) If $seq_c = seq_s$, then the TPAR requests from the client the challenged set Q that causes dispute on proof verification and retransmit it to the CSP to run the auditing scheme. The CSP computes the proof according to **ProofGen** and returns it to the TPAR for verification. The TPAR checks the proof according to **ProofVerify** using the verified index switcher.
4) If there is a mismatch in seq_c and seq_s. The TPAR can be sure that the party who gives a smaller sequence number is performing an replay attack, he may punish the cheating party. Specifically, if $seq_c > seq_s$, the client is cheating by replaying an old signature from the CSP; if $seq_s > seq_c$, the CSP is cheating by replaying an old signature from the client.

The security of this protocol relies on the security of the signature scheme used to sign the index switcher, that is, each party has only negligible probability to forge a signature signed with the other party's private key. Therefore, what should be prevented in the protocol is possible replay attacks launched by a malicious party. As we have included a sequence number in the exchaned signature for each update, we can check whether a replay attack is launched or not by sequence number match. If both signatures verify correctly and their sequence numbers match ($seq_c = seq_s$) then we have $\Omega_c = \Omega_s$. Due to the initial signature exchange on $(0, \Omega_0)$ in **TagGen** and **Commitment**, there is at least one round of successful signature exchange before a conflict on proof verification occurs.

4.3 Arbitration on Dynamic Update

Case 2 and case 3 involves the failure of a signature exchange in the current round of update, so it is necessary for the TPAR to help to complete the update and signature exchange. To accomplish this, the successfully exchanged signatures in the previous round should be verified to proceed the current round.

The first two steps of the protocol is the same as that of the arbitration protocol on integrity proof, the TPAR requests $\{seq_c, \Omega_c, Sig_s\}$ from the client and $\{seq_s, \Omega_s, Sig_c\}$ from the CSP. If the TPAR finds any invalid signature, he punishes the corresponding party. According to the result of sequence number comparison (seq_c and seq_s), we divide the protocol into two situations.

The sequence numbers match($seq_c = seq_s$).

1) The TPAR requests the update record $\{seq_c + 1, op, k, t'_k, m'_k, \sigma'_k, Q'\}$ from the client.
2) For block modification and insertion, the TPAR verifies the correctness of (t'_k, m'_k, σ'_k) by verifying $e(\sigma'_k, g) \stackrel{?}{=} e(H(t'_k) \cdot u^{m'_k}, v)$. If fails, the TPAR may punish the client for cheating; otherwise, the TPAR

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

is convinced that the updated block and its tag are consistent with each other. For block deletion, this step can be omitted.

3) The TPAR transmits $\{seq_c + 1, op, k, t'_k, m'_k, \sigma'_k, Q'\}$ to the CSP, and requests (μ', σ') on the small challenge set Q' from the CSP. Then he verifies the validity of μ' and σ' according to algorithm **ProofVerify**. If fails, the TPAR may punish the CSP for denying the update; otherwise, the TPAR proceeds.
4) The TPAR updates the index switcher to Ω', then he requests and verifies new signatures $Sig'_c = Sig_{sk_c}(seq_c + 1, \Omega')$ and $Sig'_s = Sig_{sk_s}(seq_s + 1, \Omega')$ from both parties. The TPAR may punish the party who sends an invalid signature. If both signatures verify, the TPAR forwards Sig'_c to the CSP, and Sig'_s to the client.

Now the update has been successfully completed with the help of the TPAR, and new signatures on the updated index switcher are successfully exchanged. The client and the CSP can proceed in the next round of update.

The sequence numbers mismatch ($seq_c \neq seq_s$).

1) $seq_c < seq_s$. The server is cheating by replaying an old signature from the client.
2) $seq_c > seq_s + 1$. The client is cheating by replaying an old signature from the CSP.
3) $seq_c = seq_s + 1$. This occurs when the CSP receives the client's update request and refuses to update and send his signature to the client. There are three possibilities here. (i) The update record from the client is invalid (inconsistent block-tag pair but with valid signature on the updated index switcher), so the CSP refuses to update and contacts the TPAR for arbitration. (ii) The update record from the client is valid, but the CSP responds with invalid signature, so the client contacts the TPAR for arbitration. (iii) The update record from the client is valid, but the CSP maliciously denies the update, so the client contacts the TPAR for arbitration.

For the denial-of-update case($seq_c = seq_s + 1$), it is difficult for the TPAR to decide which party is responsible for the update failure. Because each party can behave maliciously to the other party and behave friendly to the TPAR, e.g., the client can send an incorrect update record to the CSP in the current round and send a correct update record to the TPAR in the following arbitration. In this case, the TPAR simply runs the protocol of match situation ($seq_c = seq_s$) to finish the update and signature exchange in the current round, so that both parties can proceed with further rounds of auditing or update.

4.4 Discussion

In our arbitration protocol, each party needs to send his signature on the latest metadata (the sequence number and the index switcher) to the other party. Actually, letting just one party sign the index switcher and the other party only sign the sequence number is also feasible, e.g., the client's signature is $Sig_c = Sig_{sk_c}(seq_c, \Omega_c)$ and the CSP's signature is $Sig_s = Sig_{sk_s}(seq_s)$. Then during an arbitration, the client sends $\{seq_c, Sig_s\}$ to the TPAR and the CSP sends $\{seq_s, \Omega_s, Sig_c\}$ to the TPAR.

In our design, the index switcher contains block-tag index pairs for all blocks, whose size is linear to the number of blocks. To achieve stateless arbitration at the TPAR, during an arbitration, each party has to send his version of the index switcher to the TPAR for signature verification, which brings $O(n)$ communication overhead for a dispute arbitration. Fortunately, although both parties have the potential possibility to misbehave, we still can assume such disputes are occasionally, or at least not frequently. After all, each party has some basic trust toward the other party, otherwise they can not cooperate together to store and manage client's outsourced data. In this sense, the arbitration protocol is to arbitrate possible disputes, the $O(n)$ communication may not be so serious a problem in terms of the service-oriented characteristic of cloud storage. On the other hand, the cheating party should be severely punished to reduce the possibilities of future misbehavior.

5 Security Analysis

In this section, we prove the security of our auditing scheme and arbitration protocol by proving the following two theorems. Theorem 1 adapts similar definitions of correctness and soundness in [5]: (i) The scheme is correct if the verification algorithm accepts when interacting with the valid prover who returns a valid response. (ii) The scheme is sound if any cheating prover that convinces the verification algorithm that it is storing a file is actually storing that file. Theorem 2 proves the fairness of our arbitration protocols by proving that both the owner and the CSP have negligible probability to frame the other party in the protocol.

Theorem 1. *If the two signature schemes used for block tags and index switcher are existentially unforgeable and the computational Diffie-Hellman problem is hard in bilinear groups, then no adversary against the soundness of our scheme could cause the verifier to accept in an auditing protocol instance with non-negligible probability, except by responding with correct proof.*

Proof: Since each tag index is a globally unique integer, without loss of generality, the index mapping from a block index $i \in \{1, 2, ..., n\}$ to its tag index $t_i \in \mathbb{N}$ can be regarded as an injective function. Thus, the hash query of block index in [5] is similar to that of tag index in our scheme. We now prove theorem 1 in the random oracle model by using a series of games defined in [5]. 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. Game 2 is the same as Game 1, except that the challenger keeps a list of its responses to queries from the adversary.

Suppose $Q = (i, v_i)_{i \in I}$ is the query that causes the adversary wins the game. Let $P = \{\mu, \sigma, Sig_{sk_s}(seq, \Omega)\}$ be the expected response from an honest prover, which satisfies the following equation $e(\sigma, g) = e((\prod_{i \in I} H(t_i)^{v_i}) \cdot u_\mu, v)$. Since the index switcher Ω is actually kept by both the client and the CSP, the verification will fail if the adversary gives an invalid signature. Thus, the signature on the

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

index switcher of the forged proof should be the same with that of the expected proof. Assume the adversary's response is $P' = \{\mu', \sigma', Sig_{sk_s}(seq, \Omega)\}$, which satisfies $e(\sigma', g) = e((\prod_{i \in I} H(t_i)^{v_i}) \cdot u_{\mu'}, v)$. 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 sets $u = g^{\beta} \cdot h^{\gamma}$. Then the simulator answers hash queries as $H(t_i) = g^{r_i}/(g^{\beta \cdot m_i} \cdot h^{\gamma \cdot m_i})$, and answers signing queries as $\sigma_i = (g^{r_i})^{\alpha}$. Finally, the adversary outputs $\pi' = (\mu', \sigma')$. From the above two equations, we can obtain

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

From this equation, we have

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

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 $\gamma\Delta\mu = 0 \bmod p$ 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 $\{\mu', \sigma', Sig_{sk_s}(seq, \Omega)\}$ to query is not equal to the expected response $\{\mu, \sigma, Sig_{sk_s}(seq, \Omega)\}$, the challenger declares failure. And in Game 2, we have proved that $\sigma' = \sigma$, and the signature on the index switcher is the same, so it is only the values μ' and μ can differ. Define $\Delta\mu = \mu' - \mu$, we have

$$\begin{aligned} e(\sigma', g) &= e((\prod_{i \in I} H(t_i)^{v_i}) \cdot u^{\mu'}, v) \\ &= e(\sigma, g) \\ &= e((\prod_{i \in I} H(t_i)^{v_i}) \cdot u^{\mu}, v) \end{aligned}$$

From this equation, we have $u^{\mu'} = u^{\mu}$ and $1 = u^{\Delta\mu}$. So we have $\Delta\mu = 0 \bmod p$. As analyzed above, there is only negligible difference between the adversary's success probabilities in these games. This completes the proof. □

Theorem 2. *Assume the signature scheme used for signing the index switcher is secure against existential forgery, and the default auditing scheme is secure, then the arbitration protocols provide secure and fair arbitration, that is, neither the client nor the CSP can succeed in framing the other party with non-negligible probability.*

Proof: Firstly, we prove the correctness of our arbitration protocol. Let $Sig_c(i, \Omega_i)$ and $Sig_s(i, \Omega_i)$ denote signatures (on the index switcher) signed by the client and the CSP, where Ω_i denote the i-th version of index switcher.

Recall that when the client uploads his data to the cloud, there is an initial signature exchange between the client and the CSP, where the sequence number is initialized to 0 and the index switcher is $\Omega_0 = \{(i, t_i = i)\}_{1 \leq i \leq n}$. Moreover, the uploaded blocks and tags have to be verified by the CSP using **ProofVerify** to ensure the consistency between blocks and tags. Since our scheme is public verifiable, and the initial content of the index switcher is also public, so even the CSP or the client misbehave in this step, it can be arbitrated by the TPAR, and finally the initial signature exchange can be successfully finished. After the initial exchange, the client has $Sig_s(0, \Omega_0)$ and the CSP has $Sig_c(0, \Omega_0)$. On the other hand, during each update, the update related information (operation type, operation position, new tag index) are sent to the CSP. Hence, the CSP is aware of the update of the index switcher and can re-construct it upon each update, which enable the CSP to check the validity of client's signature (on the index switcher).

We proceed by including several rounds of update and signature exchange. Assume the client has successfully finished $i (i \geq 2)$ updates so far. Without loss of generality, the client successively has $Sig_s(0, \Omega_0), Sig_s(1, \Omega_1), \ldots, Sig_s(i, \Omega_i)$, and the CSP successively has $Sig_c(0, \Omega_0), Sig_c(1, \Omega_1), \ldots, Sig_c(i, \Omega_i)$.

Now we analyze the situation where the signature exchange can not be normally finished. Normally, when the client performs the next update, he increments the sequence number and updates the index switcher to Ω_{i+1}, and sends an update record $\{i+1, op, k, t'_k, m'_k, \sigma'_k, Q', Sig_c(i+1, \Omega_{i+1})\}$ to the CSP. The CSP replies with $(\mu', \sigma', Sig_s(i+1, \Omega_{i+1}))$. Note in the update request, the correctness of $Sig_c(i+1, \Omega_{i+1})$ depends on the correctness of $(i+1, op, k, t'_k)$ and the index switcher Ω_i in the last successful update. And the correctness of σ'_k depends on the correctness of t'_k and m'_k. Due to the potential misbehavior of both parties, there are five possibilities here.

1) The client sends an invalid update request containing inconsistent m_k, σ_k and invalid $Sig_c(i+1, \Omega_{i+1})$ to the CSP. The CSP certainly denies the update and contacts the TPAR for update arbitration. In this case, when the TPAR requests the latest signatures from both parties, we have $seq_c = seq_s = i$.
2) The client sends an invalid update request containing inconsistent m_k, σ_k and valid $Sig_c(i+1, \Omega_{i+1})$ to the CSP. The CSP denies the update and contacts the TPAR for update arbitration. In this case, when the TPAR requests the latest signatures from both parties, we have $seq_c = seq_s + 1$.
3) The client sends a valid update request to the C-SP but the CSP replies with an invalid signature $Sig_s(i+1, \Omega_{i+1})$ (possibly with inconsistent μ' and σ'). The client contacts the TPAR for update arbitration. In this case, when the TPAR requests latest signatures from both parties, we have $seq_c = seq_s + 1$.
4) The client sends a valid update request to the CSP, the CSP replies with a valid signature $Sig_s(i+1, \Omega_{i+1})$ and inconsistent μ' and σ'. The client contacts the TPAR for proof arbitration. In this case, when the TPAR requests latest signatures from both parties, we have $seq_c = seq_s$.
5) The client sends a valid update record to the CSP, but the CSP denies the update. This case is similar to the third case, the client contacts the TPAR for arbitration. And when the TPAR requests latest signatures from both parties, we have $seq_c = seq_s + 1$.

According to the protocol of update arbitration, we can

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

see that even the client or the CSP misbehave in the current round of update, the TPAR still can help to finish the $(i+1)$-th round of update and signature exchange. And after the arbitration, the CSP has $Sig_c(i+1, \Omega_{i+1})$ and the client has $Sig_s(i+1, \Omega_{i+1})$. This completes the proof of correctness of the arbitration protocol.

Secondly, we prove the fairness of the arbitration protocol. If a malicious party wants to frame an honest party, he has to forge a valid signature of the honest party using a sequence number larger than the agreed one in the last successful update. Assume the sequence number and the index switcher in the last successful round of update are seq and Ω. The client has $Sig_s(seq, \Omega)$ and the CSP has $Sig_c(seq, \Omega)$. For the client, if he intends to frame the CSP, then he has to forge a valid signature $Sig_s(seq', \Omega')$, where $seq' > seq$ and $\Omega' \neq \Omega$. For the CSP, if he intends to frame the client, then he has to forge a valid signature $Sig_c(seq'', \Omega'')$, where $seq'' > seq + 1$ and $\Omega'' \neq \Omega$. Both cases contradict the existential unforgeability of the signature scheme used for signing the index switcher. Therefore, if the malicious party can forge such a signature with non-negligible probability, then he can break the security of the signature scheme (used for signing the index switcher) with non-negligible probability. This completes the proof of fairness. □

6 PERFORMANCE EVALUATION

Our scheme is implemented using C language on a Linux system equipped with a 4-Core Intel Xeon processor running at 2.4GHz, 4GB RAM and a 7200 RPM 2TB drive. Algorithms are implemented using the Pairing-Based Cryptographic (PBC) library 0.5.11 and the crypto library OpenSSL 1.0.0. For security parameters, we choose the curve group with a 160-bit group order and the size of modulus is 1024 bits. Our scheme provides probabilistic proof as [4]: if t fraction of the file is corrupted, by challenging a constant c blocks of the file, the auditor can detect the data corruption behavior at least with probability $p = 1 - (1-t)^c$. We choose $c = 460$, thus the detection probability is about 99%. The integrity proof is of constant size as in [5]. While most authenticated structure based schemes [6], [9], [11] need to send auxiliary authentication information to the auditor, which leads to linear communication overhead. The size of the test data is 10 GB, and the block size of fragmentation varies from 2 KB to 1 MB. All results are on the average of 10 trials.

We measure the performance of our auditing scheme from three aspects: tag generation time, proof generation time and proof verification time. For data dynamic update and dispute arbitration, we test the update overhead by inserting, deleting and modifying 100 blocks and tags. In addition, we test the cost of signature computation and verification with the index switcher containing different number of index pairs (from 5×2^{11} to 5×2^{20}), and the shifting overhead of index pairs caused by block insertion and deletion.

6.1 Tag computation

For each block size(from 2KB to 1MB), we firstly fragment the data file into multiple blocks and generate their tags,

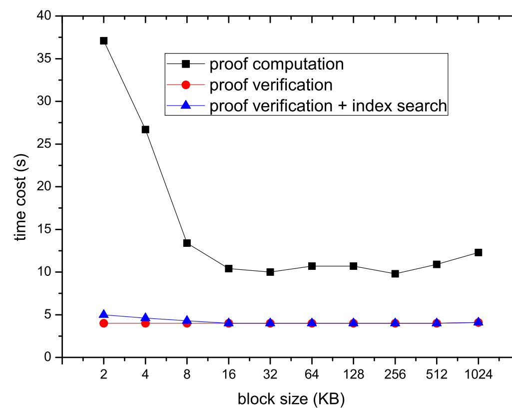

Fig. 3. Cost of proof generation and verification.

then we store these blocks and tags into small files, e.g., when block size is 2KB, the original data file corresponds to 5×2^{20} block files and 5×2^{20} tag files. And we allocate a storage node to store the block files and their tags for each block size.

According to our tag computation formula $\sigma_i = (H(t_i) \cdot u^{m_i})^{\alpha}$, we can see that each tag generation involves two exponentiations, first with m_i then with user's secret key α, which are the main overhead for computing a tag. For same size data, bigger block fragmentation means less tag files and therefore less exponentiations. As illustrated in Table 1, the cost of tag computation for the same file decreases almost linearly with the increase of block size. On the other hand, we test the overhead of tag generation for 1-GB data and 10-GB data, and we find for each block size, the tag generation overhead of the 10-GB file is nearly 10 times of that of the 1-GB file, which also demonstrates that the number of exponentiations dominate the tag generation overhead.

When block size is small, the cost is really heavy, e.g., when the block size is 2KB, the tag generation overhead for 10-GB file is nearly 18 hours. The reason is that small fragmentation means a large number of exponentiations. But when the block size exceeds 32KB, the tag generation time decreases fast, e.g., when block size is 512 KB, it costs 52 seconds and 528 seconds for 1-GB data and 10-GB data, respectively. Therefore it is better to choose a big block size for data fragmentation to audit large-size data files. Fortunately, for any data file to be outsourced to the cloud, tag generation phase happens only once.

6.2 Proof computation and verification.

Proof generation needs to access $c = 460$ randomly selected block files and corresponding tag files to compute $\mu = \sum_{i \in I} v_i \cdot m_i$ and $\sigma = \prod_{i \in I} \sigma_i^{v_i}$, where computing μ involves c multiplications and c additions on $\mathbb{Z}_p$, and computing σ involves c exponentiations and c multiplications on group $\mathbb{G}_1$. In addition to these computations, there are also two other factors having influence on the overhead of proof generation: one is the search time for a specific file in a directory, the other is the I/O cost of reading 460 block

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

TABLE 1
Tag generation time cost with different block size

Block size	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1024KB
1GB data (s)	6284	3146	1570	791	409	220	122	75	52	38
10GB data (s)	63489	35445	15734	8096	4206	2127	1167	715	528	408

files and 460 tag files from a storage node. The I/O cost of reading 460 block files is linear to the block size, and reading 460 tag files incur a constant overhead, since each tag file is fixed (128 bytes). Furthermore, the sampling strategy of the auditing challenge makes the access of block and tag files randomly, so the principle of locality in I/O access does not work well.

Fig.3 illustrates the cost of proof generation and verification in our scheme. It can be observed that when the block size is small, i.e., less than 8 KB, the cost decreases fast. This is mainly because the search time for a specific block or tag file is large at small block size. For the same file, small block size means a large number of block and tag files, e.g., 2 KB fragmentation corresponds to 5×2^{21} block and tag files totally. In contrast, the I/O cost of reading 460 block files and 460 tag files are relative small, e.g., for 2KB block size, the I/O overhead is about reading 978 KB data in total. With the growth of the block size, on one hand, the search time for a specific file decreases, and on the other hand, the I/O overhead for reading 460 block files increases. These two reasons explain why the cost curve remains relatively stable and achieves an optimal value in the range of 16 KB to 256 KB. But when the block size exceeds 256 KB, the cost begins to increase, the reason is that the I/O overhead has a greater influence (e.g., 460 MB for 460 blocks with block size being 1024KB) than the file search time.

The overhead of proof verification can be divided into two parts: the search cost of tag indices for challenged blocks, and the cost of verifying the proof validity according to equation (1). In implementation, for each block size, we write all the index pairs into a file for storage, thus the fragmentation size decides the number of index pairs in the index switcher. To optimize the search time for tag indices, we sort the indices of challenged blocks before searching. The bottom two curves in Fig. 3 show the proof verification overhead with and without the index search cost, we can see they are very close to each other, except that when block size is less than 8 KB, the index search cost is non-negligible, which takes about 20% and 13% in total overhead for 2 KB and 4 KB respectively. But when block size exceeds 8 KB, both curves remain steady, because then the main overhead are the bilinear pairing operations and exponentiations involved in proof verification, which are independent of the block size.

6.3 Update and arbitration

For data dynamics, we test the overhead of inserting, deleting and modifying 100 blocks and corresponding tags, as illustrated in Fig. 4. We find the curves of insertion and modification are very close to each other. This is because both inserting and modifying a block needs to compute a new tag, then write the updated block and tag to the storage

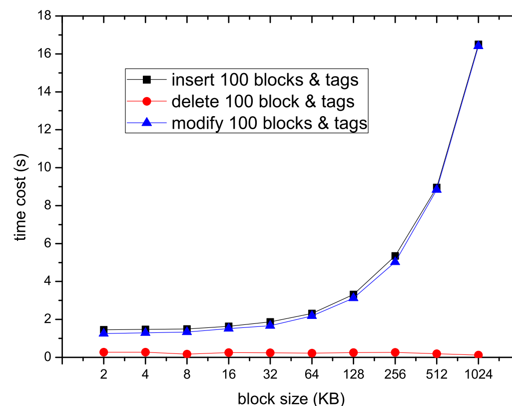

Fig. 4. Cost of block and tag update.

device. When the block size increases, the I/O overhead also increases. This explains why inserting 100 2-KB blocks costs only 1.5 seconds while inserting 100 1024-KB blocks costs 16.5 seconds. The deletion curve in Fig. 4 remains steady, with the cost being about 0.2 seconds.

On the other hand, data update and dispute arbitration involve the computation and verification of the signature on the index switcher Ω, whose size is linear to the number of block-tag index pairs. We allocate 4 bytes for each block index and 8 bytes for each tag index, thus the size of the index switcher is $12n$ bytes, where n denotes the number of blocks. For same amount of data, the value n is inversely proportional to the block size used for fragmentation. As illustrated in Table 2, for a 10-GB data file, the size of the index switcher is 60MB when block size is 2 KB, and decreases to 120KB when block size is 1024 KB.

In implementation, we write the content of the index switcher into a file for storage. Thus, computing or verifying the signature on the index switcher needs to read its content from the file. Table 2 illustrates the efficiency of computing and verifying the signature on the index switcher, with the number of index pairs varying from 5×2^{11} to 5×2^{20} . When the index switcher contains 5×2^{20} index pairs, both signature computation and verification cost about 200 miliseconds, and this cost almost decreases linearly with the reduction of index pairs. Moreover, dynamic update (block insertion and deletion) will cause index pairs of all subsequent blocks (after the operation position) to be shifted backward or forward a position. We test the overhead of updating index switcher by shifting all index pairs through insertion and deletion, e.g., shifting 5×2^{20} index pairs incurs a cost about 46.5 miliseconds. Even we include the overhead of reading all index pairs from the file and the

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

TABLE 2
Time cost of signature (on the index switcher) generation and verification

Block size	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1024KB
Number of index pairs	5×2^{20}	5×2^{19}	5×2^{18}	5×2^{17}	5×2^{16}	5×2^{15}	5×2^{14}	5×2^{13}	5×2^{12}	5×2^{11}
Size of index switcher	60 MB	30 MB	15 MB	7.5 MB	3.75 MB	1.875 MB	960 KB	480 KB	240 KB	120 KB
Read index switcher (ms)	285.0	155.1	68.7	18.3	9.5	4.0	0.2	0.09	0.05	0.03
Signing index switcher (ms)	201.1	100.8	50.9	26.0	13.5	7.2	4.1	2.6	1.8	1.4
Signature verification (ms)	198.8	99.5	50.1	24.8	12.5	6.3	3.2	1.7	0.9	0.5
Shifting all index pairs (ms)	46.5	24.3	11.7	6.2	3.2	1.8	1.0	0.5	0.2	0.1

overhead of shifting index pairs, for 2-KB fragmentation, the overhead of signing and updating the index switcher is about 533 seconds, which shows our additional overhead of data dynamics is acceptable. Further, the evaluation also shows the scalability of our scheme, with the block size being 512 KB and the number of index pairs being 5×2^{20}, the total auditing data can be 2.5 TB.

7 Related Work

Remote integrity check could be sourced to memory check schemes [21], [22] that aim to verify read and write operations to a remote memory. Recently, many auditing schemes [1], [2], [23], [24], [25], [26] have been proposed around checking the integrity of outsourced data.

Deswarte et al. [1] and Filho et al. [2] use RSA-based hash functions to check a file's integrity. Although their approaches allow unlimited auditing times and offer constant communication complexity, their computation overhead is too expensive because their schemes have to treat the whole file as an exponent. Opera et al. [23] propose a scheme based on tweakable block cipher to detect unauthorized modification of data blocks, but verification needs to retrieve the entire file, thus the overhead of data file access and communication are linear with the file size. Schwarz et al. [24] propose an algebraic signature based scheme, which has the property that the signature of the parity block equals to the parity of the signatures on the data blocks. However, the security of their scheme is not proved. Sebe et al. [26] provide an integrity checking scheme based on the Diffie-Hellman problem. They fragment the data file into blocks of the same size and fingerprint each data block with an RSA-based hash function. But the scheme only works when the block size is much larger than the RSA modulus N, and it still needs to access the whole data file. Shah et al. [7], [27] propose a privacy-preserving auditing protocol that allows a third party auditor to verify the integrity of remotely stored data and assist to extract the original data to the user. As their scheme need firstly encrypt the data and pre-compute a number of hashes, the number of auditing times is limited and it only works on encrypted data. Furthermore, when these hash values are used up, the auditor has to re-generate a list of new hash values, which leads to extremely high communication overhead.

From above analysis, it can be seen that earlier schemes usually generate a deterministic proof by accessing the whole data file, thus their efficiency is limited due to the high computation overhead. To address this problem, later schemes tend to generate a probabilistic proof by accessing part of the date file. Jules et al. [3], [28] propose a proofs of retrievability (PoR) model, where spot-checking and error-correcting code are used to guarantee the possession and retrievability of remote stored data. However, PoR can only be applied to encrypted data, and the number of auditing times is a fixed priori due to the fact that sentinels embedded in the encrypted data could not be reused once revealed. Dodis el al. identify several other variants of PoR in [29]. Ateniese et al. [4] are the first to put forward the notion of public verifiability in their provable data possession (PDP) scheme, where the auditing tasks can be delegated to a third-party auditor. In PDP, they propose to randomly sample a few data blocks to obtain a probabilistic proof, which greatly reduces the computation overhead. Moreover, PDP scheme allows unlimited number of auditing. Shacham et al. [5] design an improved PoR scheme and provide strict security proofs in the security model defined in [3], they use homomorphic authenticators and provable secure BLS signatures [30] to achieve public verifiability, which is not provided in Jules' main PoR scheme. Some other schemes [7], [14], [31] with public auditability aim to provide privacy protection against information leakage toward a third-party auditor in the process of integrity auditing.

However, all above-mentioned schemes are designed for static data only, direct extension of these schemes to support data dynamics may suffer from security problems, as analyzed in [14]. But in cloud environment, remotely stored data may not only be read but also be updated by users, which is a common requirement. In this sense, schemes can only audit static data is insufficient and lacks of practicability.

To support data dynamics in auditing schemes, Ateniese et al. [32] propose a dynamic version of their original PDP scheme using symmetric encryption, however, the number of auditing times is limited and fully block insertion is not supported (only append-type insertion is supported). Erway et al. [9] firstly propose to construct a fully dynamic provable data possession (DPDP) scheme. To eliminate the index limitation of tag computation in original PDP scheme and avoid tag re-computation brought by data dynamics, they use the rank of a skip list node (similar to block index) to uniquely differentiate among blocks and authenticate the tag information of challenged blocks before proof verification. However, the skip list in essence is an authenticated structure used to test set-membership for a set of elements. To prove the membership of a specific node, a verification path from the start node to the queried node must be provided, its communication cost is linear to the number

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

of challenged blocks. Moreover, there's no explicit implementation of public verifiability given for their scheme.

Qian Wang et al. [6] combine BLS signature based homomorphic authenticator with Merkle hash tree to provide both public auditability and fully dynamic operations support. Specifically, their scheme constructs a Merkle hash tree, stores the hashes of tags in the leaf nodes and recursively computes the root and signs it, which is used to authenticate the tags of challenged blocks. Furthermore, they eliminate the index limitation in tag computation by using $H(m_i)$ to replace $H(name||i)$ in [5], which requires blocks to be different with each other. However, such a requirement on data blocks is not appropriate since the probability of block resemblance increases when block size decreases. In addition, due to the authentication of challenged blocks with a Merkle Hash Tree [33], the communication cost of their scheme is also linear to the number of requested blocks. Zhu et al. [10], [34] use index-hash table to construct their dynamic auditing scheme based on zero-knowledge proof, which is similar to our index switcher in terms of index differentiation and avoidance of tag re-computation. But their design mainly focuses on data dynamics support, while our scheme goes further by achieving dynamic operations support and fair arbitration together.

Recently, providing fairness and arbitration in auditing schemes has become an important trend, which extends and improves the threat model in early schemes to achieve a higher level of security insurance. Zheng et al. [11] construct a fair and dynamic auditing scheme to prevent a dishonest client accusing an honest CSP. But their scheme only realizes private auditing, and is difficult to be extended to support public auditing. Kupcu [12] proposes a framework on top of Erway's DPDP scheme [9], where the author designs arbitration protocols on the basis of fair signature exchange protocols in [13]. Moreover, the author goes further by designing arbitration protocols with automated payments through the use of electronic cash. Compared to these schemes, our work is the first to combine public verifiability, data dynamics support and dispute arbitration simultaneously.

Other extensions to both PDPs and PoRs are given in [35], [36], [37], [38], [39]. Chen et al. [37] introduce a mechanism for data integrity auditing under the multi-server scenario, where data are encoded with network code. Curtmola et al. [35] propose to ensure data possession of multiple replicas across the distributed storage scenario. They also integrate forward error-correcting codes into PDP to provide robust data possession in [36]. Wang et al. [39] utilize the idea of proxy re-signatures to provide efficient user revocations, where the shared data are signed by a group of users. And in [16], [38], they exploit ring signatures to protect the identity-privacy of signers from being known by public verifiers during the auditing.

8 Conclusion

The aim of this paper is to provide an integrity auditing scheme with public verifiability, efficient data dynamics and fair disputes arbitration. To eliminate the limitation of index usage in tag computation and efficiently support data dynamics, we differentiate between block indices and tag indices, and devise an index switcher to keep block-tag index mapping to avoid tag re-computation caused by block update operations, which incurs limited additional overhead, as shown in our performance evaluation. Meanwhile, since both clients and the CSP potentially may misbehave during auditing and data update, we extend the existing threat model in current research to provide fair arbitration for solving disputes between clients and the CSP, which is of vital significance for the deployment and promotion of auditing schemes in the cloud environment. We achieve this by designing arbitration protocols based on the idea of exchanging metadata signatures upon each update operation. Our experiments demonstrate the efficiency of our proposed scheme, whose overhead for dynamic update and dispute arbitration are reasonable.

Acknowledgments

Firstly, the authors would like to thank the anonymous referees for their reviews and insightful 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] Y. Deswarte, J.-J. Quisquater, and A. Saïdane, "Remote integrity checking," in *Proc. 5th Working Conf. Integrity and Intl Control in Information Systems*, 2004, pp. 1–11.

[2] D. L. Gazzoni Filho and P. S. L. M. Barreto, "Demonstrating data possession and uncheatable data transfer." *IACR Cryptology ePrint Archive, Report 2006/150*, 2006.

[3] A. Juels and B. S. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *Proc. 14th ACM Conf. Computer and Comm. Security (CCS07)*, 2007, pp. 584–597.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. 14th ACM Conf. Computer and Comm. Security (CCS07)*, 2007, pp. 598–609.

[5] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. 14th Intl Conf. Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT 08)*, 2008, pp. 90–107.

[6] Q. Wang, C. Wang, J. Li, K. Ren, and 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.

[7] M. A. Shah, R. Swaminathan, and M. Baker, "Privacy-preserving audit and extraction of digital contents." *IACR Cryptology ePrint Archive, Report 2008/186*, 2008.

[8] C. Wang, K. Ren, W. Lou, and J. Li, "Toward publicly auditable secure cloud data storage services," *Network, IEEE*, vol. 24, no. 4, pp. 19–24, 2010.

[9] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proc. 16th ACM Conf. Computer and Comm. Security (CCS 09)*, 2009, pp. 213–222.

[10] Y. Zhu, H. Wang, Z. Hu, G.-J. Ahn, H. Hu, and 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.

[11] Q. Zheng and 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.

[12] A. Küpçü, "Official arbitration with secure cloud storage application," *The Computer Journal*, pp. 138–169, 2013.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2016.2525998, IEEE Transactions on Cloud Computing

[13] N. Asokan, V. Shoup, and M. Waidner, "Optimistic fair exchange of digital signatures," in *Proc. 17th Intl Conf. Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT98)*, 1998, pp. 591–606.

[14] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[15] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE Trans. Computers*, vol. 62, no. 2, pp. 362–375, 2013.

[16] B. Wang, B. Li, and H. Li, "Oruta: Privacy-preserving public auditing for shared data in the cloud," *IEEE Trans. Cloud Computing*, vol. 2, no. 1, pp. 43–56, 2014.

[17] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Proc. 22nd Intl Conf. Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT03)*, 2003, pp. 416–432.

[18] P. A. Bernstein and N. Goodman, "An algorithm for concurrency control and recovery in replicated distributed databases," *ACM Trans. Database Systems*, vol. 9, no. 4, pp. 596–615, 1984.

[19] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Low-overhead byzantine fault-tolerant storage," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007, pp. 73–86.

[20] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *ACM SIGMOD Record*, vol. 25, no. 2, 1996, pp. 173–182.

[21] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," *Algorithmica*, vol. 12, no. 2-3, pp. 225–244, 1994.

[22] M. Naor and G. N. Rothblum, "The complexity of online memory checking," in *Proc. 46th Ann. IEEE Symp. Foundations of Computer Science*, 2005, pp. 573–582.

[23] A. Oprea, M. K. Reiter, and K. Yang, "Space-efficient block storage integrity." in *Proc. 9th Network and Distributed System Security Symp. (NDSS '05)*, 2005.

[24] T. S. Schwarz and E. L. Miller, "Store, forget, and check: Using algebraic signatures to check remotely administered storage," in *Proc. IEEE Intl Conf. Distributed Computing Systems (ICDCS 06)*, 2006, pp. 12–12.

[25] E.-C. Chang and J. Xu, "Remote integrity check with dishonest storage server," in *Proc. 13th European Conf. Research in Computer Security (ESORICS 08)*, 2008, pp. 223–237.

[26] 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. Knowledge and Data Eng.*, vol. 20, no. 8, pp. 1034–1038, 2008.

[27] M. A. Shah, M. Baker, J. C. Mogul, R. Swaminathan *et al.*, "Auditing to keep online storage services honest." in *Proc. 11th USENIX Workshop Hot Topics in Operating Systems (HotOS 07)*, 2007, pp. 1–6.

[28] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proc. ACM Cloud Computing Security Workshop (CCSW 09)*, 2009, pp. 43–54.

[29] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in *Proc. Theory of cryptography(TCC '09)*, 2009, pp. 109–127.

[30] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Proc. 7th Intl Conf. Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT 01)*, 2001, pp. 514–532.

[31] Z. Hao, S. Zhong, and N. Yu, "A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability," *IEEE Trans. Knowledge and Data Eng.*, vol. 23, no. 9, pp. 1432–1437, 2011.

[32] G. Ateniese, R. Di Pietro, L. V. Mancini, and 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.

[33] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. IEEE Symp. Security and Privacy*, 1980, pp. 122–133.

[34] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2231–2244, 2012.

[35] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "Mr-pdp: Multiple-replica provable data possession," in *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS '08)*, 2008, pp. 411–420.

[36] R. Curtmola, O. Khan, and R. Burns, "Robust remote data checking," in *Proc. 4th ACM int'l Workshop on Storage Security and Survivability*, 2008, pp. 63–68.

[37] B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote data checking for network coding-based distributed storage systems," in *Proc. ACM Cloud Computing Security Workshop (CCSW 10)*, 2010, pp. 31–42.

[38] B. Wang, B. Li, and 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.

[39] ——, "Panda: Public auditing for shared data with efficient user revocation in the cloud," *IEEE Trans. Services Computing*, vol. 8, no. 1, pp. 92–106, 2013.

Hao Jin 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 PhD 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.

Hong Jiang 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 PhD 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.

Ke Zhou Ke Zhou received his PhD 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 .

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.