Journal of Ambient Intelligence and Humanized Computing (2021) 12:9127–9143
https://doi.org/10.1007/s12652-020-02610-x

ORIGINAL RESEARCH

Reliable verification of distributed encoded data fragments in the cloud

Vikas Chouhan[1] · **Sateesh K. Peddoju**[1]

Received: 30 March 2020 / Accepted: 10 October 2020 / Published online: 3 November 2020

Abstract
Cloud storage services allow users to remotely store their data in a distributed environment and enjoy the cloud applications ubiquitously. To maximize users' trust, it also integrates a verification mechanism that guarantees the stored data's correctness. The storage application fragments the user data and stores them on multiple cloud storage servers. However, it suffers from expensive data aggregation computations while processing verification services, and inevitably poses a data integrity verification challenge. To avoid these expensive computations, we simplify the verification procedure without needing the data aggregation, just by storing the evidence fragments and data fragments across the datacenters. In distributed environments, the storage correctness verification mechanism depends on the availability of storage servers. Therefore, the challenge of proof/evidence availability may arise due to a server failure or data corruption, hence, decreasing the reliability of storage correctness verification. Thus, the problem of proof reliability is introduced over the distributed data. A few techniques proposed in the literature provide the data reliability; however, none of these existing works have considered the proof reliability to the best of our knowledge. To address the new issue of proof reliability, in this paper, we utilize and leverage the Erasure Coding (EC) to propose a reliable storage correctness verification solution that guarantees the retrieval of evidence and minimizes the effect of server failure/unavailability. The experimental results demonstrate that the proposed approach achieves reliability even after the loss of a certain number of fragments, ranging between 2 and 12 depending upon the number of parity fragments used in the EC scheme. Extensive experiments are performed in real-time, and results show that our proposed solution is highly efficient than well-known state-of-the-art verification schemes.

Keywords Cloud computing · Cloud storage · Data auditability · Reliability · Third-party auditor · Erasure coding · Data integrity · Cloud datacenter

1 Introduction

Cloud computing (Mell and Grance 2011) rapidly provisions ubiquitous services as per the user requests via the Internet. However, the multi-tenancy concept of the cloud allows resources to be shared, leading to new security and privacy concerns (Khan 2016; Singh and Chatterjee 2017). Therefore, various insider and outsider attacks (Behl 2011; Gřivna and Drápal 2019; Tabrizchi and Rafsanjani 2020; Punitha and Indumathi 2020) occur in the cloud. These attacks can alter stored data integrity. Hence, the user requires a verification mechanism to validate the correctness of the stored data. Today, we are surrounded by many devices (i.e., IoT Sensors, Mobile, Laptop) that generate a massive amount of data in real-time. These data can be outsourced by the client to enjoy the ubiquitous services of the cloud. Cloud utilizes resource optimization and allocation policies to enhance its services and reduce the managing cost. In recent years, the work in (Sangaiah et al. 2019a, b) targets to reduce energy consumption while processing and transmitting the data. The authors in work (Sangaiah et al. 2019a) introduce the energy-aware solution to achieve information confidentiality for cyber-physical security. On the other hand, our work is relatively towards the integrity verification of the stored data and minimizing the impact of server failures. The storage systems may experience the unavailability of customer's data and proof fragments mostly during peak times and may

✉ Vikas Chouhan
vchouhan@cs.iitr.ac.in

Sateesh K. Peddoju
sateesh@ieee.org

[1] Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, Roorkee, India

lead to customer loss, which may subsequently impact the business metrics. We believe that any cloud system trust enhances if CSP deploys a reliable verification mechanism to maximize the customer's trust and minimize the impact of server failures. For the expansion and leveraging of their business and improving user's trust, CSP should promise to incorporate the reliable verification mechanism in the storage service, which helps in data validation. Several related works in the literature explain the audit mechanism (Schwarz and Miller 2006; Wang et al. 2010, 2011, 2013, 2015; Yuchuan et al. 2014; Zhang et al. 2016; Hou et al. 2019; Premkamal et al. 2019; Jayaraman and Panneerselvam 2020) that employ the third-party to verify the correctness of stored data. Few other works on data correctness verification are based on signature matching mechanism (Ateniese et al. 2007; Bowers et al. 2009b; Wang 2015). These schemes can verify the correctness of stored data, but they are incapable of providing proof reliability when servers fail or data corrupt. Therefore, we focus on the reliability of the proof fragments across distributed servers.

Cloud Service Providers (CSPs) such as Microsoft, Google, Facebook deploy the Erasure Coding (EC) (Plank et al. 2009; Li et al. 2016; Chouhan and Peddoju 2020) technique, and that is better than 3-replication technique (Weatherspoon and Kubiatowicz 2002; Li et al. 2016). The purpose of using EC is to achieve data consistency, availability, and reliability. Therefore, the storage systems such as Windows Azure Storage (WAS) (Huang et al. 2012), Google's Colossus (successor of Google File System) (Ghemawat et al. 2003), Facebook's HDFS (Hadoop's Distributed File System) (Thusoo et al. 2010), Hadoop Adaptively Coded Distributed File System (HACFS) (Xia et al. 2015), Hitchhiker (Rashmi et al. 2014), and IBM's Parallel File System (GPFS) (Schmuck and Haskin 2002) have agreed on erasure coding concept.

It is important to note that the CSPs will lose the customers and their trust if they do not provide a reliable verification mechanism. Hence, achieving high availability and reliability are the prime concerns in the distributed environment; therefore, the study in this paper is important.

As the data size grows big, huge sized data fragments are stored across the servers. During verification, these stored fragments are required to generate the response against the received verification request. Thus, the current verification schemes suffer from expensive aggregation and computation problems. Therefore, to avoid the burden of data aggregation and expensive computation operations, we proposed a mechanism that simplifies the evidence extraction procedure without data fragments aggregation.

To meet the reliability goal, we utilize and leverage the EC technique. This technique achieves the reliability by reconstructing the *original data* and *proof* even if some servers are unavailable or few fragments are corrupted. Thus, we create the data and proof fragments and then distribute them to multiple storage servers to achieve availability and reliability. In this paper, we use the words *proof* and *signature* interchangeably.

1.1 Motivation

In the cloud environment, CSPs distribute and store the encoded data fragments across the datacenters, which inevitably pose a new data integrity verification challenge. However, the users are not aware of their data because these encoded fragments are not stored as it is in a single place. Hence, the EC technique brings the auditability challenge over the distributed secure encoded fragments. In the last few years, limited erasure code-based verification studies, e.g., Schwarz and Miller (2006), Wang et al. (2011), Zhang et al. (2016), Li et al. (2017) and Vasilopoulos et al. (2018, 2019), have been conducted. Among them, the works such as Schwarz and Miller (2006), Wang et al. (2011) and Vasilopoulos et al. (2018, 2019) have considered the data reliability, but they did not consider proof reliability and require expensive computations during response generation. Therefore, the existing techniques are not appropriate for reliable auditing against such dispersed secure encoded fragments, leading to the problem of integrity verification. Hence, it is necessary to provide an auditing mechanism in the cloud due to the lack of reliable auditing techniques over distributed secure encoded fragments, which is the design objective of our proposed approach.

1.2 Contributions

To provide reliable storage correctness verification in the cloud environment, we design our own challenge-and-response queries according to the proposed framework that verifies the integrity of remote data. In this paper, we made the following contributions:

- We propose a novel reliable verification technique for distributed encoded data fragments using the Erasure Coding (EC) technique. It simplifies verification operation by avoiding aggregation of data fragments. To achieve reliability, the EC technique enables support in retrieving back the original data and proof even if some fragments are lost. We also demonstrate the secure data storing procedure to the Cloud Storage Servers (CSS) via the compute server.
- We describe a concrete construction of proof retrievability based on EC. The proposed approach determines the signature and uses the EC technique to generate the data and signature fragments. Then distributes them to the distinct storage servers. Our proposed solution uses the third-party to audit the stored data validity over distrib-

uted encoded fragments without downloading the stored data fragments.
- We perform extensive experiments in real-time and discuss the performance analysis of operations, such as challenge/response generation, encoding, decoding, reliability, and proof extraction, followed by the comparative analysis with various well-known state-of-the-art verification schemes.

1.3 Organization

The rest of the paper is organized as follows. Preliminaries are defined in Sect. 2. In Sect. 3, we discuss the erasure coding mechanism. Related work is summarized in Sect. 4. In Sect. 5, we present the system model and design goals. The proposed scheme is presented in Sect. 6. The implementation and evaluations are shown in Sect. 7. Finally, we conclude the paper in Sect. 8.

2 Preliminaries

In this section, we list the notations used throughout the paper, shown in Table 1, and then describes the primitives used in our proposed work.

2.1 Security functions and file operations

This subsection defines the set of functions that are used in security and file operations.

***KeyGen*(1^λ)** This algorithm takes a security parameter $(1^\lambda | \lambda \in \mathbb{N}$, *i.e.*, sequence of 1's) as an input. This probabilistic algorithm works as per the user requirements. It generates the secret key $\{S_k\}$ in the case of symmetric encryption and generates a public and private key pair $\{P_k, S_k\}$ in the case of asymmetric encryption.

***C* ⟵ *E*(k, η)** This encryption algorithm takes input as the key k and file contents η, and generates the corresponding ciphertext C.

***η* ⟵ *D*(k, C)** This decryption algorithm takes input as the key k and ciphertext C, and outputs the file contents η.

***τ* ⟵ *TagGen*(η)** This algorithm maps the input content η, and outputs a fixed length tag τ.

***σ* ⟵ *GenSig*(η)** This algorithm runs on both client and compute server side. It takes data as an input, and generates the corresponding signature $\sigma = \{0, 1\}^*$.

***δ* ⟵ *ComFrag*(η)** This function runs at both client and compute server, that takes an input data, and then returns their fragments. The number of fragments in the set depends on the Erasure Code encoding parameter (α, β).

***χ* ⟵ *GetEvc*(η)** This function runs at client, which takes data as an input, and then it computes and returns the corresponding metadata χ. The metadata computation process first calls the $GenSig(\eta)$ algorithm to compute the corresponding signature σ. Then it calls $ComFrag(\sigma)$ function to obtain the set of signature fragment δ_σ. Further, it chooses any one random fragment $\delta_{\sigma_{x^{th}}} \in \delta_\sigma$ as an evidence, called metadata χ.

***PUT*$_x(\eta_f, \delta'_i)$** This function runs at compute server, which stores the ith data member $\delta'_i \in \delta'$ with file name η_f to the corresponding storage server x, where x is a member of the available storage server set S_L.

***δ*$_{\sigma_x}$ ⟵ *GetSig*$_x(\eta_f)$** This function runs at compute server, which takes file name η_f as an input.

This function calls the Cloud Storage Server (CSS) to obtain a stored signature fragment (δ_{σ_x}) from the storage server x.

***Validate*(χ, δ_σ)** This function, invoked by TPA, takes input as the metadata χ and a set of signature fragments δ_σ. It returns accept status if χ belongs to the subset of δ_σ or reject otherwise.

Table 1 Notations

Symbol	Description	Symbol	Description
η_f	Filename	δ	Set of fragments
η	File content	ℓ	Length of one data fragment
τ	Tag	σ	Signature
C	Ciphertext	δ_σ	Set of signature fragments
C_η	Ciphertext of file content	C_σ	Encrypted signature
η_{ch}	Challenge	$\delta'_{i^{th}}$	Concatenation of $\delta_{i^{th}}$ and $C_{\sigma_{i^{th}}}$
k	Key	U_{id}	User ID
S_k	Secret key	T_{id}	TPA ID
P_k	Public key	α	Number of data fragments
χ	Metadata	β	Number of parity fragments
S_L	Set of available storage server	γ	Total fragments, $\alpha + \beta$

3 Background: erasure coding (EC)

In this section, we discuss the EC mechanism that is required to achieve the reliability.

The cloud datacenters ensure the reliability of stored data and offer the least affected services even in the unavailability of a certain number of storage servers. The traditional storage system uses replication technique to provide reliability. It is too expensive to use the replication system for the big file in terms of storage and performance (Li et al. 2016). Therefore, Erasure Coding (EC) (Plank et al. 2009; Li et al. 2016; Chouhan and Peddoju 2020) came into the light because of its fault tolerance characteristics with least storage cost (Weatherspoon and Kubiatowicz 2002). The process of data encoding and decoding is shown in Fig. 1. EC encoder partitions the input data into α parts, and then it generates a set of encoded fragments $\delta = \{\delta_1, \delta_2, \delta_3, \ldots, \delta_\gamma\}$ where $\gamma > \alpha$. Each encoded fragment contains the portion of original data. These encoded fragments consists of a subset with α number of data fragments $\delta^d = \{\delta_1^d, \delta_2^d, \delta_3^d, \ldots, \delta_\alpha^d\}$ and the subset with β number of parity fragments $\delta^p = \{\delta_1^p, \delta_2^p, \delta_3^p, \ldots, \delta_\beta^p\}$. This scheme takes only 1/r additional storage space to offer reliability if the encoding rate r $(= \alpha/\gamma)$ is less than one. EC decoder is used to reconstruct the actual data from these encoded fragments. Any subset with $\alpha \in \delta$ number of fragments is sufficient to reconstruct the actual data. It can tolerates the loss up to β number of fragments (Li et al. 2016). Reed–Solomon (RS) (Reed and Solomon 1960) code is one of the well-known EC technique that is deployed in datacenters. This technique offers significantly better reliability than the replication system (Li et al. 2016). The coding scheme is represented as $RS(\alpha, \beta)$. RS code applies $\gamma \times \alpha$ generator matrix of w *bit* words which performs *XOR* and multiply operations in a Galois field with 2^w different symbols ($GF(2^w)$) (Reed and Solomon 1960).

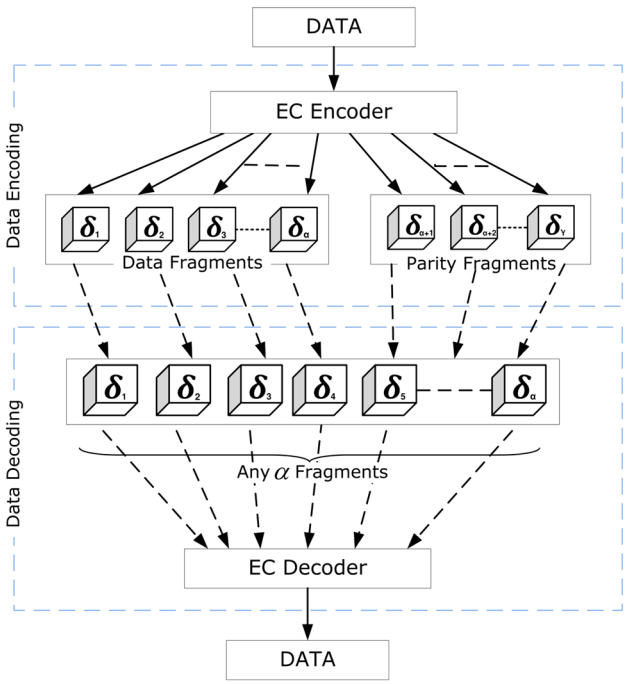


Fig. 1 Process of the data encoding and decoding

In the proposed approach, we use the *Reed–Solomon* erasure coding mechanism based on the input sequence of α and β values to achieve the reliability, error recovery, and fault tolerance of the data. We achieve the reliability by partitioning the data and proof into the fragments to tolerate the loss/corruption of a few fragments. This process increases data availability and reliability. We evaluate the probability of the stored data availability, denoted by P_a, using Eq. 1 (Weatherspoon and Kubiatowicz 2002), where M and N denote the number of currently unavailable servers and the total number of servers, respectively,

$$\textit{Probability of Data Availability}, \quad P_a = \sum_{i=0}^{\gamma-\alpha} \frac{\binom{M}{i}\binom{N-M}{\gamma-i}}{\binom{N}{\gamma}}. \tag{1}$$

4 Related work

In this section, we review and analyze the existing works that aim to perform the verification of the remote storage data. We discuss the related works in these categories: Provable Data Possession (PDP) based verification, Proof of Retrievability (PoR) based verification, and Third-Party Auditor (TPA) based verification.

4.1 Provable data possession based verification

In the recent years, many works in the literature proposed the verification techniques of stored data in the cloud (Ateniese et al. 2007; Bowers et al. 2009b; Wang et al. 2013, 2015; Wang 2015; Jayaraman and Panneerselvam 2020). All these techniques verify the data integrity on untrusted storage without downloading the entire file. Ateniese et al. (2007) and Wang (2015) discussed the PDP scheme. This scheme pre-processes the data before uploading, and it allows the user to verify the server possession. However, it does not guarantee the data retrievability. Recently, Jayaraman and Panneerselvam (2020) introduced a privacy-preserving PDP-based integrity checking framework for cloud data that supports many functions such as public auditing, an infinite number of audits, and data confidentiality. In the case of distributed systems, the PDP scheme cannot verify

the storage integrity due to the failure or unreachability of some servers.

4.2 Proof of retrievability based verification

Juels and Kaliski (2007) and Bowers et al. (2009b) discussed the PoR technique to verify the correctness of stored cloud data. This technique uses sentinels (random-valued check blocks) to enable spot-checking and it uses error-correcting code technique to recover the file. Bowers et al. (2009a) proposed a distributed cryptographic system called High-Availability and Integrity Layer (HAIL). This protocol employs the Test and Redistribute (TAR) strategy and targets integrity check using the PoRs scheme. However, PDP or PoR schemes alone are incapable of providing data reliability and recovery assurance. In other words, these PoR based verification schemes do not directly support the integrity verification of dispersed encoded fragments across the CSS because it does not guarantee the data availability during server failures. The mentioned schemes can verify the correctness of stored data, but they fail to provide proof reliability guarantees when servers failure or data corruption occurs.

4.3 Third-party auditor based verification

Several approaches have been proposed in the literature that employ a third-party (e.g., Schwarz and Miller 2006; Ateniese et al. 2007; Wang et al. 2010, 2011, 2016, 2020; Yuchuan et al. 2014; Zhang et al. 2016; Yu et al. 2016; Garg and Bawa 2017; Sookhak et al. 2017; Li et al. 2017; Vasilopoulos et al. 2019; Premkamal et al. 2019) for data integrity verification. Wang et al. (2012) introduced a lightweight cloud auditing mechanism. It includes components such as data dispersal, token pre-computation, error localization, dynamic update to correct the data error and encounter the faulty server. This scheme uses a *Reed–Solomon* coding scheme and homomorphic tokens to provide data integrity and availability. Further, Wang et al. (2013) introduced a TPA scheme to verify the integrity of outsourced data in the cloud. TPA eliminates the integrity checking and verification overhead from customers' end. The third-party enables trust in a cloud environment by establishing an agreement between customers and providers. Besides, Wang et al. (2011) introduced public auditing mechanism for dynamic data by manipulating the classic Merkle Hash Tree for block tag authentication. Moreover, they explored the bilinear aggregate signature (Wang et al. 2011) to allow TPA to perform the multiple auditing operations simultaneously. It considered TPA to perform integrity verification of storage data which differs from traditional PDP (Ateniese et al. 2007; Wang 2015) and PoR (Juels and Kaliski 2007; Bowers et al. 2009b) models. The author used the error-correcting code to tolerate failures in storage.

Later, Wang et al. (2016) introduced comprehensive auditing and an identity-based integrity verification solution to eliminate certificate management overhead. Sookhak et al. (2017) introduced identity-based data auditing solutions along with data privacy preservation. They also reduced the system and managing the cost of the authentication framework by using key-homomorphic cryptographic primitive. Subsequently, Garg and Bawa (2017) presented an auditing approach based on Relative Index and Time Stamped (RITS) and Merkle Hash Tree (MHT). They reduced searching complexity and guaranteed the freshness of data. The work in Sookhak et al. (2017) introduced the Divide and Conquer Table based data structure and utilized algebraic properties to support data auditing for big data. However, this scheme does not support distributed servers. To deal with shared data, Gudeme et al. (2020) introduced an attribute-based integrity verification mechanism for shared data in the cloud environment. They simplified the key management by using a unique public key for integrity verification. Besides, Wang et al. (2020) introduced a blockchain-based private PDP mechanism to provide distributed data integrity verification in the cloud storage.

Recently, few studies in Zhang et al. (2016), Li et al. (2017) and Vasilopoulos et al. (2018, 2019) demonstrated data auditability using erasure codes. Zhang et al. (2016) used an indistinguishability obfuscation mechanism to verify the data integrity and reduced computation overhead for auditors. Later, Li et al. (2017) introduced a fuzzy identity-based data verification scheme with error-tolerance properties. For their work, they used the biometric-based identity. Vasilopoulos et al. (2018), first proposed a data reliability solution and integrity verification of cloud data by exploiting PDP with time-constrained operations. They affixed redundant information with data to provide recovery from data corruption. Later, they upgraded their solution in Vasilopoulos et al. (2019) for distributed storage using a time-lock puzzle to guarantee data reliability.

In summary, we reviewed the most relevant literature that verifies the data integrity of remote storage. Additionally, we compared several existing works in Table 2 based on various parameters such as public auditing, auditing entity, proof reliability over distributed fragments, encoding scheme, auditability over distributed EC fragments, number of verification, data integrity, data recovery support, reduced server dependency, maintaining the confidentiality of data, privacy preservation, and data reliability. Only a few authors in Schwarz and Miller (2006), Wang et al. (2011) and Vasilopoulos et al. (2019) incurred the auditability over EC fragments. However, none of them have addressed the issue of proof reliability over distributed storage. The existing works in Wang et al. (2011) and Vasilopoulos et al. (2018, 2019)

Table 2 Functionality comparison of verification schemes

Scheme	1	2	3	4	5	6	7	8	9	10	11	12
Schwarz and Miller (2006)	Y	U/TPA	N	EC	Y	INF	Y	Y	–	N	–	Y
Ateniese et al. (2007)	Y	U/TPA	N	–	N	INF	Y	N	N	N	N	N
Ateniese et al. (2008)	N	U	N	–	N	FIN	Y	N	N	–	Y	N
Erway et al. (2009)	N	U	N	–	N	INF	Y	N	N	N	N	N
Wang et al. (2010)	Y	TPA	N	–	N	INF	Y	N	N	N	Y	N
Wang et al. (2011)	Y	U/TPA	N	EC	Y	INF	Y	Y	Y	–	N	Y
Chen (2013)	N	U	N	–	N	INF	Y	N	N	N	N	N
Chen et al. (2013)	N	U	N	–	N	INF	Y	N	N	N	N	N
Yuchuan et al. (2014)	Y	U/TPA	N	–	N	INF	Y	N	N	N	N	N
Yu et al. (2014)	N	U	N	–	N	INF	Y	N	N	–	N	N
Yu et al. (2015)	N	U	N	–	N	INF	Y	N	N	N	N	N
Zhang et al. (2016)	Y	TPA	N	EC	N	INF	Y	N	–	N	Y	N
Wang et al. (2016)	Y	TPA	N	–	N	INF	Y	N	N	N	N	N
Yu et al. (2016)	Y	U/TPA	N	–	N	INF	Y	–	N	Y	Y	N
Garg and Bawa (2017)	Y	U/TPA	N	–	N	INF	Y	N	N	N	N	N
Li et al. (2017)	Y	U/TPA	N	EC	N	INF	Y	N	–	N	N	N
Sookhak et al. (2017)	Y	U/TPA	N	–	N	–	Y	N	N	–	Y	N
Vasilopoulos et al. (2018)	N	U	–	MDS	N	–	Y	Y	Y	–	–	Y
Vasilopoulos et al. (2019)	Y	U/TPA	–	MDS	Y	–	Y	Y	Y	–	–	Y
Jayaraman and Panneerselvam (2020)	Y	TPA	N	–	N	INF	Y	N	N	Y	Y	–
Ours	Y	U/TPA	Y	RS	Y	INF	Y	Y	Y	Y	Y	Y

Labels 1–12 represent the following properties: (1) public auditing, (2) auditing entity, (3) proof reliability over distributed fragments, (4) encoding scheme, (5) auditability over distributed EC fragments, (6) no. of verification, (7) data integrity, (8) data recovery support, (9) reduced server dependency, (10) maintain confidentiality of data, (11) privacy preservation, (12) data reliability

Abbreviations are defined as follows: *U* user/client, *TPA* third-party auditor, *FIN* finite, *INF* infinite, *EC* erasure coding, *RS* Reed–Solomon, *MDS* maximum distance separable, *Y* Yes, *N* No, and "–" means not mentioned

have been able to reduce the server requirements while downloading the data. None of the existing works, except Schwarz and Miller (2006), Wang et al. (2011) and Vasilopoulos et al. (2018, 2019), incurred data reliability, and recovery support.

The discussed solutions in the literature mainly focus on TPA-based verification techniques, and a few of them consider the reliability mechanism for data only. However, our scheme focuses on the reliability of the proof fragments across distributed servers. Thus any of the existing schemes are not directly comparable to the proposed work. Therefore, we present the functionality based comparison in Table 2. Besides, we present a complexity-based comparison in Sect. 7.2.6.

To the best of our knowledge, the existing literature has not explored towards the reliability of stored proof or evidence. Incited by the aforementioned discussions, the proposed scheme focuses on proof reliability. Moreover, the proposed scheme provides storage correctness assurance and recoverability, even in case of the unavailability of a few data fragments or storage servers.

5 System model and design goals

In this section, we present the system model that describes the proposed scheme. Then we present the design goals.

5.1 System model

A representative system model of the proposed work is illustrated in Fig. 2. The model consists of the following entities:

Client/User (*U*): an entity that uses the cloud data storage services. It initiates data auditing, uploading, downloading, and deleting requests. The user can be an individual client or an enterprise.
Third-Party Auditor (*TPA*): a trusted entity that verifies the integrity of stored user data on cloud storage servers.
Compute Server (*CS*): an entity that manages the clients' data operations and handles TPA audit requests. It processes the received user data, and then it computes the

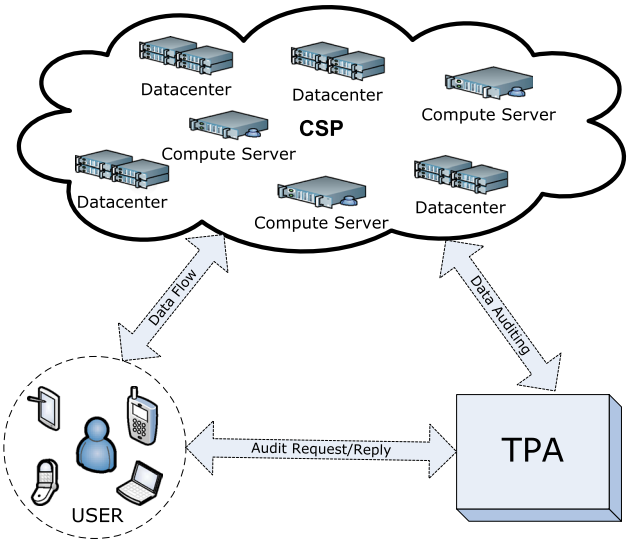


Fig. 2 System model of the proposed work

corresponding fragments using EC. These fragments are uploaded to the distinct CSS's.
Cloud Datacenters: an entity which is operated by CSP to store/retrieve user data.

The proposed scheme focuses on the verification mechanism rather than security aspects. Thus, we assume that the underneath communication between entities is secured via IPSec or SSL/TLS.

5.2 Design goals

We aim to design an efficient data verification mechanism to achieve the following goals:

Storage Correctness Assurance: The proposed approach aims to ensure the correctness of dispersed secure encoded fragments across the cloud datacenters. It enables the third-party verification mechanism. Subsequently, TPA verifies the integrity of stored data at any time or periodically.
Error Correction and Fault Tolerance: The proposed framework achieves data and signature availability and reliability. It refers to the scenario where the verification process should be unaffected even if some servers are down. In addition, the compute server repairs the stored data/signature when it finds an error during the reconstruction phase.
Lightweight Communication: The verification process exchanges minute evidence messages in the form of signatures rather than actual data between entities.

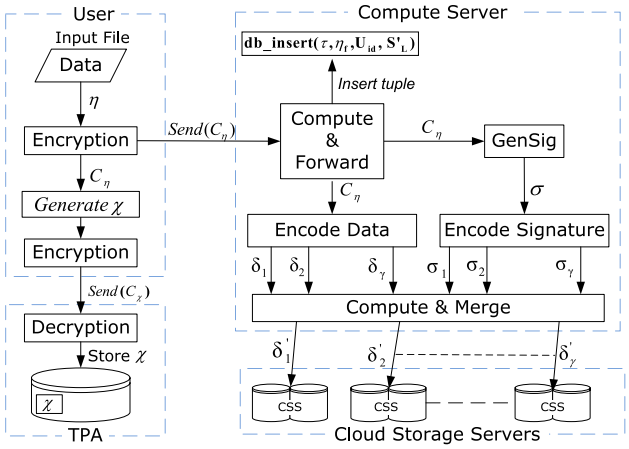


Fig. 3 Data uploading scenario

6 Proposed scheme

In this section, we present an overview of the proposed work, followed by a detailed description of the scheme.

6.1 Overview

This subsection presents an overview of the proposed scheme that offers the stored data verification mechanism using the EC technique. Specifically, we focus on the proof availability and reliability against the unavailability of β fragments or storage nodes.

We also demonstrate the proposed upload and dispersal procedure in order to validate stored encoded fragments during the verification phase. Apart from that, our scheme supports efficient fault tolerance and high availability for both data and proof fragments. It allows the data and proof reconstruction from the minimum number of fragments, thus providing fault tolerance and reliability. TPA and compute server play a crucial role in this verification process. The client and compute server both perform some crucial tasks during the data uploading process to enable reliable auditability over the encoded fragments. Subsequently, we present the data verification procedure where TPA can verify the integrity of the stored user data at any time or periodically. First, we illustrate a file uploading scenario followed by a detailed description of the proposed scheme.

6.1.1 Data uploading scenario

This section demonstrates, through Fig. 3, the file uploading scenario to achieve reliability during the verification phase. The user initiates the uploading process. To achieve confidentiality, first it encrypts the input data η with their own secret key U_{S_k} to get the encrypted data C_η, and then it generates the corresponding metadata χ from C_η. Further, it

encrypts χ using dedicated TPA public key and sends this encrypted metadata with some essential information, i.e., U_{id}, S_{id},and τ to the registered TPA. Meanwhile, the user sends the encrypted data C_η to the compute server. Subsequently, TPA decrypts the received encrypted data using its own private key and then stores this decrypted information for future data validation testing.

The compute server processes the received data to determine the corresponding tag and list of available storage servers. Subsequently, it inserts the essential information, i.e., $\tau, \eta_f, U_{id}, S'_L$ into the compute database. Then it generates the signature from the received data. Compute server performs encoding operation simultaneously for both data and signature. Data encoding module generates the set of encoded data fragments $\{\delta_1, \delta_2, \delta_3, \ldots, \delta_\gamma\}$ corresponding to the received data C_η and the signature encoding module generates the set of encoded signature fragments $\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_\gamma\}$ corresponding to the input signature. Further, the compute server merges the data and signature fragments and then stores them at the distinct available storage servers. It is noted that the proof fragments are stored along with data fragments across the storage servers. Therefore, in response generation against received verification requests, our approach extracts only the proof fragments rather than the data fragments. Thus, it avoids the burden of data fragment aggregation.

Here we achieve the proof reliability using erasure coding. We store the γ number of encoded signature fragments along with the data fragments so that we can recover the actual proof from any α number of signature fragments. It can tolerate up to the β number of corrupted proof fragments.

6.2 Description of the scheme

This subsection presents a comprehensive exhibition of the proposed scheme. First, we discuss the metadata computation procedure, and then we describe the data uploading and dispersal procedure, followed by the discussion on the data verification procedure.

6.2.1 Metadata computation

User preprocesses the encrypted data for the verification purpose. Figure 4 shows the metadata computation process where user generates the secret key S_k via *KeyGen* algorithm. During computation process, it encrypts the data η using the key S_k to get the corresponding ciphertext C_η. Subsequently, it executes the *GenSig* algorithm to compute the signature σ corresponding to the ciphertext C_η. It calls the algorithm *ComFrag* which takes η as an input and generates the set of signature fragments δ_σ. Eventually, user picks a random element $\delta_{\sigma_{xth}}$ from the resultant set δ_σ. This $\delta_{\sigma_{xth}}$ is referred as a metadata χ.

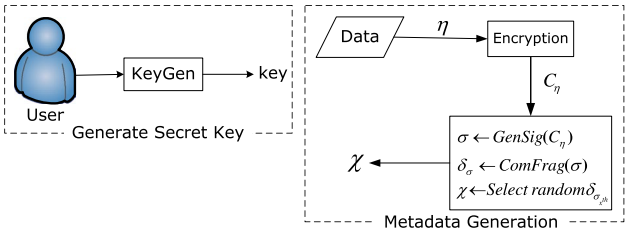


Fig. 4 Preprocess: compute metadata

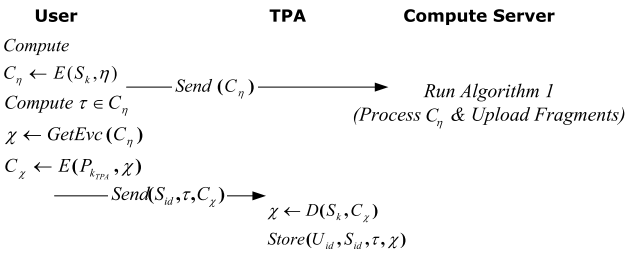


Fig. 5 Operation sequence for uploading and dispersal of data

6.2.2 Data uploading and dispersal procedure

In the data uploading procedure, four entities are primarily involved, including User, Compute Server, Cloud Storage Servers, and TPA. Initially, the client is registered with TPA to get a particular cloud service where TPA has a list of CSPs and corresponding compute servers. Figure 5 shows the operation sequence of uploading and dispersal of user data to ensure auditability over distributed encoded data. The user initiates the data uploading process. First it encrypts the data η with its own secret key S_k to create data ciphertext C_η. This data ciphertext is sent with the upload request to the compute server. Further, it determines the tag τ of C_η by calling *TagGen* algorithm and also extracts the metadata χ by executing the function *GetEvc*. Then it encrypts the metadata χ with the TPA public key $P_{k_{TPA}}$. Subsequently, it sends some essential information with the encrypted metadata C_χ to the registered TPA. At TPA, it determines the metadata by decrypting the received C_χ with its own secret key S_k and then TPA securely keeps this metadata χ. At the compute server, it processes and stores the received encrypted data in a way to provide reliable auditability over the securely encoded data fragments. It executes Algorithm 1, which fragments the received ciphertext C using EC technique and distributes all these encoded fragments across the γ storage servers.

Algorithm 1: Compute and Stores the Fragments to the Datacenters

```
Input  : Ciphertext C
Output : Success/Failure ACK
1  begin
2      σ ⟵ GenSig(C)
3      δ ⟵ ComFrag(C)
4      δ_σ ⟵ ComFrag(σ)
5      Compute τ, η_f
6      S_L ⟵ getAvailSSList()
       /* S_L denotes the list of available storage servers          */
       /* Data fragment uploading starts                             */
7      Let N denotes the number of elements in δ
8      for i = 1 to N do
9          k ⟵ H(δ_i)
10         C_σi ⟵ E(k, δ_σi)
11         δ'_i ⟵ (δ_i || C_σi)
12         status_i ⟵ PUT_SLi(η_f, δ'_i) where S_Li ∈ S_L
           /* S_Li stores the fragment at one of the available storage server
              */
13         S'_L.append(S_Li)
           /* S'_L is the set of storage server addresses where fragments are
              stored                                                 */
14         S_L.remove(S_Li)
15     end
       /* Data fragment uploading ends                               */
16     DB_i(U_id, τ, η_f, S'_L)
17     return status
18 end
```

Algorithm 1 takes the received ciphertext C as an input and returns the status either success or failure. The algorithm performs the following operations:

1. It calls the algorithm *GenSig* which takes ciphertext C as an input and generates the corresponding signature σ.
2. Then it parallelly executes the function *ComFrag* for both C and σ to get the set of data fragments δ and set of signature fragments δ_σ, respectively.
3. Further, it generates the tag and filename corresponding to the received data using *Base*64 or the preferred encoding.
4. Now, the function *getAvailSSList* is executed to get the list of available storage servers, stored in a set S_L.
5. For each element of the set δ, it performs the following operations:
   (a) It computes the hash value from the data fragment δ_i where $\delta_i \in \delta$, using *SHA* hash function. This hash value is used as a key k.
   (b) Then it encrypts the signature fragment δ_{σ_i} where $\delta_{\sigma_i} \in \delta_\sigma$, with key k, to get the resultant ciphertext C_{σ_i}
   (c) Then it appends the computed ciphertext C_{σ_i} with data fragment δ_i to get the new composite coded fragment δ'_i.
   (d) Then it calls the function PUT_{SL_i} to store δ'_i at one of the available storage server $S_{L_i} \in S_L$ with filename η_f for all i where $1 \le i \le \gamma$.
   (e) It appends the storage server address S_{L_i} of the stored fragment to the set S'_L, and then it removes the S_{L_i} entry from the set S_L.
6. The algorithm finally inserts a tuple $\langle U_{id}, \tau, \eta_f, S'_L \rangle$ into the compute database and returns the status as either success or failure. The algorithm returns a failure status if it cannot store all the encoded fragments δ'_i at some storage servers, due to some server failure.

Our proposed procedure encodes the data and signature into the fragments and distributes them across the datacenters to achieve privacy and reliability. Further, it encrypts each signature fragment using the data fragment-based key to secure the signature fragments. Finally, these fragments are used to validate the storage integrity.

6.2.3 Data verification/auditing procedure

This section illustrates the auditing procedure, shown in Fig. 6, to validate the stored data fragments across the datacenters. To audit the stored data integrity, the user sends the audit request to the TPA and gets back its validity status.

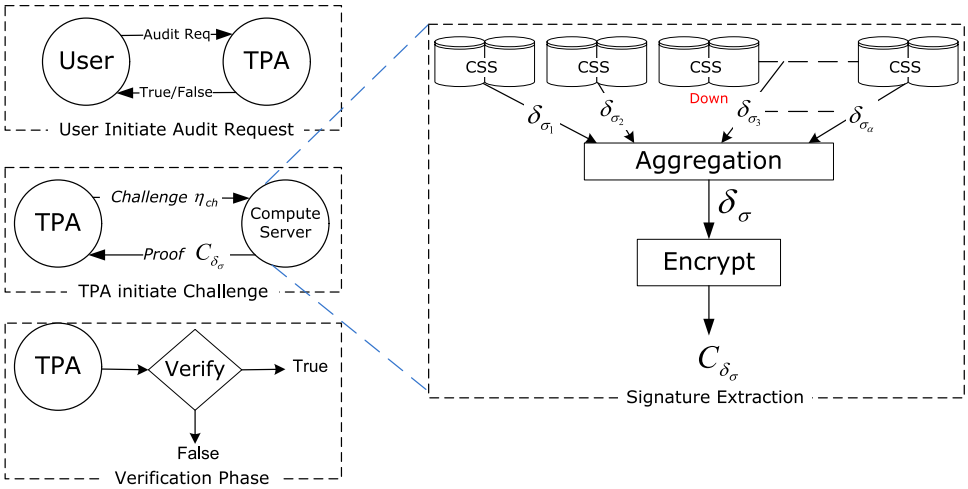


Fig. 6 Verification procedure of erasure fragments

Moreover, TPA periodically performs a validity check to ensure the correctness of the stored data. This verification procedure is divided into two phases, as discussed below.

Phase I: TPA Side Operations
In this phase, TPA plays the role of the verifier. It initiates the challenge η_{ch} request that supplies U_{id}, τ to the compute server and gets back the proof C_{δ_σ} (encrypted signature fragments set). It picks tag τ_i of particular file that belongs to user U_{id} to create the challenge η_{ch}, and further sends it to the compute server and gets back the resulting proof evidence C_{δ_σ}:

$$Challenge,\ \eta_{ch} = \{U_{id} \,||\, \tau_i\}.$$

During validation check, TPA decrypts the received C_{δ_σ} with its own secret key S_k to get the resulting set of fragments δ_σ. Then it attempts to reconstruct the signature σ' from the received fragments set $\delta_\sigma = \{\delta_1, \delta_2, \delta_3, \ldots, \delta_\alpha\}$. This reconstruction process uses the decoding module of erasure code.
Decoding error may occur during the reconstruction process that represents the invalidity of the stored information. To verify the integrity of stored data, it calls the function $Validate(\chi, \delta_\sigma)$ that returns the validity status as either accepted or rejected. The stored data at CSP is valid if decoding operation finishes without any error and metadata χ is equivalent to any fragment of the set δ_σ. If χ doesn't match with any of the fragment belonging to δ_σ, the TPA performs the additional operations. It selects all α fragments from δ_σ and creates a pair of the signature sets, each with $\alpha - 1$ elements, and both the sets must contain one distinct element that is not present in other set. Then the stored metadata χ is inserted into both the sets to get the temporary pair of fragment set ${}_{t1}\delta'_\sigma = \{{}_{t1}\delta'_1, {}_{t1}\delta'_2, {}_{t1}\delta'_3, \ldots, {}_{t1}\delta'_\alpha\}$ and ${}_{t2}\delta'_\sigma = \{{}_{t2}\delta'_1, {}_{t2}\delta'_2, {}_{t2}\delta'_3, \ldots, {}_{t2}\delta'_\alpha\}$. Then it attempts to reconstruct the signature σ''_1 and σ''_2 from the set ${}_{t1}\delta'_\sigma$ and ${}_{t2}\delta'_\sigma$, respectively. To decide the correctness of the stored data, the proposed approach compares σ' and σ''_1. If result is false, then we need to compare σ' and σ''_2. Stored data is corrupted when both the comparisons return the false result; otherwise, stored data is uncorrupted.

Phase II: Compute Server Side Operations
This phase generates a response against the received verification challenge. Upon receiving the challenge η_{ch}, it invokes the Algorithm 2, which extracts the stored signature fragment from any α storage servers, and then it creates a response message corresponding to the received challenge.

Evidence extraction: The evidence extraction procedure for verification is demonstrated using Algorithm 2, which runs at the compute server. This algorithm takes $\eta_f, S'_L, P_{k_{TPA}}$ as input and returns the ciphertext of the extracted signature fragments set C_{δ_σ} as a response. The signature fragment is stored on all the γ number of servers, and α is the minimum number of server access required to get all α fragments. These fragments are sufficient to reconstruct the original signature to achieve proof reliability. Therefore, the signature fragments are retrieved from any α number of storage servers, which belong to the set of available storage servers, S'_L are sufficient to achieve reliability. The following steps are involved in the evidence extraction procedure:

1. For each element in the set S'_L, it performs the following operations in parallel, at α number of storage servers:

(a) It calls the function $GetSig_{S'_{L_i}}(\eta_f)$ to obtain the stored signature fragments δ_{σ_i}. The function $GetSig_{S'_{L_i}}$ fetches the stored signature fragments by executing the Algorithm 3. This is deployed on the storage server nodes.
(b) Then it appends the δ_{σ_i} to the signature fragment set δ_σ.
(c) The count variable keeps track of the number of accessed fragments. If fragments are retrieved from α number of storage servers, the algorithm performs the break operation to exit from the loop and performs the subsequent steps.

2. Further, it encrypts the resulting signature set δ_σ with the public key $P_{k_{TPA}}$ of TPA to get the encrypted signature fragments set C_{δ_σ}.
3. Then, it returns this encrypted signature set C_{δ_σ}.

The CSS node executes Algorithm 3 to retrieve the stored signature fragment. This algorithm takes filename η_f as input and returns the stored signature fragment. The following steps are involved in retrieving the stored signature fragment.

1. First, it calls the function *ReadFile*$(\cdot)$. This function reads the file η_f and returns the file contents η, where η is the stored fragment $\delta'_{x^{th}}$.
2. Then it extracts the data fragment $\delta_{x^{th}}$ and signature fragment $C_{\sigma_{x^{th}}}$ from η.
3. Now, the function $H(\cdot)$ returns the hash value of the input data $\delta_{x^{th}}$. This hash value is used as a key to decrypt the encrypted signature fragment which helps in storage correctness identification.
4. This decrypted signature fragment $\delta_{\sigma_{x^{th}}}$ is returned to the caller.

Algorithm 2: Evidence Extraction for Verification

Input : $Filename\ \eta_f, Server\ List\ S'_L, Public\ Key\ P_{k_{TPA}}$
Output: $Encrypted\ Set\ of\ Signature\ Fragments$

begin
  $count = 0$
  $Let\ N\ denotes\ the\ number\ of\ elements\ in\ S'_L$
  for $i = 1\ to\ N$ **do**
    /* $Run\ parallelly\ for\ any\ \alpha\ out\ of\ \gamma\ fragments$ */
    if $count < \alpha$ **then**
      $\delta_{\sigma_i} \longleftarrow GetSig_{S'_{L_i}}(\eta_f)\ where\ S'_{L_i} \in S'_L$ // $Call\ Algorithm\ 3$
      $\delta_\sigma.append(\delta_{\sigma_i})$
      $count = count + 1$
    end
    else
      $break$
    end
  end
  $C_{\delta_\sigma} \longleftarrow E(P_{k_{TPA}}, \delta_\sigma)$
  $return\ C_{\delta_\sigma}$
end

Algorithm 3: Fetch the Stored Signature Fragment.

Input : $Filename\ \eta_f$
Output: $Signature\ Fragment\ \delta_{\sigma_{x^{th}}}$

begin
  $\eta \longleftarrow ReadFile(\eta_f)$
  $Extract\ \delta_{x^{th}}\ \&\ C_{\sigma_{x^{th}}}\ from\ \eta$
  $k \longleftarrow H(\delta_{x^{th}})$
  $\delta_{\sigma_{x^{th}}} \longleftarrow D(k, C_{\sigma_{x^{th}}})$
  $return\ \delta_{\sigma_{x^{th}}}$
end

During a minor corruption/loss of fragments, The CSP initiates the recovery process. Before sending the TPA response, the compute server attempts to decode δ_σ to ensure that all the fragments are uncorrupted. If the compute server is unable to decode the signature from any α number of fragments, it tries to recover the actual signature from all the γ fragments. Even if it cannot decode and reconstruct the signature from all these γ fragments, it tries to recover the original signature from the stored data fragments. Then it retrieves any α data fragments from the storage servers, decodes them, and tries to reconstruct the stored data. If a decoding error occurs, it requires all γ number of data fragments as input to the decoding module to recover the original data. After recovering the original data, it calls the Algorithm 1 to compute the signature and data fragments, and then stores them to the storage servers. This reconstruction and recovery process enables fault tolerance to the storage.

In this verification phase, the data is secured from the TPA. Suppose, TPA wants to retrieve the stored data from the cloud storage. TPA will not be able to retrieve because it can deal only with the signature fragments. Our scheme retrieves a signature fragment from any α out of γ storage servers, to create the required response. Thus, we achieve proof reliability in constructing a response against the received challenges.

7 Performance evaluation

In this section, we discuss the implementation setup including packages and libraries, followed by the performance analysis and results.

7.1 Implementation details

Experimental setup: The testing environment comprises of Intel®*Core*^{TM} i7-3770 CPU @ 3.40 GHz processor with 8 cores, 8 GB RAM, Ubuntu 14.04 64-bit operating system with Python2.7-dev package, and various Python-based libraries for each entity. Our implementation setup uses the Python language and the MySQL database. We consider the *SHA*-256 secure hash algorithm for generating cryptographic hash value, PyECLib-1.2.0 library with liberasurecode-dev for constructing encoded fragments, and Advanced Encryption Standard (AES) for symmetric-key encryption/decryption algorithms using Python's crypto library. Any customizable storage datacenter can be used for storing and retrieving the data. Our implementation uses Dropbox as a data center, where we create multiple Dropbox accounts. These accounts are used as storage servers to store the encoded fragments.

We implement cloud storage data verification technique over the distributed encoded fragments and perform extensive experiments in real-time. Our implementation comprises multiple modules including User, TPA, Compute Server, and Datacenters. Each module is implemented on a separate machine. The user can request uploading, auditing, downloading, deleting, and viewing the stored files through the designed command-line tool. Further, the command line operations are offered to the users, such as *EOF* to exit from the command prompt, *upload* < *source file* > to upload the source file to the cloud, *ls* command to view the list of the stored filenames, *download* < *file name* > to download the file from the cloud, *audit* < *file name* > to audit the stored file and *delete* < *file name* > to remove the file from the cloud storage.

In our experiments, the compute server performs the CSS's computation jobs because, at present, Dropbox does not allow the users to run the algorithm on the top of their

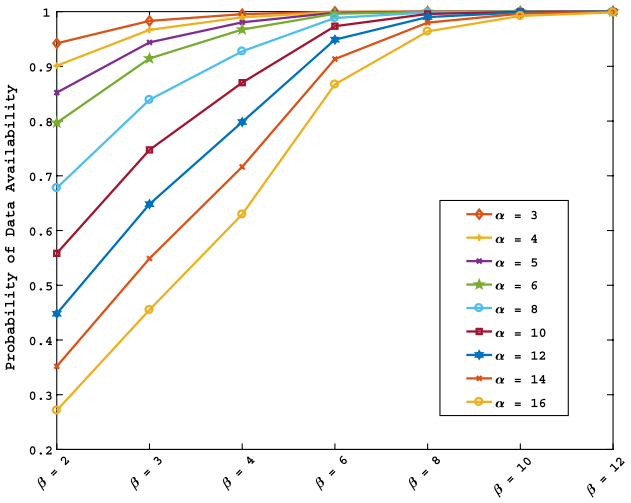


Fig. 7 Reliability achieved with *RS* encoding

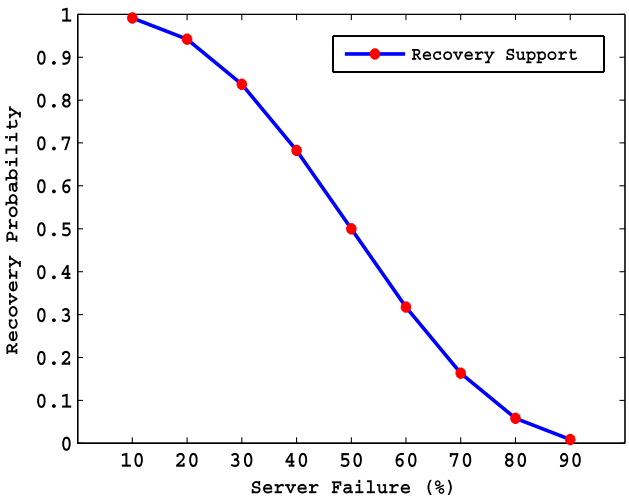


Fig. 8 Recovery of the unavailable fragments during server failures

storage servers. Therefore, the compute server first retrieves the extracted signatures from the CSS and performs the requested auditing operations on the local compute machine for the testing purpose. We create an app for each Dropbox account and generate the app keys and secrets for accessing permissions, which is a standard mechanism used to access the Dropbox account. Further, we also generate self-signed certificates via OpenSSL in order to authenticate all the entities.

7.2 Performance analysis and results

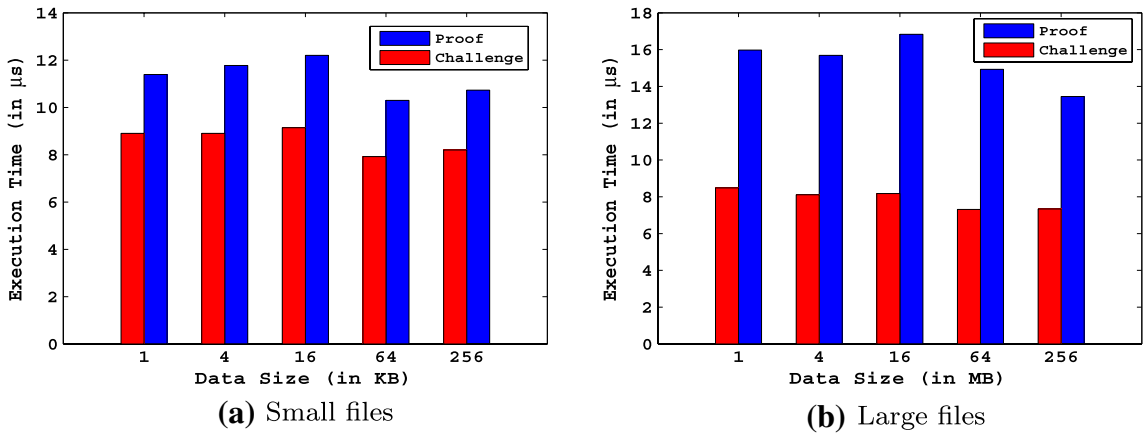
In this subsection, first, we discuss the availability and reliability results with various input variables. Then, we analyze the computation cost of various operations for proof verification. We also analyze the execution time for encoding, decoding, and signature fragment extraction operations. Later, we analyze the operation cost of signature fragments. Then we present the comparative analysis between the proposed technique and other state-of-the-art techniques of data verification.

We consider the testing datasets for conducting the experiments. These datasets contain multiple files that are created with random contents, and the size of files belongs to $\{2^{2*i}; 5 \leq i < 15\}$ bytes, giving file range of 2^0 to 2^{18} Kilobytes. We grouped these files into two sets, i.e., small and large, for analyzing the execution costs. In our experiments, the Python *timeit* module is used to evaluate execution time.

7.2.1 Reliability analysis

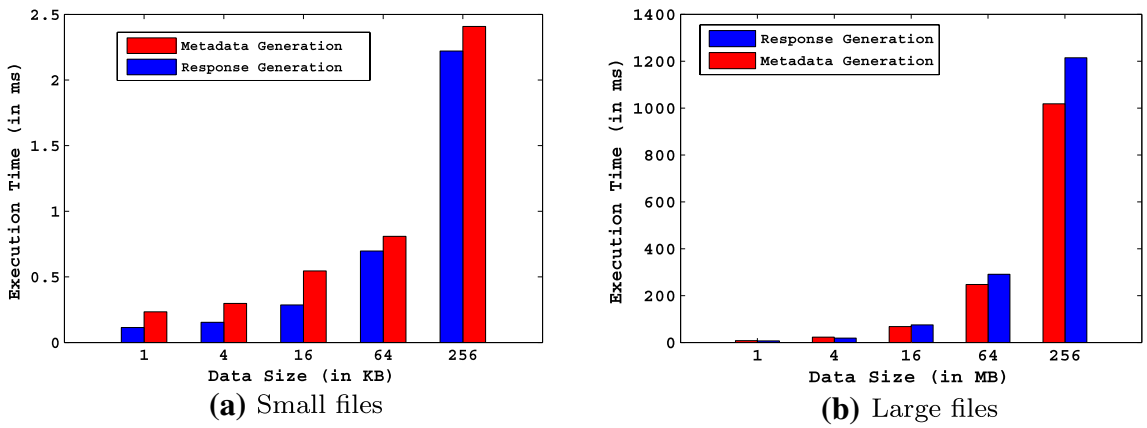
We analyze the data availability with *RS* encoding on various (α, β) pairs. Assume that the total existing servers be $N = 100000$ and let $M = 20\%$ are unavailable.

Figure 7 shows the achieved reliability in terms of data availability. We consider the number of data fragments ranging between 3 and 16 and the number of parity fragments ranging between 2 and 12. Then we use all the



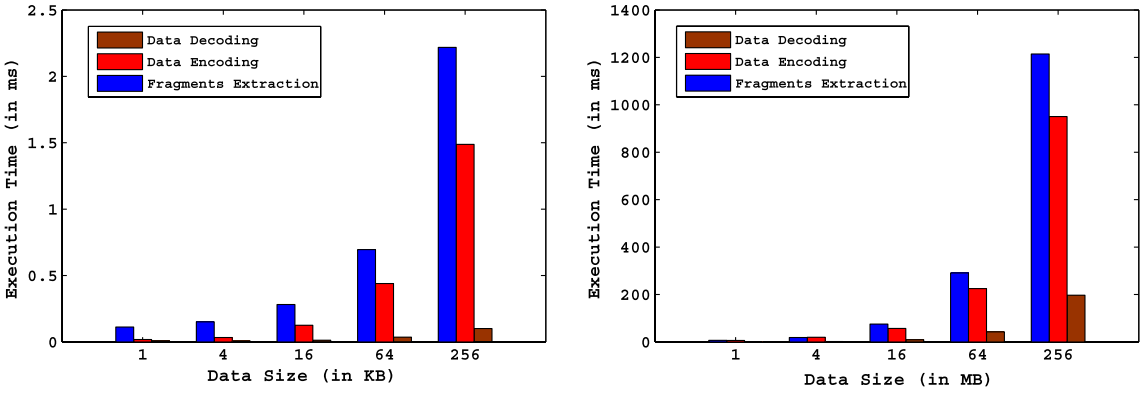
(a) Small files **(b)** Large files

Fig. 9 Execution time of challenge generation and integrity proof operations during verification



(a) Small files **(b)** Large files

Fig. 10 Execution time for metadata and response generation



(a) Small files **(b)** Large files

Fig. 11 Execution time of data encoding, decoding, and signature fragments extraction

combinations that can be generated from the data fragments $\alpha = \{3, 4, 5, 6, 8, 10, 12, 14, 16\}$ and the parity fragments $\beta = \{2, 3, 4, 6, 8, 10, 12\}$ to analyze the data availability. The minimum availability $P_a = 0.2713$ is achieved at $RS(16, 2)$ and maximum availability $P_a = 0.999999943$ at $RS(3, 12)$. As can be seen from the Fig. 7, it achieves almost 99.99% of data availability and high reliability at $\beta = 10$ and 12. Further, the reliability increases with the increase in the parity fragments. In particular, at $\alpha = 3$ and $\beta = 2$, the RS encoding provides the data availability with 0.94 probability and can tolerate the loss of two fragments. These encoding parameters take limited parity costs; thus, we use $RS(3, 2)$ for evaluating the execution time of various operations on data and signature fragments.

Further, we examined the impact of server failures during response generation operation, as presented in Fig. 8. As seen in the graph, it can guarantee the recovery of the evidence up to 40% of server failures.

7.2.2 Computation cost analysis of various operations for proof verification

We repeat each operation 1000 times to evaluate minimum, average, and maximum execution time. Then we consider the average time to present the result of these operations. First, we evaluate the computation cost of challenge generation and integrity proof operations in Fig. 9. The verifier executes these operations where verification operation takes constant time in microseconds ranging between 7−16 µs because it is independent of data size. Since the challenge generation operation requires extremely less time relative to Proof operations, we scale the execution cost of challenge generation by 10^5 in the graph. We then present the computation cost for metadata and response generation operations in Fig. 10, which depends on the input data size. As can be seen in Fig. 10a, both take constant time (< 0.9 ms) for data

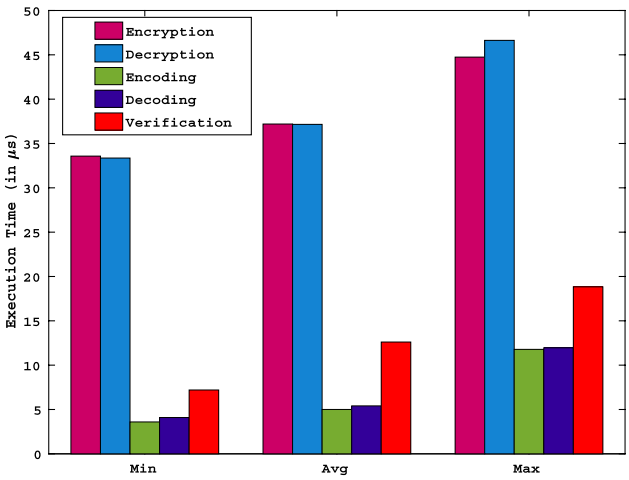


Fig. 12 Execution time of various operations on signature fragments

size up to 64 KB, and then the execution time exponentially increases with the increase in data size for larger files in Fig. 10b. However, these operations take a minimal amount of time in milliseconds (ms). Metadata generation operation works on the user-side while the response generation operation performs on the cloud server.

7.2.3 Encoding, decoding and signature extraction analysis

The compute server executes all the operations such as data encoding and decoding operations to store and retrieve the user data to/from the CSS's. We use *Reed–Solomon*, $RS(\alpha = 3, \beta = 2)$ scheme to evaluate the execution time of these encoding and decoding operations. Moreover, we also evaluate the extraction time of all the stored signature fragments from the CSS corresponding to the input dataset. We repeat these operations 1000 times to evaluate minimum, average, and maximum execution time. We consider the

Table 3 Operation complexity comparison between the proposed and existing verification schemes

Scheme	Client storage complexity	Communication complexity	Client computation complexity	Server computation complexity	Verifier computation cost
Schwarz and Miller (2006)	O(1)	O(n)	O(1)	O(n)	O(1)
Ateniese et al. (2007)	O(1)	O(1)	O(1)	O(1)	O(c)
Ateniese et al. (2008)	O(1)	O(1)	O(1)	O(1)	O(1)
Erway et al. (2009)	O(1)	O(log n)	O(log n)	O(clog n)	O(clog n)
Wang et al. (2010)	O(1)	O(clog n)	O(s)	O(clog n)	O(clog n)
Wang et al. (2011)	O(1)	O(log n)	O(log n)	O(log n)	O(1)
Chen (2013)	O(1)	O(log n)	O(1)	O(1)	O(1)
Chen et al. (2013)	O(1)	O(log n)	O(1)	O(1)	O(1)
Yuchuan et al. (2014)	O(1)	O(1)	O(n)	O(c)	O(c)
Yu et al. (2014)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)
Yu et al. (2015)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)
Zhang et al. (2016)	O(1)	O(clog n)	–	O(clog n)	O(clog n)
Wang et al. (2016)	O(1)	O(c + s)	–	O(c)	O(c + s)
Yu et al. (2016)	O(1)	O(c)	–	O(c)	O(c)
Garg and Bawa (2017)	O(1)	O(clog n)	–	O(clog n)	O(clog n)
Li et al. (2017)	O(1)	O(c + s)	–	O(c)	O(d.c + s)
Sookhak et al. (2017)	–	O(c)	O(c)	O(cn)	O(c)
Vasilopoulos et al. (2019)	O(s.α)	–	O(2n)	O(2n)	O(2n)
Ours	O(1)	O(1)	O(1)	O(ℓ)	O(1)

n is the file blocks, *c* is the sampled blocks, α is the data blocks, ℓ is the length of the fragment, *s* is the block segments, *d* is the authorized users, α is the data blocks, and "–" means not mentioned

average time to present the results of these operations. Figure 11 shows the execution time of these operations corresponding to the input dataset. As can be seen in Fig. 11, the execution time for data encoding, decoding, and signature fragments extraction time is very low (< 6.74 ms) for data size up to 1 MB. Then the time for encoding and signature fragment exponentially increases with the increase in data size. However, decoding time doesn't increase at the rate of encoding time due to more computation overhead in the encoding phase (Plank et al. 2009).

The signature extraction process runs simultaneously on each server due to the parallelism achieved because of the stored signature (single fragment) extraction process at one CSS is independent of the other.

7.2.4 Operation cost analysis of the signature fragments

We measure the execution cost of operations like encoding, decoding, encryption, decryption, and verifying the signature fragments. We repeat these operations 1000 times to evaluate minimum, average, and maximum execution time. We consider the average execution time to present the results of these operations. Figure 12 shows these various operations with the minimum, average, and maximum execution time in microseconds (µs). The verification operation is performed on the TPA, and all other operations are executed on the compute server. The average execution time of signature encoding and decoding operations are approximately 5 µs, and the average encryption and decryption execution time of signature fragment are approximately 37 µs. The verification process takes approximately 7−19 µs. These operations take a very minute amount of time which is in microseconds (µs).

7.2.5 Complexity analysis

This subsection analyses the complexity of various computation operations. In the proposed approach, $O(\ell)$ is the server's computation complexity. It is computed by extracting the signature fragment using a hash and a decryption operation. To produce the empirical evidence, these operations are simultaneously executed on the α number of CSS where alpha is a constant. One hash function takes the input that is the length of a fragment and produces an output of constant length with $O(\ell)$ complexity. Whereas, decryption operation completes in constant time.

The proposed scheme has constant client computation complexity and client storage complexity because the client does not need to store any metadata, and the client doesn't

perform any computation. It only initiates the verification request to the TPA. The proposed scheme has a constant verifier computation cost at TPA because TPA performs the signature matching operation which takes only constant time. Our scheme's communication complexity takes constant time to communicate proofs with a client because it sends only the small signature fragments for correctness verification.

7.2.6 Comparative analysis of various verification schemes

Table 3 compares the operation complexity of the proposed approach with the existing verification schemes based on various parameters such as client storage complexity, communication complexity, client computation complexity, server computation complexity, and verifier computation cost. From the table, we can see that the Vasilopoulos et al. (2019) has maximum client, server, and verifier computation cost and maximum client storage complexity while Schwarz and Miller (2006) has maximum communication cost during the data verification. However, the proposed scheme takes constant time, i.e., $O(1)$ to perform all these operations except server computation costs during the verification. Besides, our scheme considers proof reliability over distributed fragments. Hence, we can conclude that the proposed approach is relatively better than other state-of-the-art approaches.

8 Conclusion

To conclude, this research introduces a novel technique for the reliable verification of encoded fragments stored across the cloud datacenters. $RS(\alpha, \beta)$, a type of eraser coding, is innovatively used in this research work for creating data and signature fragments. The proposed technique achieves high availability and reliability of signature by storing its fragments in a distributed fashion similar to the data. We perform extensive experiments in real-time and discuss the performance analysis of various operations. Experimental results vividly demonstrate the availability and reliability matrices as per the various combinations of data and parity fragments. Results show that $RS(3, 2)$ provides 0.94 probability of data availability with least parity cost and significantly high reliability up to 99.99% with parity fragments 10 and 12. Hence, high availability and reliability of signature, along with the data, leads to reliable verification of the distributed fragments of data, even in the case of unavailability of a set of servers in the cloud. In the future, we will explore our work by adopting the blockchain technology to establish direct trust between the users. So, the user does not need to depend on the third-party to validate stored data integrity.

References

Ateniese G, Burns R, Curtmola R, Herring J, Kissner L, Peterson Z, Song D (2007) Provable data possession at untrusted stores. In: Proceedings of the 14th ACM conference on computer and communications security. ACM, New York, NY, USA, pp 598–609

Ateniese G, Di Pietro R, Mancini LV, Tsudik G (2008) Scalable and efficient provable data possession. In: Proceedings of the 4th international conference on security and privacy in communication networks. ACM, New York, NY, USA, pp 1–10 (Article number 9)

Behl A (2011) Emerging security challenges in cloud computing: an insight to cloud security challenges and their mitigation. In: 2011 World Congress on information and communication technologies. Mumbai, MH, India, pp 217–222

Bowers KD, Juels A, Oprea A (2009a) HAIL: a high-availability and integrity layer for cloud storage. In: Proceedings of the 16th ACM conference on computer and communications security. ACM, New York, NY, USA, pp 187–198

Bowers KD, Juels A, Oprea A (2009b) Proofs of retrievability: theory and implementation. In: Proceedings of the 2009 ACM workshop on cloud computing security. ACM, New York, NY, USA, pp 43–54

Chen L (2013) Using algebraic signatures to check data possession in cloud storage. Future Gener Comput Syst 29(7):1709–1715

Chen L, Zhou S, Huang X, Xu L (2013) Data dynamics for remote data possession checking in cloud storage. Comput Electr Eng 39(7):2413–2424

Chouhan V, Peddoju SK (2020) Investigation of optimal data encoding parameters based on user preference for cloud storage. IEEE Access 8:75105–75118

Erway C, Küpçü A, Papamanthou C, Tamassia R (2009) Dynamic provable data possession. In: Proceedings of the 16th ACM conference on computer and communications security. ACM, New York, NY, USA, pp 213–222

Garg N, Bawa S (2017) RITS-MHT: relative indexed and time stamped Merkle hash tree based data auditing protocol for cloud computing. J Netw Comput Appl 84:1–13

Ghemawat S, Gobioff H, Leung ST (2003) The google file system. In: Proceedings of the nineteenth ACM symposium on operating systems principles. ACM, New York, NY, USA, pp 29–43

Gřivna T, Drápal J (2019) Attacks on the confidentiality, integrity and availability of data and computer systems in the criminal case law of the Czech Republic. Digit Investig 28:1–13

Gudeme JR, Pasupuleti SK, Kandukuri R (2020) Attribute-based public integrity auditing for shared data with efficient user revocation in cloud storage. J Ambient Intell Humaniz Comput 1–14

Hou H, Yu J, Hao R (2019) Cloud storage auditing with deduplication supporting different security levels according to data popularity. J Netw Comput Appl 134:26–39

Huang C, Simitci H, Xu Y, Ogus A, Calder B, Gopalan P, Li J, Yekhanin S (2012) Erasure coding in windows azure storage. In: Proceedings of the 2012 USENIX conference on annual technical conference. USENIX Association, pp 15–26

Jayaraman I, Panneerselvam AS (2020) A novel privacy preserving digital forensic readiness provable data possession technique for health care data in cloud. J Ambient Intell Humaniz Comput 1–14

Juels A, Kaliski BS Jr (2007) Pors: proofs of retrievability for large files. In: Proceedings of the 14th ACM conference on computer and communications security. ACM, New York, NY, USA, pp 584–597

Khan MA (2016) A survey of security issues for cloud computing. J Netw Comput Appl 71:11–29

Li P, Jin X, Stones RJ, Wang G, Li Z, Liu X, Ren M (2016) Parallelizing degraded read for erasure coded cloud storage systems using

collective communications. In: 2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, pp 1272–1279
Li Y, Yu Y, Min G, Susilo W, Ni J, Choo KKR (2017) Fuzzy identity-based data integrity auditing for reliable cloud storage systems. IEEE Trans Dependable Secur Comput 16(1):72–83
Mell P, Grance T et al (2011) The NIST definition of cloud computing. NIST Spec Publ 800(145):1–4
Plank JS, Luo J, Schuman CD, Xu L, Wilcox-O'Hearn Z et al (2009) A performance evaluation and examination of open-source erasure coding libraries for storage. Fast 9:253–265
Premkamal PK, Pasupuleti SK, Alphonse P (2019) A new verifiable outsourced ciphertext-policy attribute based encryption for big data privacy and access control in cloud. J Ambient Intell Humaniz Comput 10(7):2693–2707
Punitha AAA, Indumathi G (2020) A novel centralized cloud information accountability integrity with ensemble neural network based attack detection approach for cloud data. J Ambient Intell Humaniz Comput 1–12
Rashmi K, Shah NB, Gu D, Kuang H, Borthakur D, Ramchandran K (2014) A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. SIGCOMM Comput Commun Rev 44(4):331–342
Reed I, Solomon G (1960) Polynomial codes over certain finite fields. J Soc Ind Appl Math 8(2):300–304
Sangaiah AK, Medhane DV, Bian GB, Ghoneim A, Alrashoud M, Hossain MS (2019a) Energy-aware green adversary model for cyber-physical security in industrial system. IEEE Trans Ind Inform 16(5):3322–3329
Sangaiah AK, Sadeghilalimi M, Hosseinabadi AAR, Zhang W (2019b) Energy consumption in point-coverage wireless sensor networks via bat algorithm. IEEE Access 7:180258–180269
Schmuck FB, Haskin RL (2002) GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the conference on file and storage technologies. USENIX Association, Berkeley, CA, USA, pp 231–244
Schwarz TSJ, Miller EL (2006) Store, forget, and check: using algebraic signatures to check remotely administered storage. In: 26th IEEE international conference on distributed computing systems (ICDCS). Lisboa, Portugal, p 12
Singh A, Chatterjee K (2017) Cloud security issues and challenges: a survey. J Netw Comput Appl 79:88–115
Sookhak M, Yu FR, Zomaya AY (2017) Auditing big data storage in cloud computing using divide and conquer tables. IEEE Trans Parallel Distrib Syst 29(5):999–1012
Tabrizchi H, Rafsanjani MK (2020) A survey on security challenges in cloud computing: issues, threats, and solutions. J Supercomput 76:9493–9532
Thusoo A, Shao Z, Anthony S, Borthakur D, Jain N, Sen Sarma J, Murthy R, Liu H (2010) Data warehousing and analytics infrastructure at facebook. In: Proceedings of the 2010 ACM SIGMOD international conference on management of data. ACM, New York, NY, USA, pp 1013–1020
Vasilopoulos D, Elkhiyaoui K, Molva R, Onen M (2018) POROS: proof of data reliability for outsourced storage. In: Proceedings of the 6th international workshop on security in cloud computing. Association for Computing Machinery, New York, NY, USA, pp 27–37
Vasilopoulos D, Önen M, Molva R (2019) PORTOS: proof of data reliability for real-world distributed outsourced storage. In: Obaidat MS, Samarati P (eds) Proceedings of the 16th international joint conference on e-business and telecommunications, ICETE 2019—volume 2: SECRYPT, Prague, Czech Republic, July 26–28, 2019. SciTePress, Setúbal, pp 173–186
Wang H (2015) Identity-based distributed provable data possession in multicloud storage. IEEE Trans Serv Comput 8(2):328–340
Wang C, Wang Q, Ren K, Lou W (2010) Privacy-preserving public auditing for data storage security in cloud computing. In: 2010 Proceedings IEEE INFOCOM, San Diego, CA, USA, pp 1–9
Wang Q, Wang C, Ren K, Lou W, Li J (2011) Enabling public auditability and data dynamics for storage security in cloud computing. IEEE Trans Parallel Distrib Syst 22(5):847–859
Wang C, Wang Q, Ren K, Cao N, Lou W (2012) Toward secure and dependable storage services in cloud computing. IEEE Trans Serv Comput 5(2):220–232
Wang C, Chow SSM, Wang Q, Ren K, Lou W (2013) Privacy-preserving public auditing for secure cloud storage. IEEE Trans Comput 62(2):362–375
Wang B, Li B, Li H (2015) Panda: public auditing for shared data with efficient user revocation in the cloud. IEEE Trans Serv Comput 8(1):92–106
Wang Y, Wu Q, Qin B, Shi W, Deng RH, Hu J (2016) Identity-based data outsourcing with comprehensive auditing in clouds. IEEE Trans Inf Forensics Secur 12(4):940–952
Wang H, Wang XA, Xiao S, Liu J (2020) Decentralized data outsourcing auditing protocol based on blockchain. J Ambient Intell Humaniz Comput 1–12
Weatherspoon H, Kubiatowicz J (2002) Erasure coding vs. replication: a quantitative comparison. In: Revised papers from the first international workshop on peer-to-peer systems. Springer, London, UK, pp 328–338
Xia M, Saxena M, Blaum M, Pease DA (2015) A tale of two erasure codes in HDFS. In: Proceedings of the 13th USENIX conference on file and storage technologies. USENIX Association, Berkeley, CA, USA, pp 213–226
Yu Y, Ni J, Au MH, Liu H, Wang H, Xu C (2014) Improved security of a dynamic remote data possession checking protocol for cloud storage. Expert Syst Appl 41(17):7789–7796
Yu Y, Zhang Y, Ni J, Au MH, Chen L, Liu H (2015) Remote data possession checking with enhanced security for cloud storage. Future Gener Comput Syst 52(C):77–85
Yu Y, Au MH, Ateniese G, Huang X, Susilo W, Dai Y, Min G (2016) Identity-based remote data integrity checking with perfect data privacy preserving for cloud storage. IEEE Trans Inf Forensics Secur 12(4):767–778
Yuchuan L, Shaojing F, Ming X, Dongsheng W (2014) Enable data dynamics for algebraic signatures based remote data possession checking in the cloud storage. China Commun 11(11):114–124
Zhang Y, Xu C, Liang X, Li H, Mu Y, Zhang X (2016) Efficient public verification of data integrity for cloud storage systems from indistinguishability obfuscation. IEEE Trans Inf Forensics Secur 12(3):676–688