Dynamic Proofs of Retrievability for Coded Cloud Storage Systems

Zhengwei Ren, Lina Wang, Qian Wang, *Member, IEEE,* Rongwei Yu, and Ruyi Deng

Abstract—Cloud storage allows users to store their data in a remote server to get rid of expensive local storage and management costs and then access data of interest anytime anywhere. A number of solutions have been proposed to tackle the verification of remote data integrity and retrievability in cloud storage systems. Most of existing schemes, however, do not support efficient data dynamics and/or suffer from security vulnerabilities when involving dynamic data operations. In this paper, we propose an enhanced dynamic proof of retrievability scheme supporting public auditability and communication-efficient recovery from data corruptions. To this end, we split up the data into small data blocks and encode each data block individually using network coding before outsourcing so that i) an update inside any data block only affects a few codeword symbols and ii) communication-efficient data repair for a breakdown server can be achieved. To eliminate the communication overhead for small data corruptions within a server, each encoded data block is further encoded via erasure codes. Based on the encoded data blocks, we utilize range-based 2-3 tree (rb23Tree) to enforce the data sequence for dynamic operations, preventing the cloud service provider from manipulating data block to pass the integrity check in the dynamic scenario. We also analyze the effectiveness of the proposed construction in defending against pollution attacks during data recovery. Formal security analysis and extensive experimental evaluations are conducted, showing that the proposed scheme is practical for use in cloud storage systems.

Index Terms—Cloud storage, Data integrity, Data availability, Public audit, Data dynamics.

1 Introduction

Cloud computing has attracted more and more attentions in both academia and industry due to its appealing advantages such as on-demand self-service, location independent resource pooling and rapid resource elasticity, etc. [1]. As an indispensable part of cloud computing, cloud storage makes data storage a service in which data is outsourced to a cloud server maintained by a cloud provider. From the users' perspective, storing data remotely into the cloud, especially for IT enterprises, brings attracting benefits, e.g., avoidance of capital expenditure on hardware and software, relief of the burden for storage management, etc. [2].

Although cloud storage has numerous advantages, some challenging security issues have to be addressed for cloud storage to be widely accepted by users. A cloud server storing many users' data will be the preferred attack target and the data may be faced a broad range of threats [3]–[5]. Moreover, differently from traditional data storage, in cloud storage the data owner does not possess data physically after data is outsourced into the cloud service providers (CSPs) who are not fully trusted. For their own benefits, CSPs might discard portion of rarely accessed data to save storage space. Also, the CSPs may be tempted to hide the data corruptions caused by server hacks or Byzantine failures to maintain reputation [6], [7]. It has been recognized that the security issues such as data integrity and availability are the main obstacles for cloud storage to be successfully adopted.

To prevent cheating from the "semi-trusted" CSPs, a trivial solution is to remotely download the entire data and periodically check their integrity. The resulting large communication costs, however, renders the loss of benefits of cloud storage service. Recently, two effective solutions have been proposed under different models such as the "Provable Data Possession (PDP)" model [6] and the "Proof of Retrievability (PoR)" model [8]. Following these models, improved schemes [7], [9]–[17] have been proposed to support new features such as data dynamics, public auditability etc. The support of data dynamics should allow the data owners to modify, delete the existing data blocks and insert new blocks. The support of public auditability allows the data owner to delegate the checking tasks to a third party auditor (TPA) so as to save his own computational resources. Unfortunately, the existing solutions still have severe limitations, which makes them impractical for cloud storage systems. In particular, it has been shown that prior PoR solutions suffer from security vulnerabilities when involving dynamic data operations [12]. Thus, it is necessary to defend against this realistic attack. In addition, the existing PoR schemes usually assume data coding is done before outsourcing. However, a single bit update will affect a large fraction of data storage, which makes data updates extremely inefficient. Finally, although prior solutions argue original data can be retrieved using coding techniques even after corruption,

• *Z. Ren, L. Wang, Q. Wang, R. Yu and R. Deng are with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, and School of Computer, Wuhan University, Wuhan, China, 430072. E-mail: {zhengwei_ren lnawang}@163.com, willwq@msn.com, yrw123@126.com, rzw@whu.edu.cn.*

how to perform efficient data recovery on corrupted data blocks has not been explicitly discussed.

In this paper, we propose a new dynamic proof of retrievability scheme for coded cloud storage systems. To ensure a server stores the data blocks that it is supposed to store, we utilize the rb23Tree [15] to organize the encoded blocks in the leaf nodes to enforce data access sequence and support cheat-proof data dynamic operations. To support public auditability, we use the aggregated signature-based broadcast (ASBB) encryption scheme [18] to generate metadata tags of the encoded blocks. In order to support efficient dynamic data operations, we achieve high coding granularity by encoding at the data block level instead of the whole large data file, *i.e.*, we split up the data into small data blocks and encode each data block individually. In using this coding strategy, an update inside any data block only affects the current data block itself and its associated codeword symbols without having to update the whole large data file. To improve data reliability and availability, we exploit both within-server redundancy and cross-server redundancy to encode data blocks before outsourcing. Specifically, we use regenerating codes [19] to encode the original data file as the outer code, providing data decoding and data recovery. The use of regenerating codes can maintain the same data redundancy level and same storage requirement as in traditional erasure codes (e.g., RAID-6 [20]), but incurs less data recovery communication cost across multiple storage servers. In addition, each encoded block is further encoded via erasure codes as the inner code to save communication cost when a small amount of data corruption is detected inside a storage server. The contributions of this paper are summarized as follows:

- We propose a new dynamic proof of retrievability scheme for coded cloud storage systems. We use rb23Tree to authenticate the sampling data blocks to prevent the storage server from manipulating data to pass the integrity check after dynamic data operations. In addition, we adopt both the within-server redundancy and the cross-server redundancy architecture to improve data availability while achieving high communication efficiency in data recovery process.
- We give a detailed description of the data encoding, decoding and recovery procedures, which were lacking in most existing PoR based solutions. By utilizing the network coding as the outer code and the erasure codes as the inner code, communication-efficient and effective data recovery can be realized.
- We give a formal security analysis and an extensive experimental study to demonstrate the security and efficiency. We show the correctness, the soundness and the data retrievability of our proposed construction. We analyze and evaluate the system performance in terms of data recovery cost and the data auditability cost.

2 Related Work

Remote data integrity checks for public cloud storage have been investigated in various systems and security models [6]–[17], [21]–[27]. Considering the large size of the outsourced data and the owner's constrained resource capability, the cost to audit data integrity in the cloud environment could be formidable and expensive to the data owner. Therefore, it is preferable to allow an independent expertise-equipped TPA to check the data integrity on behalf of the data owners [11].

Ateniese et al. [6] was the first to introduce the "Provable Data Possession (PDP)" model and proposed an integrity verification scheme for static data using RSA-based homomorphic authenticators. At the same time, Juels et al. [8] proposed the "Proof of Retrievability (PoR)" model which is stronger than the PDP model in the sense that the system additionally guarantees the retrievability of outsourced data. Specifically, the authors proposed a spot-checking approach to guarantee possession of data files and employed error-correcting coding technologies to ensure the retrievability. A limitation of their scheme is that the number of challenges is constrained. Shacham et al. [10] utilized the homomorphic signatures in [28] to design an improved PoR scheme. Although the scheme supported public auditability of static data using publicly verifiable homomorphic authenticators, how to perform data recovery was not explicitly discussed. To achieve strong data retrievability, Bowers et al. [21] proposed a data coding structure achieving the within-server redundancy and cross-server redundancy. Chen et al. [22] and Chen et al. [26] constructed their remote data checking schemes based on network coding which can save the communication cost of data recovery compared with erasure codes. In particular, [26] considered the cross-server redundancy as [21] in a multiple-server setting, where the cross-server coding was done using network coding instead of erasure codes in [21]. Recently, Cao et al. [25] designed a secure cloud storage system using LT codes.

To support data dynamics, many remote data auditability schemes have been proposed on the basis of PDP and PoR models. Erway et al. [9] developed a PDP scheme with full data dynamics using skip list. Meantime, Wang et al. [7] proposed a scheme supporting public auditability and data dynamics using BLS based signatures and Merkle hash tree (MHT). Zhu et al. [12] presented a cooperative PDP model in a multi-CSPs setting and illustrated security vulnerabilities of data dynamics in existing data auditability schemes. Zheng et al. [15] presented a fair and data dynamics enabled PoR scheme based on ranged-based 2-3 tree. Unfortunately, the authors did not consider how to update the redundant encoded data and their scheme does not support public auditability. Stefanov et al. [24] considered dynamic PoR in a more complex setting where an additional trusted portal performs some operations on behalf of the client and caches updates for an extended

period of time. More recently, Cash et al. [27] proposed a solution providing PoR for dynamic storage in a single-server setting via oblivious RAM.

3 System Model and Security Requirement

3.1 Public Cloud Storage Auditing Architecture

A number of proposals [6], [7], [9]–[13] employ a tripartite architecture as illustrated in Fig.1. In this framework, there are three different entities including data owners, CSP and TPA. The data owner has a large number of files to be stored in the CSP who has significant storage space and computation resources to provide data storage service. In this model, TPA is a trusted third party auditor who checks the data integrity on behalf of the data owners periodically or upon requests.

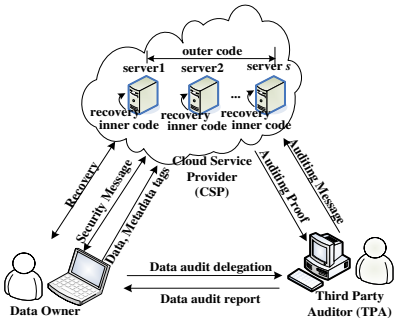


Fig. 1. Cloud Storage Public Auditing Architecture.

3.2 Public Auditing Model for Coded Cloud Storage

Our encoded cloud storage system consists of the following algorithms:

$KeyGen(1^{\kappa}) \rightarrow (pk, sk)$ is run by the data owner. It takes as input a security parameter κ and returns the public key *pk* and the secret key *sk*.

$TagGen(sk, F) \rightarrow (\phi_t, T_t)$ is run by the data owner. Given a data file *F*, the data owner first divides it into data blocks $m_i (1 \leq i \leq n)$. Then, the data owner encodes each block m_i into an encoded block. The encoded blocks are organized by rb23Trees $T_t (1 \leq t \leq s)$. The data owner computes the metadata tags ϕ_t for the encoded blocks generated from the same original block. Finally, coded data and metadata tags are outsourced to the cloud.

$ChalGen(c) \rightarrow chal$ is run by the TPA. The input is the total number of blocks and the output is a challenge $chal$ which contains sampling data blocks to be checked. The $chal$ is sent to the CSP as a request to check the integrity of the sample blocks.

$ProofGen(chal, T_t, \phi_t, F') \rightarrow P$ is run by the server. It takes as input the challenge $chal$, the rb23Tree T_t, the metadata tags ϕ_t and the encoded data file F'. It outputs a proof P to allow the TPA to verify the data integrity.

$VerifyProof(P) \rightarrow (True, False)$ is executed by the TPA. After receiving the proof P, TPA verifies the integrity of the sample blocks. It outputs $True$ if the checking results pass the integrity verifications for each server. Otherwise, it returns $False$.

$UpdateRequest() \rightarrow R$ is run by the data owner. It outputs an update request R which contains: the operator $op = M, I, D$, the index i. When the op is M or I, the request R indicates the new encoded blocks and the corresponding metadata tags.

$Update(R, F', \phi_t, T_t) \rightarrow v(root)'_t$ is executed by the server. It takes as input the encoded file F', the metadata tags ϕ_t , the rb23Tree T_t and the update request R and outputs the new root value $v(root)'_t$ of the new rb23Tree T'_t.

$Recovery(i) \rightarrow F''$ is run by the data owner. It takes as input the indices of the corrupted data. When the amount of corrupted data are less than a threshold, the data owner can recovery the corrupted data and generate a new encoded file F'' of the original data file F.

3.3 Adversarial Model and Security Requirements

Since data owners are out of physical control of their data, it is essential to convince the data owners that their data are securely kept without being modified by unauthorized parties. Usually, two basic security requirements are considered.

First, the data integrity should be verifiable and the data owners need to be convinced that their data has not been altered without their authorization. Since the data is physically controlled by CSPs who may be tempted to occasionally delete rarely accessed data to save the storage space or try to forge arguments of honest behaviors to cheat data owners for their own benefits. Many researches have focused on the remote data integrity check topic [6], [7], [10]–[16]. However, the support of secure and efficient data dynamic operations still remains a challenging problem. On the one hand, provable data possession on remote cloud servers should be guaranteed even if the data owner has dynamically updated their data. On the other hand, efficient data updates on encoded data should be achieved to ensure that an update on a single or a few data blocks will not affect the whole bunch of data.

Second, the data should be retrievable when data corruptions are within a threshold. To achieve data recovery, data replica and data encoding are the common methods. Although data replica is easy to be implemented and deployed, it consumes too much storage resource. There are many data encoding schemes which have different advantages and disadvantages in various application scenarios. As for cloud storage systems, network bandwidth bottleneck is one of the important parameters of system performance. Moreover, the encoding of outsourced data should support efficient data dynamics to avoid the decoding and re-encoding of the entire data. However, the existing PoR schemes usually encode the

entire file data before outsourcing and do not provide implementation details when data corruption occurs. Consider the great need of supporting data dynamics in practice, the cloud storage system should provide efficient data operations and recovery over encoded data in terms of both communication and computation costs.

We define $\langle P(F,\sigma), V\rangle(pk)$ to be a public proof, where P takes as input a file F and a set of tags σ, and a public key pk, where $P(x)$ denotes the prover P holds the secret x and $\langle P, V\rangle(x)$ denotes the prover P and the verifier V share x in the protocol execution. In the following, we give the definition of the security model of a PoR based cloud storage system.

Definition 1. Security model: A pair of interactive machines (P, V) [23] is called an available proof of retrievability for a file F if P is a (unbounded) probabilistic algorithm, V is a deterministic polynomial-time algorithm, and the following conditions hold for some polynomial $p_1(\cdot)$, $p_2(\cdot)$, and all $\kappa \in \mathbb{N}$:

•$Correctness$: For every $\sigma \in TagGen(sk, F)$,

$$Pr[\langle P(F,\sigma), V\rangle(pk) = 1] \geq 1 - 1/p_1(\kappa) \quad (1)$$

•$Soundness$: For every $\sigma^* \notin TagGen(sk, F)$, every interactive machine P^*,

$$Pr[\langle P^*(F,\sigma^*), V\rangle(pk) = 1] \leq 1/p_2(\kappa) \quad (2)$$

where $p_1(\cdot)$ and $p_2(\cdot)$ are two polynomials, and κ is the security parameter.

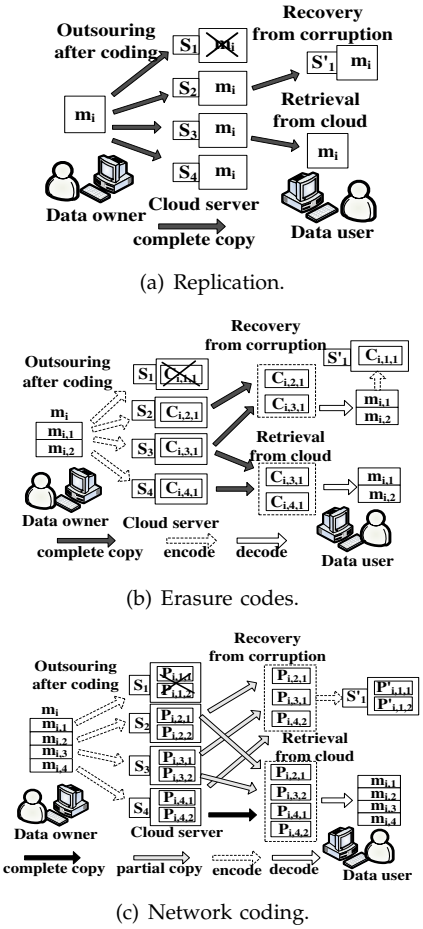
In this definition, the function $1/p_1(\kappa)$ is called correctness error, and the function $1/p_2(\kappa)$ is called soundness error. For non-triviality, we require $1/p_1(\kappa) + 1/p_2(\kappa) \leq 1 - 1/poly(\kappa)$. Different from correctness, soundness implies that it is infeasible to fool the verifier into accepting false statements. Soundness can also be regarded as a more strict notion of unforgeability for the file tags. Thus, the above definition means that the prover cannot forge the file tags or tamper with the data if soundness property holds.

3.4 Preliminaries on Redundancy Approaches for Data Recovery

Data redundancy techniques such as replication, coding (e.g., erasure codes and network coding) are usually employed in the distributed storage systems to improve the data reliability and availability [25].

1) Replication is the simplest way to generate redundancy. In the replication approach, the original data is completely copied to each of n storage servers. Data users can retrieve the original data by accessing any one of the storage servers. When one server is corrupted, the original data can be recovered by simply copying the entire data from any one of the healthy servers. Fig.2(a) gives an example of the replication technique. Obviously, the storage overhead of this redundancy method is very expensive.

2) Fig.2(b) gives an example of using erasure codes for obtaining data redundancy. Data users can recover the whole m original data blocks by retrieving the same number of encoded data blocks from any k of n servers. As a redundancy-reliability tradeoff each server only needs to store m/k encoded data blocks. However, compared with the replication-based solution, erasure codes may have a higher network overhead for the data recovery. For example, Reed-Solomon codes [29], which is a typical erasure code, usually need to reconstruct all the original data blocks in order to generate an encoded block. In other words, in order to generate only m/k encoded data blocks for a failed server, we have to retrieve m encoded data blocks. In Fig.2(b), the parameters (m, k, n) are set to be (2,2,4), respectively. The original data block m_i is first divided into 2 native data blocks equally. These two native blocks are then encoded into 4 encoded blocks $C_{i,j,l}$ which are stored in 4 cloud servers. Each cloud server stores one encoded blocks. The size of $C_{i,j,l}$ is equal to the size of the native data block.



(a) Replication.

(b) Erasure codes.

(c) Network coding.

Fig. 2. Data redundancy techniques.

3) Using network coding [19], [30], we first divide a given file object into equal-size native data blocks. The native data blocks then are encoded by linear combination to form encoded data blocks which are distributed over $n > k$ storage servers. The original data file object can be reconstructed from the encoded blocks contained in any k of the n servers. Thus, it tolerates the failure

of any $n-k$ storage servers. The network coding can achieve a tradeoff between storage and communication cost. Fig.2(c) gives an example of network coding for achieving data redundancy. The parameters (n,k) are set to be (4,2). In this example, the original data block m_i is first divided into 4 native data blocks with equal size. These native data blocks are then encoded into 8 encoded blocks $P_{i,j,l}$ and store them in 4 cloud servers, where each cloud server stores two encoded blocks. Here, the size of $P_{i,j,l}$ is equal to the size of the native data block.

Based on the above illustrations, the storage cost of replication is more expensive and the recovery communication cost of erasure codes is higher than replication. Compared to replication and erasure coding, network coding [19] can provide a significant advantage over erasure codes: a new encoded fragment can be constructed with optimal minimum communication cost by contacting some of the healthy servers (the communication cost of data recovery can be made as low as the repaired fragment) [22]. Actually, the use of network coding can make a tradeoff between the storage cost and the communication cost. Motivated by this observation, we employ the network coding as the outer code to tolerate the entire cloud server corruption. In particular, instead of encoding the entire data file, we split up the data into data blocks and redundantly encode each data block individually via network coding to support data dynamics and make a performance tradeoff. To tolerate data corruption inside a server and achieve a low level of communication overhead, we adopt the erasure codes as the inner code.

In the following discussions, we define the *repair traffic factor* as the ratio between the amount of data that needs to be retrieved from other surviving servers to the amount of data that is generated to be stored on a new server. The *repair traffic factor* can be a metric to measure the communication cost of the repair component. In Fig.2, assume the size of data block m_i is 1MB, and the subblock of m_i is equal-size. Then, the total storage cost and *repair traffic factor* of replication, erasure codes and network coding are (4MB,1),(2MB,2),(2MB,1.5), respectively. We can see the network coding can save 50% storage cost compared with replication and 25% communication cost compared with erasure codes.

4 Our Construction

In this section, we will give a detailed description of our proposed construction. To obtain efficient data recovery, the key idea is to split up the data into data blocks and redundantly encode each data block individually so that an update inside any data block only affects the encoded blocks of the same original data block. The encoded blocks are then organized by rb23Tree to support secure and efficient data dynamic operations The aggregatable signature-based broadcast (ASBB) encryption scheme [18] is adopted to generate metadata tags for the encoded blocks, providing guarantee of data integrity. The inter coding and outer coding of outsourced data enables efficient recovery when data corruption occurs.

4.1 Overview

Our scheme consists of five stages: (1) Encoding stage. The data owner splits up the data into data blocks and encode them using network coding. Blocks stored in the same server are further encoded using erasure codes. (2) Setup stage. In this stage, the data owner executes the *KeyGen* and *TagGen* algorithms to generate system parameters and metadata tags for the encoded blocks. Then the data owner outsources the encoded blocks and metadata tags to the CSP. (3) Verification stage. The TPA interacts with CSP periodically to check the integrity of data on behalf of the data owner. To do so, the TPA runs *ChalGen* to launch a challenge, and the server executes *ProofGen* to compute the response which will be verified by *VerifyProof* executed by TPA. (4) Update stage. The data owner executes *UpdateRequest* to send an update request to the server, who runs *Update* to execute the update operation. In each update round, the data owner can verify whether the update request is correctly executed or not. (5) Recovery stage. When data corruption occurs, data owner executes the *Recovery* algorithm to recover from corrupted data files.

4.2 Encoding

We follow the same coding structure as HAIL [21] by exploiting both the within-server redundancy and cross-server redundancy to improve data reliability and availability. The key difference between the existing approaches and ours is that we adopt network coding instead of erasure codes for obtaining the cross-server redundancy.

Specifically, we utilize the *functional minimum storage regenerating (FMSR)* code [19] as the cross-server code. FMSR belongs to *Maximum Distance Separable (MDS)* codes. An MDS code is defined by the parameters (s,k), where $k<s$. An (s,k)-MDS code means that the original data can be reconstructed from any k out of s servers. FMSR encodes a data file F of size $|F|$ into $s(s-k)$ data blocks of size $|F|/(k(s-k))$ each.

1) Outer code. Let $F=(m_1,m_2,\cdots,m_n)$ be the data file, and $EM=[\alpha_{l,j}]$ be an encoding matrix for some coefficients in the Galois field $GF(2^8)$ where $l=1,\cdots,s(s-k), j=1,\cdots,k(s-k)$. Each row vector of EM is an encoding coefficient vector (ECV) that contains $k(s-k)$ elements. We use ECV_i to denote the ith row vector of EM. For each block $m_i(1\le i\le n)$, we first divide it into $k(s-k)$ equal-size native blocks. Then, we encode these $k(s-k)$ native blocks into $s(s-k)$ encoded blocks, denoted by $P_{i,l}$ which is computed by the scalar product of ECV_i and the native blocks vector m_i, i.e. $P_{i,l}=\sum_{j=1}^{k(s-k)}\alpha_{l,j}m_{i,j}$, where $1\le i\le n, 1\le l\le s(s-k)$.

All arithmetic operations are performed over $GF(2^8)$. Each $P_{i,l}$ is formed by a linear combination of the $k(s-k)$

native blocks. The encoded blocks $P_{i,l}$ are then stored in the s storage servers, each having $s-k$ blocks. We use $P_{i,t,j}(1 \leq i \leq n, 1 \leq t \leq s, 1 \leq j \leq s-k)$ to denote the encoded block on a server, i.e. the j-th encoded block of m_i on the t server. There are many ways of constructing EM, as long as it satisfies the MDS property and the repair MDS property [19].

2) Inner code. In order to save communication cost, we use an (s', k') erasure codes as the within-server code to encode each $P_{i,t,j}$ into a new encoded block $P'_{i,t,j,q}(1 \leq q \leq s')$. An (s', k') erasure code encodes k' fragments of data block into s' fragments such that up to $\lfloor (n'-k')/2 \rfloor$ errors, or up to $n'-k'$ erasures can be corrected. When a small fragment is corrupted, the server can recover the original data from the corruption locally instead of retrieving data blocks from other healthy servers.

As shown in Fig.3, we use an (4,2)-FMSR code to achieve the cross-server redundancy and an (5,3) erasure code to achieve the within-server redundancy.

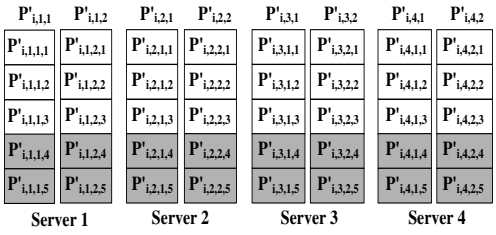


Fig. 3. Within-server redundancy and cross-server redundancy.

4.3 Initialization

Let G and G_T be multiplicative cyclic groups of the same prime order p. A bilinear map is a map $e : G \times G \rightarrow G_T$ with the following properties [31]: 1)Computable: there exists an efficiently computable algorithm for computing e; 2)Bilinear: for all $u, v \in \mathbb{Z}_p$, it holds that $e(g^u, g^v) = e(g, g)^{uv}$; 3)Non-degenerate: $e(g, g) \neq 1$ for any generator g of G; 4)for any $u_1, u_2, v \in G$, $e(u_1 u_2, v) = e(u_1, v) \cdot e(u_2, v)$. Let $h(\cdot) : \{0,1\}^* \rightarrow G$ be a secure hash function mapping a string to G uniformly. The system parameters and metadata tags are generated as follows.

1) *KeyGen*: The data owner randomly selects an element $r \in \mathbb{Z}_p^*, X \in G \backslash \{1\}$. Then the data owner computes $R = g^{-r}, A = e(X, g)$. The system public parameters are (g, h, p, G, G_T, e), the public key is $pk = (R, A)$ and the secret key is $sk = (r, X)$.

2) *TagGen*: Given a file F, the data owner generates an identity fid for F and divides F into n blocks, i.e. $F = (m_1, \cdots, m_n)$ where $m_i \in \mathbb{Z}_p^*$. For each block m_i, the data owner encodes it into $s's(s-k)$ encoded blocks $P_{i,l,q}(1 \leq i \leq n, 1 \leq l \leq s(s-k), 1 \leq q \leq s')$ via a (s, k)-FMSR code and a (s', k')-erasure code. In order to tolerate cloud server data corruption, the data owner stores $ns's(s-k)$ encoded blocks in s cloud severs. Each cloud server stores $ns'(s-k)$ encoded blocks. In each cloud server, these $ns'(s-k)$ encoded blocks are organized by an rb23Tree $T_t(1 \leq t \leq s)$. In each T_t, each node stores $s'(s-k)$ encoded blocks of the same original data block.

The data owner then computes the hash value of the $s'(s-k)$ encoded blocks of data block $m_{i,t}(1 \leq i \leq n, 1 \leq t \leq s)$ for each cloud server, i.e. $H_{i,t} = h(fid||P_{en})$, where $P_{en} = P'_{i,t,1,1}|| \cdots ||P'_{i,t,1,s'}|| \cdots ||P'_{i,t,s-k,1}|| \cdots ||P'_{i,t,s-k,s'}$ Finally, the data owner computes the tag $\sigma_{i,t}$ for the encoded blocks in each server $m_{i,t} : \sigma_{i,t} = (X^{P_{en}} H_{i,t})^r$. Denote the set of all tags by $\phi_t = \{\sigma_{i,t}\}_{1 \leq i \leq n, 1 \leq t \leq s}$. Then, the data owner sends the encoded blocks $P'_{i,t,j,q}$ with the information $au_{s,t} = \{fid, \phi_t, T_t\}$ to the corresponding server, sends the fid and tag value $v(root)_t$ of root node of each rb23Tree to TPA and keeps the information $au_{o,t} = \{fid, v(root)_t\}$ with the encoding matrix EM locally.

3) rb23Tree: The range-based 2-3 tree or rb23Tree for short can not only offer the dynamic property of 2-3 trees with logarithmic complexity but also allows the verifier to verify the value and index of the leaf node.

In the rb23Tree, each node stores three types of information:

•$l(k)$: the height of node k. The height of leaf node is defined 1.

•$r(k)$: the *range value* of node k, namely the number of leaves corresponding to the subtree rooted at k. If k is a leaf, $r(k)$ is 1 and if k is NULL, $r(k)$ is 0. The $r(k)$ of the root node is the number of leaves in the rb23Tree.

•$v(k)$the *tag value* of node k. $v(k)$ is defined as $H(l(k)||r(k)||v(ch_1)||v(ch_2)||v(ch_3))$ or e_k or 0 when $l(k) > 1$ or k is a leaf or k is NULL, respectively. Here, $||$ is the concatenation operation, ch_1, ch_2, ch_3 are the tree left-to-right children of k (when k only has two children, ch_3 is NULL), e_k is the element value stored in leaf node k, and $H()$ is a collision-resistant hash function.

Following [15], we also define π_i to be a proof path for locating the ith leaf by traversing the path starting at the root node. We also define the $min(k)$ and $max(k)$, which denote the minimum and maximum leaf indices that can be reached via node k, respectively.

When locating an appointed leaf node whose index is i, we need to calculate the values of $min()$ and $max()$ from the root node to subjacent node step by step. Note that a node k is on the path from the ith leaf node to the root node if and only if $i \in \{min(k), max(k)\}$.

Assuming the proof path $\pi_i = \{k_1, \cdots, k_z\}$, where k_1 is the ith leaf node, k_z is the root node, and each node $k_j \in \pi_i$ is associated with an 8-element tuple $mark(k_j) = \{l(k_j), r(k_j), r(c_1), v(c_1), r(c_2), v(c_2), r(c_3), v(c_3)\}$. When $j = 1$, k_j is the leaf node and $r(k_j)$ in $mark(k_j)$ is $v(k_j)$, i.e., the tag value of the leaf node. c_1, c_2, c_3 are k_j's three left-to-right children and $r(c_t) = v(c_t) = -1 (1 \leq t \leq 3)$ if $c_t \in \pi_i$, $r(c_t) = v(c_t) = 0$ if c_t does not exist.

In Fig.4, we give an example of rb23Tree. We use $k_{i,j}$ to denote a node where i is the height and the j is the

index. Each inner node k stores $(l(k), r(k), v(k))$. Suppose we want to verify the information of the 7th leaf node. The proof path is $\pi_i = \{k_{1,7}, k_{2,3}, k_{3,2}, k_{4,1}\}$. The tuples along the proof path are shown in Table 1.

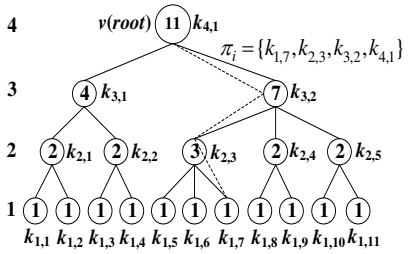


Fig. 4. An Example of rb23Tree.

TABLE 1
Information on proof path of the 7th leaf node

	$l(k)$	$r(k)$	$r(c_1)$	$v(c_1)$	$r(c_2)$	$v(c_2)$	$r(c_3)$	$v(c_3)$
$k_{4,1}$	4	11	4	$v(k_{3,1})$	-1	-1	0	0
$k_{3,2}$	3	7	-1	-1	2	$v(k_{2,4})$	2	$v(k_{2,5})$
$k_{2,3}$	2	3	1	$v(k_{1,5})$	1	$v(k_{1,6})$	-1	-1
$k_{1,7}$	1	$v(k_{1,7})$	0	0	0	0	0	0

4.4 Data Integrity Verification

After the encoded data file with the metadata tags and the rb23Tree are outsourced to the server, TPA can periodically launch integrity checks on behalf of the data owner. On receiving the challenge, the server generates a proof and sends it to TPA.

1) *ChalGen*: In each auditing round, TPA first randomly selects a number $t_1 \in \mathbb{Z}_p^*$ and computes $c_1 = g^{t_1}$. Then, TPA randomly picks c elements $I = \{s_1, \cdots, s_c\}$ from the set $1, \cdots, n$ where n is the number of data blocks. Without loss of generality, we assume $s_1 \leq \cdots \leq s_c$ that can be generated using pseudo-random permutation. For each s_i in I, TPA chooses a random value $v_i \in \mathbb{Z}_p^*$. Then, the TPA sends the challenge $chal = \{(i, v_i)_{i \in I}, c_1\}$ to each server. According to the $chal$, each server returns $\{H_{i,t(i \in I, 1 \leq t \leq s)}\}$ to TPA. In response, TPA chooses a random element $m \in G_T$ and computes $c_2 = R^{t_1}, \omega_t = m \cdot e(\prod_{i \in I} H_{i,t}^{v_i}, c_2)$. Finally, the value ω_t is sent back to each server.

2) *ProofGen*: All servers run this algorithm to generate proofs to prove the integrity of the checked encoded blocks. Specifically, each server executes the following computing independently:

$$\sigma_t = \prod\nolimits_{i \in I} \sigma_{i,t}^{v_i}, \mu_t = \sum\nolimits_{i \in I} v_i P_{en}, \quad m_t^* = \omega_t \cdot e(\sigma_t, c_1) \tag{3}$$

Each server then sends the m_t^* and μ_t as the proofs to TPA. All the proof paths $\pi_{i(i \in I)}$ are also returned to TPA. So for each server the proof P is $P = \{m_t^*, \mu_t, \pi_{i(i \in I)}\}$.

3) *VerifyProof*: TPA runs this algorithm to validate the proof P from each server. For $t = 1, \cdots, s$ TPA first calculates $B_t = e(X, c_2^{-1})^{\mu_t}$ according to the μ_t returned by CSP. Then, TPA checks whether equation (4) holds and executes **Algorithm 1** to verify whether $position[k] = i$ and $value[k] = v(root)_t$.

$$mB_t \stackrel{?}{=} m_t^* \tag{4}$$

Algorithm 1 $Examine(v(root)_t, \pi_{i,t})$

This algorithm allows an entity, who knows $v(root)_t$, to verify the ith element e_i of (ordered set) $S = \{e_1, \cdots, e_n\}$ is stored exactly at the ith leaf by examining proof path (ordered set) $\pi_{i,t} = v_1, \cdots, v_k$ provided by the server.

1)initialize an array $position[1 \cdots k] = 0$ and array $value[1 \cdots k] = 0$. $//position$ tracks the index of the leaf that will be checked with $v(\cdot) = e_i$, $value$ tracks $v(v_j)$ where $v_j \in \pi_{i,t(i \in I)}$.

2)$position[1] \leftarrow 1$, $value[1] \leftarrow e_i$.

3)for $j = 2, \cdots, k$, do $//v_j$ has three children ch_1, ch_2, ch_3

if $ch_1 \in \pi_{i,t(i \in I)}$, i.e., $r(ch_1) = -1, v(ch_1) = -1$, then

$position[j] \leftarrow position[j-1]$,
$value[j] \leftarrow H(l(k_j)||r(k_j)||value[j-1]||v(ch_2)||v(ch_3))$.

end if

if $ch_2 \in \pi_{i,t(i \in I)}$, i.e., $r(ch_2) = -1, v(ch_2) = -1$, then

$position[j] \leftarrow position[j-1] + r(ch_1)$,
$value[j] \leftarrow H(l(k_j)||r(k_j)||v(ch_1)||value[j-1]||v(ch_3))$.

end if

if $ch_3 \in \pi_{i,t(i \in I)}$, i.e., $r(ch_3) = -1, v(ch_3) = -1$, then

$position[j] \leftarrow position[j-1] + r(ch_1) + r(ch_2)$,
$value[j] \leftarrow H(l(k_j)||r(k_j)||v(ch_1)||v(ch_2)||value[j-1])$.

end if

end for

4)if $position[k] = i$ and $value[k] = v(root)_t$, then

return *TRUE*

else

return *FALSE*

end if

If they all hold, *VerifyProof* outputs 1, otherwise outputs 0, meaning there is a data corruption. The TPA will return the results and the corruption location to the data owner.

4.5 Secure Data Updates

In this subsection, we discuss the dynamic update operations including block modification, block insertion and block deletion.

1)*UpdateRequest*: The data owner sends the update request $\{op, I_2 = \{i-1, i, i+1\}\}$ to each server where $op \in \{M, I, D\}$ is the update operation, i is the update index. After receiving the update request, each server returns the corresponding proof path $\pi_{j,t}(j \in I_2)$ to the data owner. The data owner then calls the $Examine(v(root)_t, \pi_{j,t})(j \in I_2)$ to verify the validity of

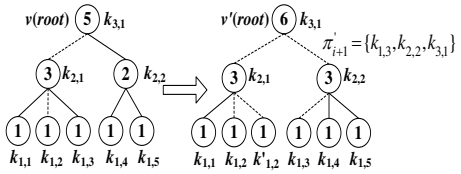
the path. If all verifications have been passed, the data owner executes the following operations according to the op (Without loss of generality, we assume $2 \leq i \leq n$.).

$op = M$: The data owner downloads $s'k(s-k)$ encoded blocks of m_i from any k of the s servers and decodes them to recover the original data block m_i (see the ***Decoding Procedure***). Then data owner encodes the new block m_i^* using the original encoding matrix EM stored locally, generate new encoded blocks $P^*_{i,t,j,q}$ and compute the new tags $\sigma^*_{i,t}$. The new encoded blocks, the new tags are sent to the each corresponding server.

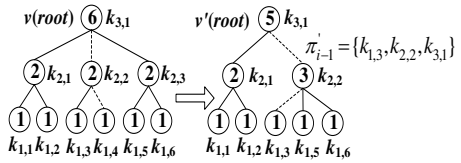
$op = I$: The data owner generates the encoded blocks $P^*_{i,t,j,q}$ for the new block m_i^* and computes the tags $\sigma^*_{i,t}$. The new encoded blocks, the new tags are then sent to the each corresponding server.

$op = D$: The data owner sends the deletion instruction and the index i to the each corresponding server.

2) *Update*: After receiving the update request, each server will adjust his own rb23Tree T_t according to the request. Fig.5 gives the examples of block insertion and block deletion. According to the proof path $\pi_{j,t}(j \in I_2)$ received from each server, the data owner can construct a partial rb23Tree and update the information on $\pi_{j,t}(j \in I_2)$ by himself. We use $\pi_{j\_new,t}(j \in I_2)$ to denote the new proof path the data owner maintains after updating path $\pi_{j,t}$ himself. The data owner can compute a new $v(root)_{\_new,t}$ according to the $\pi_{j\_new,t}$. In addition, we use T'_t to denote the new rb23Tree in the server t after updates. After adjustment, each server will send a new path information $\pi'_{i-1,t}$ or $\pi'_{i+1,t}$ to the data owner according to the new rb23Tree T'_t. Then, the data owner calls the $Examine(v(root)_t, \pi_{j',t})(j' = \{(i-1)', (i+1)'\})$ to compute the new root value $v(root)'_t$ of T'_t, which is further compared with the $v(root)_{\_new,t}$ computed by the data owner himself to verify the correctness of the update request execution and the rb23Tree update.



(a) Data block insertion.



(b) Data block deletion.

Fig. 5. Example of data dynamic operation.

4.6 Data Recovery

By periodical integrity checking, the TPA may find out the outsourced data is corrupted. Then, the TPA can locate the corrupted fragments or failed server via binary verification just like the binary search and return the positions to the data owner.

When a server is still available but some small fragments of data are corrupted, i.e., for the encoded block $P'_{i,t,j,q}(1 \leq i \leq n, 1 \leq t \leq s, 1 \leq j \leq s-k, 1 \leq q \leq s')$, the number of errors is less than $\lfloor (s'-k')/2 \rfloor$, or the number of erasures is less than $s'-k'$. The server then can correct the errors or erasures locally by the erasure codes, which involves no communication cost, and there is no need to recompute the metadata tags. When a server is down, the data owner can execute the ***iterative recovery procedure*** to recover the failure and generate new encoded blocks and the corresponding metadata tags.

Iterative Recovery Procedure: The recovery process for a permanent single-server data corruption is as follows:

1) Select $s-1$ $ECVs$ randomly. Suppose the encoded blocks of the m_i on the server t are corrupted, i.e., $P'_{i,t,1,1}, \cdots, P'_{i,t,1,s'}, \cdots, P'_{i,t,s-k,1}, \cdots, P'_{i,t,s-k,s'}$. The data owner selects $s-1$ $ECVs$ from encoded matrix EM as follows: exclude the $s-k$ $ECVs$ from $(i-1)(s-k)+1$ to $(i-1)(s-k)+s-k$ where $i = 1, \cdots, s$. For each $j = 1, \cdots, s$ and $j \neq i$, choose one ECV from $(j-1)(s-k)+1$ to $(j-1)(s-k)+s-k$ randomly. Then $s-1$ $ECVs$ are selected. Each ECV in the encoded matrix EM is corresponding to one of the encoded blocks. We denote these $ECVs$ by $ECV_{i_1}, ECV_{i_2}, \cdots, ECV_{i_{s-1}}$.
2) Generate a repair matrix. The data owner constructs a repair matrix $RM = [\gamma_{i,j}]$, where $i = 1, \cdots, s-k, j = 1, \cdots, s-1$. Each element $\gamma_{i,j}$ is randomly selected from $GF(2^8)$.
3) Compute $ECVs'$ for the new encode blocks and generate a new encoding matrix EM'. The data owner multiplies the RM generated in 2) with the $ECVs$ picked in 1) to construct $s-k$ new $ECVs'$, which are denoted by $ECV'_i = \sum_{j=1}^{s-1} \gamma_{i,j} ECV_{i_j}$ for $i = 1, 2, \cdots, s-k$. Generate a new encoding matrix denoted by EM' as follows: when server $i(1 \leq i \leq s)$ is a healthy server the corresponding $s-k$ row vectors of EM' is ECV_{i_j}, where $(i-1)(s-k)+1 \leq j \leq (i-1)(s-k)+s-k$ and each ECV_i is selected from the original EM. When server i is corrupted, the corresponding $s-k$ row vectors of EM' are ECV'_j where $1 \leq j \leq s-k$.
4) Check whether both the MDS and repair MDS properties are satisfied or not. If either check fails, the data owner returns to 1) and repeats the above steps.
5) Download the actual encoded blocks and regenerate new encoded blocks. Using the RM multiply the $s-1$ blocks selected from $s-1$ servers corresponding to the $s-1$ $ECVs$ selected in 1) to generate new encoded blocks, which are encoded again via a (s', k')-erasure codes. The encoded blocks, the corresponding metadata tags and rb23Tree are stored in a new server.

Discussion. In our construction, every original data block is encoded by the same encoding matrix EM and the single-server data corruption recovery will generate a new encoding matrix EM'. Therefore, the blocks encoded by EM cannot be decoded by EM'. Observing that the new encoding matrix EM' contains ECV_i and ECV_i', we adopt the incremental storage method to solve this problem. In each round of single-server data corruption recovery, we record the data corruption server index, data block index, ECV_i index and the new ECV_i' in a table. We store the original encoding matrix EM and can generate the new encoding matrix EM' by using EM and ECV_i'. When there are $1 < d \leq k$ data corruption servers, we can download the encoded blocks from any $s-k$ healthy severs and decode them to recover the original data blocks. We encode the original data blocks again and upload the new encoded blocks and the corresponding metadata tags and rb23Trees to the servers.

Decoding Procedure: To access the original data blocks, the data owner needs to decode the encoded blocks. The data owner selects any k of the s storage servers and downloads the $k(s-k)$ encoded blocks from the k cloud servers. Then, the data owner selects the corresponding metadata object that contains the $ECVs$. The $ECVs$ of the $k(s-k)$ encoded blocks can form a $k(s-k) \times k(s-k)$ square matrix. If the MDS property is maintained, then the inverse of the square matrix must exist. Therefore, the data owner can multiply the inverse of the square matrix with the encoded blocks to obtain the $k(s-k)$ native blocks. When there are $d > k$ data corruption servers, we cannot recover the original file. In this case, the TPA and the data owner have known the data corruption and can launch a lawsuit against the CSP.

5 Security Analysis

5.1 Correctness and Soundness

The correctness of the above verification protocol can be elaborated as follows:

$$
\begin{aligned}
m_t^* &= \omega_t \cdot e(\sigma_t, c_1) = m \cdot e(\prod_{i\in I} H_{i,t}^{v_i}, c_2) \cdot e(\prod_{i\in I} \sigma_{i,t}^{v_i}, g^{t_1}) \\
&= m \cdot e(\prod_{i\in I} H_{i,t}^{v_i}, g^{-rt_1}) \cdot e(\prod_{i\in I} (X^{P_{en}} H_{i,t})^{rv_i}, g^{t_1}) \\
&= m \cdot e(\prod_{i\in I} H_{i,t}^{-rv_i} \prod_{i\in I} (X^{P_{en}} H_{i,t})^{rv_i}, g^{t_1}) \\
&= m \cdot e(\prod_{i\in I} X^{rv_i P_{en}}, g^{t_1}) \\
&= m \cdot e(X, g^{rt_1})^{\sum_{i\in I} v_i P_{en}} \\
&= m \cdot e(X, c_2^{-1})^{\sum_{i\in I} v_i P_{en}} \\
&= m \cdot B_t
\end{aligned}
\tag{5}
$$

In Equation 5, $P_{en} = P'_{i,t,1,1}||\cdots||P'_{i,t,1,s'}||\cdots||P'_{i,t,s-k,1}||\cdots||P'_{i,t,s-k,s'}$ denotes the aggregate value of the total $s'(s-k)$ encoded blocks of data block m_i on the server t using both network coding and erasure codes. We use the bilinear map property that for all $u_1, u_2, v \in G$, $e(u_1u_2, v) = e(u_1, v) \cdot e(u_2, v)$ to eliminate the $H_{i,t}$.

As discussed above, our proposed verification protocol is based on the aggregatable signature-based broadcast (ASBB) scheme. In the Sign algorithm we compute the signature $\sigma = X^{\alpha} h(\beta)^r$ instead of $\sigma = Xh(\beta)^r$. Correspondingly, in the Verify algorithm the verification equation is $e(\sigma, g)e(h(\beta), R) = A^{\alpha}$. The security of the protocol can be reduced to the security of the scheme in [18]. Take the signature scheme as an example, if the adversary can forge a valid signature (α, β, σ) in the present protocol with the public key (R, A), he can forge a valid signature $(s, \sigma^{\alpha^{-1}})$ for the ASBB scheme with the public key $(R^{\alpha^{-1}}, A)$. However, the probability is negligible [18].

Definition 2. Discrete Logarithm (DL) Problem: Given $g, g^t \in G$, it is hard to compute $t \in \mathbb{Z}_p$.

Definition 3. Computational Diffie-Hellman (CDH) Problem: Given $g, g^{\alpha}, h \in G$ for unknown α, it is hard to compute $h^{\alpha} \in G$.

Theorem 1. *If the signature scheme used for file tags is existentially unforgeable and the DL problem and the CDH problem are hard, then if the cloud server does not possess the specific data intact as it is, he cannot pass the audit phase with non-negligible probability.*

Proof. After the *ChalGen* stage, if the server can obtain the value of c_2, he can compute $e(\prod_{i\in I} H_{i,t}^{v_i}, c_2)$ to recover the random element m according to the $\omega_t = m \cdot e(\prod_{i\in I} H_{i,t}^{v_i}, c_2)$. Following the equation (4), the server can know the B_t which leak the information m and B_t that should only be known to TPA to the server. However, it is hard to compute t_1 and $c_2 = R^{t_1}$ from $c_1 = g^{t_1}$ as the discrete logarithm problem is hard. Thus, the server cannot compute and obtain c_2.

Now we will prove if the adversary can cheat TPA and pass the verification, then the CDH problem can be broken by a simulator. Given $g, g^r, h \in G$ and r is unknown, the simulator's goal is to output $h^r \in G$.

The challenger keeps a list of its responses to *TagGen* queries generated by the adversary. Now the challenger observes each instance of the proof-of-retrievability protocol with the adversary. If in any of these instances the adversary is successful (i.e. *VerifyProof* outputs 1) but the adversary's aggregate signature σ_t is not equal to $\sum_{(i,v_i)\in chal} \sigma_{i,t}^{v_i}$ (where *chal* is the challenge issued by the verifier and $\sigma_{i,t}$ is the metadata tag on the blocks of the file considered in the protocol instance) the challenger declares failure and aborts.

Suppose $chal = (i, v_i)$ is the query that causes the challenger to abort, and the adversary's responses to compute the m_t^* are $(\mu_t^*, \sigma_t^*, H_{i,t}^*)$. Let the expected responses-i.e., the one that would have been obtained from an honest prover-be $(\mu_t, \sigma_t, H_{i,t})$, where $\mu_t = \sum_{i\in I} v_i P_{en}$, $\sigma_t = \prod_{i\in I} \sigma_{i,t}^{v_i}$ and $H_{i,t} = h(fid||P_{en})$. Because of the authentication in rb23Tree, the $H_{i,t}^*$ should be the same with $H_{i,t}$. Otherwise, the $Examine(v(root)_t, \pi_{i,t})$ algorithm will output *False* directly.

With this adversary, a simulator [10] could break the

CDH problem instance as follows: The simulator randomly chooses values $\beta, \gamma \in \mathbb{Z}_p$ and sets $X = g^{\alpha}h^{\gamma}$. For each i, $1 \leq i \leq n$, The simulator selects a random value $r_i \in \mathbb{Z}_p$, and programs the random oracle at i as $H_{i,t} = g^{r_i}/(g^{\beta P_{en}} \cdot h^{\gamma P_{en}})$. Now the simulator can compute the $\sigma_{i,t}$ as follows.

$$\sigma_{i,t} = (X^{P_{en}} H_{i,t})^{r} = (g^{r})^{r_i} \tag{6}$$

By the correctness of the scheme, we know that the expected response satisfies the verification equation, i.e., that

$$B_t = e(X, c_2^{-1})^{\mu_t} = e(\sigma_t, c_1) \cdot e(\prod_{i \in I} H_{i,t}^{v_i}, c_2) \tag{7}$$

Because the challenger aborted, we know that $\sigma_t \neq \sigma_t^*$ and that σ_t^* passes the verification equation, i.e., that

$$B_t^* = e(X, c_2^{-1})^{\mu_t^*} = e(\sigma_t^*, c_1) \cdot e(\prod_{i \in I} H_{i,t}^{*\ v_i}, c_2) \tag{8}$$

Obviously, $\mu_t \neq \mu_t^*$, otherwise, $\sigma_t = \sigma_t^*$, which contradicts our assumption above.

We can define $\Delta\mu_t = \mu_t^* - \mu_t \neq 0$. From equation (8) and (7), we can get:

$$\begin{aligned}
&e(X, c_2^{-1})^{\Delta\mu_t} = e(\sigma_t^*/\sigma_t, c_1) \Leftrightarrow \\
&e(X, g^{rt_1})^{\Delta\mu_t} = e(\sigma_t^*/\sigma_t, g^{t_1}) \Leftrightarrow \\
&X^{r\Delta\mu_t} = \sigma_t^*/\sigma_t \Leftrightarrow \\
&(g^{\beta}h^{\gamma})^{r\Delta\mu_t} = \sigma_t^*/\sigma_t \Leftrightarrow \\
&h^{r} = (\sigma_t^*/\sigma_t \cdot (g^{r})^{-\beta\Delta\mu_t})^{1/(\gamma\Delta\mu_t)}
\end{aligned} \tag{9}$$

meaning the simulator can output the $h^r \in G$, which contradicts the CDH problem. That is to say there is no $\sigma_t \neq \sigma_t^*$ to make σ_t^* pass the verification equation to abort the protocol.

From the above process, we can conclude that $\sigma_t = \sigma_t^*$. It is only the values μ_t and μ_t^* that can differ and cause the abortion of the protocol. We also define $\Delta\mu_t = \mu_t^* - \mu_t \neq 0$. The simulator answers the adversary's queries until the protocol is aborted. Then given $g, h \in G$, the simulator can break the discrete logarithm problem as follows.

The simulator still randomly chooses values $\beta, \gamma \in \mathbb{Z}_p$ and sets $X = g^{\alpha}h^{\gamma}$. According to the equation (8) and (7) and $\sigma_t^* = \sigma_t$, we can get:

$$\begin{aligned}
&e(X, c_2^{-1})^{\mu_t} = e(X, c_2^{-1})^{\mu_t^*} \Leftrightarrow \\
&X^{\mu_t} = X^{\mu_t^*} \Leftrightarrow \\
&X^{\Delta\mu_t} = 1 \Leftrightarrow \\
&(g^{\beta}h^{\gamma})^{\Delta\mu_t} = 1 \Leftrightarrow \\
&h = g^{-\beta\Delta\mu_t(\gamma\Delta\mu_t)^{-1}}
\end{aligned} \tag{10}$$

meaning the simulator can output $x = -\beta\Delta\mu_t(\gamma\Delta\mu_t)^{-1}$ which satisfies $h = g^x$, i.e. the simulator breaks the discrete logarithm problem. That is to say there is no $\sigma_t = \sigma_t^*$ and $\mu_t \neq \mu_t^*$ to make σ_t^* pass the verification equation and abort the protocol. □

5.2 Root Node Information Protection

The tag value $v(root)_t$ of root node is the key factor to verify one leaf node is indeed stored at the specified position. In [7], when the TPA launches a data verification, the CSP can trick TPA by using any other blocks instead of the ones to be checked. The successful of this attack is caused by the lack of data position verification in the existing solutions. The use of rb23Tree can prevent this attack by running the $Examine(v(root)_t, \pi_{i,t})$ algorithm to guarantee the positions of the data blocks to be sampled are correct. After each update operation, the data owner or the TPA can calculate the new $v(root)\_{new,t}$ by itself. Then by comparing the two values $v(root)\_{new,t}$ and $v(root)'_t$ computed by the new proof path $\pi'_{i-1,t}$ or $\pi'_{i+1,t}$ according to the new rb23Tree T'_t, the data owner or the TPA can make sure whether the CSP executes the update request and sends back the specified blocks honestly or not.

Moreover, due to the authentication in the rb23Tree and the maintainance of the $v(root)_t$ of the root node, our scheme can also prevent the replay attack discussed in [22]. That is, the adversary's attempts to reuse old encoded blocks in order to reduce the redundancy on the storage servers, which makes the original data becomes unrecoverable. Consider a pollution attack [22], i.e., corrupted servers use correct data to avoid detection in the challenge phase, but provide corrupted data for coding new blocks in the repair phase. To prevent this attack, they need to encrypt the encoding coefficients and generate repair verification tags. In our scheme, the within-server corruption is handled by the cloud server to save communication cost and the cross-server corruption is handled by the data owner itself. Because the data owner maintains the $v(root)_t$, he can detect the pollution attack by the authentication of the rb23Tree.

5.3 Data Retrievability

Theorem 2. *Suppose a cheating prover on an n-block file F is well-behaved in the sense above, and that it is ϵ-admissible. Let $\omega = 1/\#B + (\rho n)^{\ell}/(n - c + 1)^{c}$. Then, provided that $\epsilon - \omega$ is positive and non-negligible, it is possible to recover a ρ-fraction of the encoded file blocks in $O(n/(\epsilon - \rho))$ interactions with cheating prover and in $O(n^2 + (1 + \epsilon n^2)(n)/(\epsilon - \omega))$ time overall.*

Proof. The verification of the proof-of-retrievability is similar to [10], we omit the details of the proof here. The difference in our work is to replace $H(i)$ with $H_{i,t}$ such that secure update can still be realized without including the index information. These two types of tags are used for the same purpose (i.e., to prevent potential attacks), so this change will not affect the extraction algorithm defined in the proof-of-retrievability. We can also prove that extraction always succeeds against a well-behaved cheating prover, with the same probability analysis given in [10]. □

Definition 4. Epoch: Suppose that the adversary can corrupt at most $s - k$ out of s servers within any given

time interval. We define such a time interval as an *epoch*.

Definition 5. Data recovery condition: In any given epoch, the original data can be recovery as long as at least k out of s servers collectively store at least n encoded blocks which are linearly independent combinations of the original n data blocks.

Theorem 3. *The data recovery condition is a sufficient condition to ensure data recoverability in our scheme augmented with protection against small data corruption.*

Proof. The proof is similar to [22]. We give the proof briefly as follows. In the initial state, the network coding guarantees that the original data can be recovered from any k out of s servers with high probability. We want to show that our scheme preserves this guarantee throughout its lifetime, thus ensuring data recoverability.

In any given epoch, the adversary can corrupt at most $s - k$ servers. The adversary may split the corruptions between direct data corruptions and replay attacks. Faulty servers affected by direct data corruptions can be detected by the integrity checks in the challenge stage, or by the correct encoding check in the recovery stage. The data owner uses the remaining surviving servers (at least k) to regenerate new encoded blocks to be stored on new servers. Due to the authentication of rb23Tree, the replay attack and pollution attack can be detected and prevented. Thus, at the end of the epoch, the system is restored to a state equivalent to its initial state, in which the data can be recovered from any k out of the s servers.

Theorem 4. *Given a fraction of the n blocks of an encoded file F, it is possible to recover the entire original file F with all but negligible probability.*

Proof. In Theorem 3, we prove the data recovery condition for the network coding is a sufficient condition to ensure data recoverability. And for the erasure codes, based on the rate-ρ Reed-Solomon codes, this result can be easily derived, since any ρ-fraction of encoded data blocks suffices for decoding. □

6 Experiment and Evaluation

6.1 Experiment Environment

We implement our proposed scheme using C language on a Linux system Ubuntu 10.04, with Intel Core (TM) 2 Quad CPU running at 2.67GHz, 2.00GB of RAM. We utilize Pairing-Based Cryptography (PBC) library version 0.5.7 and OpenSSL0.9.8 for implementation. The elliptic curve utilized in the experiment is a MNT curve, with base field size of 159 bits and the embedding degree 6. The security level is chosen to be 80 bits, which means $|v_i| = 80$ and $|p| = 160$.

6.2 Encoding/Decoding and Recovery Time

We first test the time cost of encoding, decoding and recovery operations. Our results are all averaged over 20 rounds.

The encoding and decoding time for different sizes of files are illustrated in Figure 6 and Figure 7, respectively. Figure 8 shows the single server failure recovery time and single block corruption recovery time for different sizes of files. In Fig. 6, Fig. 7 and Fig. 8, the parameters (s, k) of FMSR and (s', k') of R-S codes are set to be (4,2). As expected, the encoding, decoding and recovery time of FMSR and R-S codes increases with the file size. The time of encoding/decoding the same file for FMSR and R-S codes is almost the same. Due to the characteristics of the encoding schemes, the recovery time of FMSR is less than R-S codes. In FMSR, the original data can be reconstructed from the encoded blocks contained in any k of the s servers while for R-S codes to generate an encoded data block all the s original data blocks need to be reconstructed.

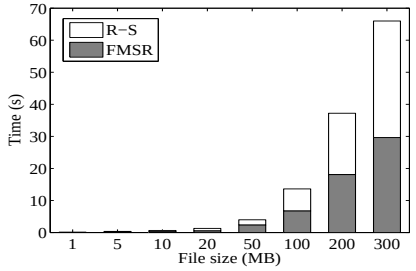


Fig. 6. *Encoding* time for different file sizes.

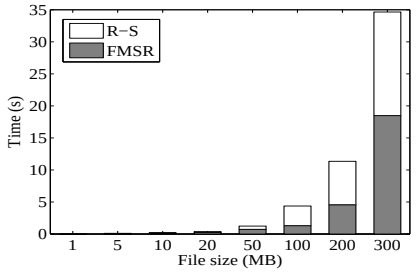


Fig. 7. *Decoding* time for different file sizes.

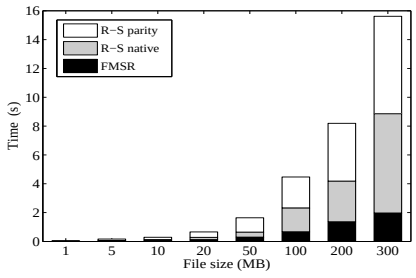


Fig. 8. Single server/block *Recovery* time for different file sizes.

We also evaluate the running time of one data block for different (s, k) values of FMSR and different (s', k') values of R-S codes. The results are shown in Fig. 9 and Fig. 10, respectively. In both Fig. 9 and Fig. 10, the block sizes are set to be 4KB. As shown, the running time increases when the parameters (s, k) and (s', k') increase.

The reason is that more subblocks have to be processed when the parameter (s, k) or (s', k') becomes larger.

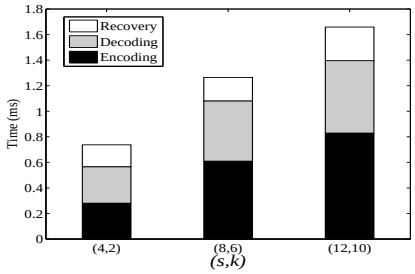


Fig. 9. Running time of different (*s*,*k*) values of FMSR.

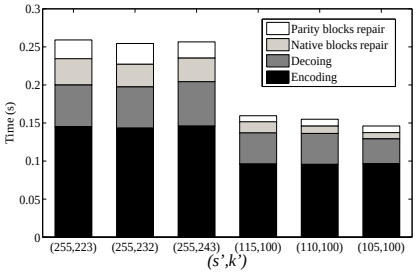


Fig. 10. Running time of different (*s'*,*k'*) values of R-S code.

6.3 The rb23Tree Generation Time and Storage Overhead

We show the time cost and storage cost under different file sizes in Table 2. The block size is set to be 4KB. Then given a file, the height of the rb23Tree is only related to the file size and the number of blocks. So, the height determines the time of generating an rb23Tree and the storage overhead.

TABLE 2
Time and storage cost of different files.

File size(MB)	64	128	256	512	1024
rb23Tree Height	10-15	11-16	11-17	12-18	13-19
Time cost(s)	0.990	1.908	4.180	7.511	15.946
Storage cost(MB)	1.125-1.5	2.25-3.0	4.5-6.0	9.0-12.0	18.0-24.0

From Table 2, we can see the time cost of generating an rb23Tree and storage cost is approximately linear with the file size. In our experiment the rb23Tree is generated randomly. Thus both the height and node number are undetermined while the value ranges are determined. We use notation $[X]$ to denote the minimum positive integer that is not less than positive integer X. Then the height of rb23Tree satisfies $[log_3 n] + 1 \leq height \leq [log_2 n] + 1$ and the node number satisfies $[(3n-1)/2] \leq num \leq 2n-1$.

6.4 Update Operation Overhead

We choose randomly 1000 blocks to execute the modification, insertion, deletion operations for different files and calculate the average time of operations for different data types. The results are shown in Fig. 11.

The results show that when there is an update operation, the leaf node and the information on the proof path will be changed. The rb23Tree will be also adjusted to keep the tree as an rb23Tree, and the length of proof path is determined by the height of rb23Tree. In the modification operation, since there is no rb23Tree adjustment, the modification cost is only related to the height. In the insertion/deletion operation, there exists rb23Tree adjustment which is determined by the type of the operated node. So there exist some fluctuations even if the heights are the same. In any case, the time cost can be maintained in the microsecond level, which shows the efficiency of our proposed protocol.

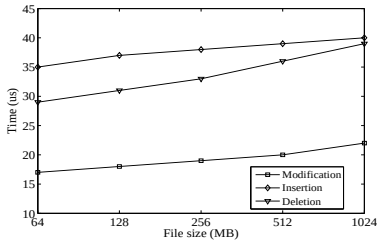


Fig. 11. Update operation overhead of rb23Tree.

6.5 Verification Cost

Now we evaluate the running time of the verification operation. For *TagGen*, we chooses 5 files with sizes of 128MB, 256MB, 512MB, 1024MB and 2048MB, respectively. Each data block is set to be 4KB. For each file, we generate the tags 10 times and compute the time average. The experiment results are shown in Fig. 12. We can see that it will take time to generate the metadata tags for the data files. However, the *TagGen* algorithm is only a one-time cost in the initialization process, and in the process of data updates only metadata of some data blocks are generated in stead of the whole file.

In our data integrity check process, we also need to verify the hash values of blocks and the proof path. As shown in [6], when there exists 1% data corruption and the sampled blocks are chosen randomly, TPA will

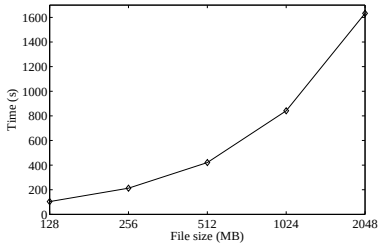


Fig. 12. *TagGen* time.

identify the corruption with a probability up to 95% and 99% by sampling 300 and 460 blocks, respectively. Hence, we follow the same strategy and also sample 300 and 460 blocks to evaluate the verification cost with a file of 1GB. We compare the time cost of our scheme with [10] and [11]. Note that we set the sector number s in [10] to be 1 to evaluate all schemes under the same benchmark. The results of performance comparison are shown in Table 3. As shown, although there are additional hash value and proof path checks, the whole verification cost of our scheme is almost the same as that of [10] and [11], which means our proposed scheme is efficient and practical for use in cloud storage systems. We next analyze the communication cost. The communication cost of the verification protocol is determined by the file size and the number of sampled blocks. Compared with [11], the additional communication overhead of our scheme is 40Bytes and the proof path. The size of each *mark* structure is 96Bytes. For a file of 1GB, the height of rb23Tree is 15, and the size of proof path is 1440Bytes. For one block verification operation, the additional communication overhead of our scheme is 1480Byte (about 1.45KB) for each block. For one block update operation, the communication overhead is about 6KB. Both these overheads are very small and acceptable in practice.

6.6 Scheme Comparisons

We compare our scheme with some other existing well-known schemes in Table 4 on the whole. In the Table 4, n is the number of the data blocks, ℓ denotes the size of the client data and λ is the security parameter. As shown, most of existing schemes do not support data dynamics when the data is encoded. And those schemes supporting data dynamics usually only consider the data recovery in single cloud server and the multiple cloud servers or single cloud server failure scenarios are out of consideration. Our proposal takes both the multiple cloud servers and single cloud server failure into consideration and supports the encoded data dynamics. The update complexity of our scheme is logarithmic as the update complexity of rb23Tree is logarithmic and the update only involves the update of node information on the proof path. The length of the proof path is related to the file size(the rb23Tree height). Our scheme also balances the communication cost and storage overhead. We can save 25% communication cost comparing with the schemes using erasure codes as outer code and 50% storage overhead comparing with the scheme adopting replication as data redundancy. Therefore, our scheme achieves more efficient data update and data recovery than most of the existing solutions.

7 Conclusion

In this paper, we proposed a new dynamic proof of retrievability scheme for coded cloud storage systems. Network coding and erasure codes are adopted to encode data blocks to achieve within-server and cross-server data redundancy, tolerating data corruptions and supporting communication-efficient data recovery. By combing range-based 2-3 tree and an improved version of aggregatable signature-based broadcast (ASBB) encryption, our construction can support efficient data dynamics while defending against data replay attack and pollution attack. Security analysis and experimental evaluations demonstrated the practicality of our construction in coded cloud storage systems.

Acknowledgments

This work is supported by the National Nature Science Foundation of China(No.61373169, 61373167, 61103219), the National Basic Research Program of China (973 Program No.2014CB340600) and the Doctoral Fund of the Ministry of Education priority areas of development projects (No.20110141130006).

References

[1] P. Mell and T. Grance, "Draft Nist Working Definition of Cloud Computing," National Institute of Standards and Technology, Tech. Rep. http://csrc.nist.gov/groups/SNS/cloudcomputing/index.html, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A.Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, pp. 50-58, 2010.

[3] Amazon.com, "Amazon S3 Availability Event: July 20, 2008," http://status.aws.amazon.com/s3-20080720.html, July 2008.

[4] S. Wilson, "Appengine Outage," http://www.cioweblog.com/50226711/appengine outage.php, June 2008.

[5] B. Krebs, "Payment Processor Breach May Be Largest Ever," http://voices.washingtonpost.com/securityfix/2009/01/payment_processor_breach_may_b.html, 2009.

[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable Data Possession at Untrusted Stores," *Proc. 14th ACM Conf. Computer and Comm. Security (CCS'07)*, pp. 598-609, 2007.

[7] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling Public Verifiability and Data Dynamic for Storage Security in Cloud Computing," *Proc. 14th European Symp. Research in Computer Security (ESORICS'09)*, pp. 355-370, 2009.

[8] A. Juels and B.S. Kaliski, "PORs: Proofs of Retrievability for Large Files," *Proc. 14th ACM Conf. Computer and Comm. Security (CCS'07)*, pp. 584-597, 2007.

[9] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia, "Dynamic Provable Data Possession," *Proc. 16th ACM Conf. Computer and Comm. Security (CCS'09)*, pp. 213-222, 2009.

[10] H. Shacham and B. Waters, "Compact Proofs of Retrievability," *Proc. 14th Int'l Conf. Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT'08)*, pp. 90-107, 2008.

[11] C, Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing," *Proc. 29th Annual IEEE Int'l Conf. Computer Comm. (INFOCOM'10)*, pp. 525-533, 2010.

[12] Y. Zhu, H. Wang, Z. Hu, G.J. Ahn, H, Hu, and S.S. Yau, "Cooperative Provable Data Possession," Report 2010/234, Cryptology ePrint Archive, 2010.

[13] Y. Zhu, H. Wang, Z. Hu, G.J. Ahn, H. Hu, and S.S. Yan, "Dynamic Audit Services for Integrity Verification of Outsourced Storages in Clouds," *Proc. 26th Symp. Applied Computing (SAC'11)*, pp.1550-1557, 2011.

TABLE 3
Verification cost of proposed scheme compared with Ref [10] and Ref [11].

	Proposed scheme		Ref [10]		Ref [11]	
Sampled blocks c	300	460	300	460	300	460
Path generation (ms)	1.093	1.172	N/A	N/A	N/A	N/A
Path verification (ms)	5.431	8.325	N/A	N/A	N/A	N/A
$ChalGen$ (ms)	447.575	696.541	238.179	345.029	231.799	352.257
$ProofGen$ (ms)	223.481	347.165	225.892	352.293	234.018	349.738
$VerifyProof$ (ms)	1.435	1.435	236.295	352.514	266.724	368.085

TABLE 4
Scheme comparisons.

	Within-server encoding	Cross-server encoding	Supporting data dynamics	Update complexity	*Repair traffic factor*
Proposed scheme	erasure code	network coding	Yes	$O(logn)$	1.5
Ref [10]	erasure code	N/A	No	N/A	N/A
Ref [21]	erasure code	erasure code	No	N/A	2
Ref [22]	N/A	network coding	No	N/A	1.5
Ref [25]	N/A	LT code	No	N/A	1
Ref [26]	erasure code	network coding	No	N/A	1.5
Ref [27]	erasure code	N/A	Yes	$O(\lambda^2 \times log^2 \ell)$	N/A
Ref [15]	erasure code	N/A	Yes	$O(logn)$	N/A

[14] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote Data Checking Using Provable Data Possession," *ACM Transactions on Information and System Security*, vol. 14, no. 1, pp. 12-34, 2011.

[15] Q. Zheng and S. Xu, "Fair and Dynamic Proofs of Retrievability," *Proc. First ACM Conf. Data and Application Security and Privacy (CODASPY'11)*, pp. 237-248, 2011.

[16] S. Wang, D. Chen, Z. Wang, and S. Chang, "Public Auditing for Ensuring Cloud Data Storage Security with Zero Knowledge Privacy," Report 2012/365, Cryptology ePrint Archive, 2012.

[17] B. Chen, and R. Curtmola, "Towards Self-Repairing Replication-Based Storage Systems Using Untrusted Clouds," *Proc. third ACM Conf. Data and Application Security and Privacy (CODASPY'13)*, pp. 377-388, 2013.

[18] Q. Wu, Y. Mu, W. Susilo, B. Qin, and J.D. Ferrer, "Asymmetric Group Key Agreement," *Proc. 28th Annual Int'l Conf. Theory and Applications of Cryptography Techniques (EUROCRYPT'09)*, pp. 153-170, 2009.

[19] Y. Hu, H.C.H. Chen, P.P.C. Lee, and Y. Tang, "NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds," *Proc. 10th USENIX Conf. File and Storage Technologies (FAST'12)*, pp. 265-273, 2012.

[20] H.P. Anvin, "The Mathematics of RAID-6," http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf, 2007.

[21] K.D. Bowers, A. Jules, and A. Oprea, "HAIL: A High-Availability and Integrity Layer for Cloud Storage," *Proc. 16th ACM Conf. Computer and Comm. Security (CCS'09)*, pp. 187-198, 2009.

[22] B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote Data Checking for Network Coding-Based Distributed Storage Systems," *Proc. 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW'10)*, pp. 31-42, 2010.

[23] Y. Zhu, H. Wang, Z. Hu, G.J. Ahn, and H. Hu, "Zero-knowledge Proofs of Retrievability,", *Science China: Information Sciences*, vol. 54, no. 8, pp. 1608-1617, 2011.

[24] E. Stefanov, M.V. Dijk, A. Oprea, and A. Jules, "Iris: A Scalable Cloud File System with Efficient Integrity Checks," Report 2011/585, Cryptology ePrint Archive, 2011.

[25] N. Cao, S. Yu, Z. Yang, W. Lou, and Y.T. Hou, "LT Codes-based Secure and Reliable Cloud Storage Service," *Proc. 31st Annual IEEE Int'l Conf. Computer Comm. (INFOCOM'12)*, pp. 693-701, 2012.

[26] H.C.H. Chen and P.P.C. Lee, "Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage," *Proc. 31st Int'l Symp. Reliable Distributed Systems (SRDS'12)*, pp. 51-60, 2012.

[27] D. Cash, A. Kupcu, and D. Wichs, "Dynamic Proofs of Retrievability via Oblivious RAM," *Proc. 32nd Annual Int'l Conf. Theory and Applications of Cryptographic Techniques (EUROCRYPT'13)*, pp. 279-295, 2013.

[28] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," *Journal of Cryptology*, vol. 17, no. 4, pp. 297-319, 2004.

[29] I.S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathmatics*, vol. 8, no. 2, pp. 300-304, 1960.

[30] A.G Dimakis, P.B. Godfrey, Y. Wu, M.J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Transaction on Information Theory*, vol. 56, no. 9, pp. 4539-4551, 2010.

[31] D. Boneh and C. Gentry, "Aggregate and verifiably encrypted signatures from bilinear maps," *Proc. of International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt'03)*, pp. 416-432, 2003.

Zhengwei Ren received his Ph.D degree from Wuhan University, Wuhan, China, in 2014. His research interests are in the areas of applied cryptography and information security, with current focus on data security in cloud computing.

Lina Wang is a professor in Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, and School of Computer, Wuhan University, Wuhan, China. She received the Ph.D. degree in computer science from the Northeastern University, Shenyang, China, in 1999. Her research interests include information security and applied cryptography, with the current focus on security and privacy in cloud computing.

Qian Wang received the B.S. degree from Wuhan University, China, in 2003, the M.S. degree from Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, China, in 2006, and the Ph.D. degree from Illinois Institute of Technology, USA, in 2012, all in Electrical Engineering. Currently, he is a faculty member with the School of Computer Science, Wuhan University. His research interests include wireless network security and privacy, cloud computing security, and applied cryptography. He is a co-recipient of the Best Paper Award from IEEE ICNP 2011. He is a Member of the IEEE and a Member of the ACM.

Rongwei Yu is a lecturer in School of Computer, Wuhan University, Wuhan, China. He received his Ph.D degree from Wuhan University in 2009. His current interests are in the areas of virtualization security and network security.

Ruyi Deng received his B.E degree from Huazhong University of Science and Technology, Wuhan, China, in 2012. He is currently a master student in School of Computer at Wuhan University.