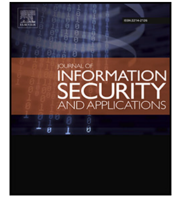
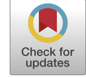
Journal of Information Security and Applications 69 (2022) 103265

Contents lists available at ScienceDirect

Journal of Information Security and Applications

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





dualDup: A secure and reliable cloud storage framework to deduplicate the encrypted data and key

Vikas Chouhan [a,*], Sateesh K. Peddoju [a], Rajkumar Buyya [b]

[a] Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, India
[b] School of Computing and Information Systems, University of Melbourne, Melbourne, VIC 3010, Australia

ARTICLE INFO

Keywords:
Cloud security
Cloud storage
Convergent Encryption
Deduplication
DupLESS
Reliability
Erasure Coding

ABSTRACT

Cloud Storage Providers generally maintain a single copy of the *identical* data received from multiple sources to optimize the space. They cannot deduplicate the identical data when the clients upload the data in the encrypted form. To address this problem, recently, Duplicateless Encryption for Simple Storage (DupLESS) scheme is introduced in the literature. Besides, the data stored in the cloud is unreliable due to the possibility of data losses in remote storage environments. The DupLESS scheme, on the other hand, keeps both the key and the data on a single storage server, which is unreliable if that server goes down. In essence, the existing related works aim to handle either secure-deduplication or reliability limited to either key reliability or the data reliability. Hence, there is a need to develop a secure-deduplication mechanism that is not vulnerable to any malicious activity, semantically secures both data and key, and achieves the reliability. To address these problems, this paper proposes the *dualDup* framework that (*a*) optimizes the storage by eliminating the duplicate encrypted data from multiple users by extending DupLESS concept, and (*b*) securely distributes the data and key fragments to achieve the privacy and reliability using Erasure Coding scheme. The proposed approach is implemented in Python on the top of the Dropbox datacenter and corresponding results are reported. Experiments are conducted in a realistic environment. The results demonstrate that the proposed framework achieves reliability with an average storage overhead of 66.66% corresponding to the Reed–Solomon(3,2) codes. We validated through security analysis that the proposed framework is secure from insider and outsider adversaries. Moreover, *dualDup* framework provides all the aspects of deduplication, attack mitigation, key security and management, reliability, and QoS features as compared to other state-of-the-art deduplication techniques.

1. Introduction

Cloud computing [1] offers ubiquitous, elastic, and utility computing services that can be rapidly provisioned and made accessible to users via the Internet. Cloud allows the users to outsource the storage services to store tremendous amounts of data, which can be retrieved as and when required [2]. Google Drive,[1] Microsoft OneDrive,[2] Apple iCloud Drive,[3] SugarSync,[4] OpenDrive,[5] IDrive,[6] and Dropbox[7] are few examples of cloud storage offerings. In general, Cloud Service Providers (CSPs) store a single copy of the *identical* data received from multiple sources to optimize the space. However, CSPs cannot distinguish identical data when the clients upload the data in an encrypted form. This problem is a secure-deduplication problem. Convergent Encryption (CE) [3], Message-Locked Encryption (MLE) [4,5], and Duplicateless Encryption for Simple Storage (DupLESS) [6] methods solve the dissimilar encryption data problem by applying identical key encryption concept. However, all three schemes have some limitations. Both CE and MLE are vulnerable to brute force and statistical attacks. The DupLESS scheme stores the key and data on the single storage server and hence, the users cannot access their data if that server is down.

* Corresponding author.
E-mail addresses: vchouhan@cs.iitr.ac.in (V. Chouhan), sateesh@ieee.org (S.K. Peddoju), rbuyya@unimelb.edu.au (R. Buyya).

[1] http://drive.google.com.
[2] https://onedrive.live.com/.
[3] https://www.apple.com/in/icloud/icloud-drive/.
[4] https://www.sugarsync.com/.
[5] https://www.opendrive.com/.
[6] https://www.idrive.com/.
[7] http://www.dropbox.com/.

https://doi.org/10.1016/j.jisa.2022.103265

Available online 19 July 2022
2214-2126/© 2022 Elsevier Ltd. All rights reserved.

Moreover, if an attacker gets access to the storage server, there is a high possibility of the key and data security being compromised. Achieving high reliability is another important concern in the cloud environment. It provides unaffected services to the users even if some servers fail or become inaccessible. It also ensures the recoverability of corrupted/lost data in some cases. This problem is a reliability problem. Therefore, we aim to design a system that can provide both secure-deduplication and reliability as part of cloud storage framework.

1.1. Motivation

Several related studies that attempt to handle secure-deduplication have been proposed in [4,6–9]. However, they have not considered how to achieve reliability. Similarly, some researches [10–16] support the reliability of cloud storage but failed to address deduplication. To the best of our knowledge, none of the previous works, except the work proposed in [17], have achieved secure-deduplication and data reliability at the same time; however, they did not support the key reliability. The authors in [18,19] have applied CE and Ramp Secret Sharing Scheme (RSSS) [20] to achieve key reliability, in addition to deduplication, but they did not support the data reliability. Since all the operations are executed on the client-side in both [18,19] the client overhead is increased. Therefore, providing reliability, reducing client-side overhead, and protecting confidentiality while achieving secure-deduplication in the cloud environment is still challenging. Hence, there is a need to develop a secure-deduplication mechanism that is not vulnerable to any malicious activity, reduces client-side overhead, semantically secures both data and key, and achieves reliability.

1.2. Contributions

In this paper, we propose a novel ***dualDup*** framework that primarily provides secure-deduplication and reliability for both data and key. We innovatively extending the DupLESS concept [6] and Erasure Coding (EC) [10,11] scheme to achieve secure-deduplication and reliability, respectively. The novelty of the proposed approach lies in the fact that the DupLESS concept executes at both client and server. In the *dualDup* framework, the DupLESS [6] concept is applied to the client to achieve confidentiality and secure-deduplication. Similarly, at the server, to protects the data from system admin and other malicious insiders or users. In addition, the proposed framework provides all the aspects of deduplication, attack mitigation, key security and management, reliability, and QoS features as compared to other state-of-the-art deduplication techniques. The main contributions of the proposed approach are highlighted below:

- We propose the novel algorithms for file upload, download, and delete operations. Further, we demonstrate the scenario where multiple clients were uploading the same file content to datacenters.
- The proposed framework employs an EC mechanism to recover the original data even if some fragments are lost, hence achieving reliability. For storing a file, the proposed approach splits the data and key into fragments using the EC technique and securely distributes them to the distinct Data Storage Servers (DSS) and Key Storage Servers (KSS), respectively.
- We handle the inside-user and cross-user deduplication at the Trusted Third Party (TTP) and Cloud Service Provider (CSP) levels, respectively. In our framework, we used TTP to eliminate client-side storage overhead. The TTP saves the network bandwidth if the data already exists in the cloud storage.
- We validated through security analysis that the proposed framework is secure from insider and outsider adversaries. In particular, the *dualDup* remains secure even if an adversary compromises a certain number of storage servers. The proposed approach ensures that a brute force attack cannot reveal sensitive information. In addition, we perform extensive experiments and discuss the performance analysis of various operations.

1.3. Organization

The rest of the paper is organized as follows. We define the preliminaries, notations and important terms used by *dualDup* framework in Section 2. We discuss the concepts of secure-deduplication, and erasure coding and review the related work in Section 3. Threat Model and Design goals are presented in Section 4. We explain the proposed *dualDup* scheme in detail along with its system components in Section 5. The evaluations of the proposed *dualDup* framework in terms of security and performance analysis are presented in Section 6. Finally, Section 7 concludes the paper with future directions.

2. Preliminaries

We present the list of notations used throughout the paper in Table 1, and then describe the primitives used in our proposed framework.

2.1. Fundamental operations

A set of function definitions that are used in symmetric encryption, security, file, and datacenter operations are discussed below:

- $k \longleftarrow KeyGen(1^\lambda)$: This probabilistic algorithm takes as an input a security parameter ($1^\lambda | \lambda \in \mathbb{N}$, i.e., sequence of 1's), and generates the secret key k.
- $C \longleftarrow E(k, x)$: The symmetric encryption algorithm that takes input as the secret key k and file content x, and generates the corresponding Ciphertext C.
- $x \longleftarrow D(k, C)$: The symmetric decryption algorithm that takes input as the secret key k and Ciphertext C, and outputs the file content x.
- $l \longleftarrow TagGen(x)$: This algorithm maps the input content x, and outputs a fixed length tag value that is also known as file locator l.

2.1.1. File and datacenter operations

These operations help in interacting with datacenters, key server, and file operations.

- $k_{dup} \longleftarrow g_d(x)$: This function takes as input the file content x, computing the content hash H_x, and then sends the content hash H_x to the Key Server (KS), which in return generates the dupless key k_{dup}. Both client and compute server with their input file contents to obtain their corresponding dupless key.
- $\delta_d \longleftarrow f_{ds}(x)$: This function runs at compute server that takes x as input data and then returns the fragments of x. Fragments are created by calling an encoding module of Erasure Code (EC) [11].
- $\delta_k, k_{dup} \longleftarrow f_{ks}(C)$: This function runs at compute server by taking ciphertext as input. This function first calls the function $g_d(C)$ to obtain the dupless key k_{dup} from KS, and then the function computes the key fragments δ_k of dupless key using f_{ds} function.
- $A \longleftarrow f_a(arg)$: This function runs on compute server to determine the available storage servers, and returns the set of available data storage servers with input argument value $\varepsilon dss\varepsilon$ and key storage servers with input argument value $\varepsilon kss\varepsilon$.
- $C_d, C_k, l_d, l_k \longleftarrow Up_f(\eta)$: This function initializes the ciphertext and locator of both data and key at the client-side during file uploading procedure by computing the data ciphertext (C_d), key ciphertext (C_k), data locator (l_d), and key locator (l_k) corresponding to the client's input file η.
- $\eta_{List} \longleftarrow G_{fl}(U_{id})$: This function runs on the client side that calls Trusted Third Party (TTP) service using a client ID. The TTP will return the corresponding file list to the client. File list is symbolized as $\eta_{List} = \sum_{i=1}^{n} \left(\eta_i, l_{d_i}, l_{k_i}\right)$ corresponding to the U_{id}, where n is the number of files stored in the datacenters.

Table 1
Notations.

Symbol	Description	Symbol	Description
U_{id}	User/Client ID	n_0	File Counter
T_{id}	TTP ID	η	Filename
S_{id}	Compute Server ID	η_c	Content Filename
U	User/Client	η_k	Key Filename
K/k	Key	x	File Content
s	Secret Key of User	H_x	Hash value of content x
P_k	Private Key	η^x	File η with content x
C	Ciphertext	k_{dup}	Dupless Key
C_d	Data Ciphertext	Q_R	Request Queue
C_k	Key Ciphertext	δ	Set of n Fragments where $\delta = \{\delta_1, \delta_2, \dots, \delta_n\}$
l	Locator/Tag	δ_d	Set of Data Fragments
l_d	Data Locator	δ_k	Set of Key Fragments
l_k	Key Locator	α	Number of Data Fragments
A	Set of Storage Server	β	Number of Parity Fragments
A_i	i^{th} Storage Server	γ	Total Fragments, $\alpha + \beta$
$DB_i(\cdot)$	Insert the input values into the database	$DB_u(\cdot)$	Update entry in database
$DB_d(\cdot)$	Delete entry from database where the input values exist	$DB_c(\cdot)$	Count all entry from the database where the input values exist in a row

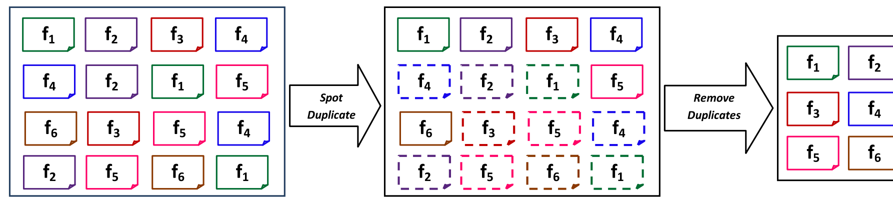


Fig. 1. A Typical Deduplication Procedure.

- $x \longleftarrow R_f(\eta)$: This function reads the file η and returns the entire file contents x stored in the file η.
- $W_f(\eta, x)$: This function creates a file with filename η and write the contents x inside the file η.
- $Put^*_f(\eta, x)$: This function puts/stores the content x with file name η to the corresponding storage server which is symbolized by *.
- $Del^*_f(\eta)$: This function deletes the file η from storage server which is symbolized by *.
- $C_{\delta_i} \longleftarrow Get_f(\eta)$: This function gets/retrieves the encrypted fragment of a file η from the storage server and then writes the contents of this retrieved fragment to the variable C_{δ_i} (i^{th} fragment of stored ciphertext).
- $decode(\delta)$: This function decodes the set of fragments (δ) by calling decoding module of EC that combines all the fragments to construct a single meaningful data.

3. Background and related work

We discuss the Erasure Coding (EC) and Convergent Encryption (CE) mechanisms that are required to achieve reliability and secure-deduplication, respectively. Further, we discuss the existing solutions that aim to achieve deduplication and reliability.

3.1. Background

3.1.1. Secure-deduplication

Deduplication [21] refers to the procedure in which CSP stores a single copy of the identical data from multiple clients to eliminate the duplicacy in storage. Hence, the deduplication saves the storage and bandwidth, as the data is not uploaded again if it already exists [22,23]. Fig. 1 shows a typical scenario where a datacenter has the multiple files $(f_i : i \in \mathbb{N}^+)$, and the datacenter identifies and spots the multiple copies of a file to remove the duplicates.

Users tend to store their data in an encrypted form to the Cloud Storage Servers (CSS) using their secret key. Therefore, when a user uploads the data in an encrypted form, CSP cannot distinguish the identical data stored in the cloud. Hence, it leads to multiple dissimilar ciphertext copies of identical data stored by multiple users which results in redundancy and huge wastage of space in the CSS [3–5,7,24]. Fig. 2 shows a scenario in which multiple users upload the encrypted copies of the same data to the datacenter. Assume that a set of n users encrypt the file η^x having same contents x with their secret key and uploads it to the datacenter. At server, encrypted files $C_i = E(k_i, \eta^x)$, for $i = 1, \dots, n$ are received from n users. Thus, cloud datacenter receives distinct encrypted files $\{C_1, C_2, C_3, \dots, C_n\}$ for the same file η^x and it leads to the failure of deduplication. The secure-deduplication removes all such redundant encrypted files to save the storage space. *Convergent Encryption* and *Duplicateless Encryption for Simple Storage (DupLESS)* are two basic schemes that provide secure-deduplication.

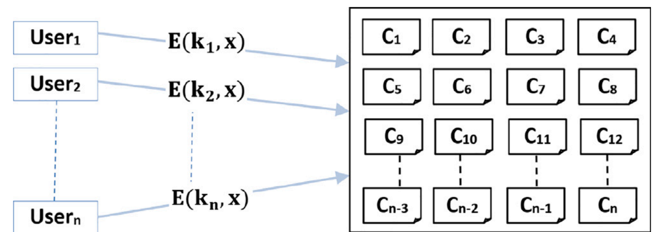


Fig. 2. Encrypted Copies of Identical Data Stored by Multiple Users.

(1) *Convergent Encryption*
Convergent Encryption (CE) scheme [3–5] solves secure-deduplication problem with confidentiality by applying identical key encryption. It reduces the redundancy of encrypted data by generating a unique key for similar file contents using hash algorithms.
Fig. 3 shows the convergent encryption and decryption process of a message $m = \{0,1\}^*$. In the encryption process, following two steps are executed:

(a) The hash function H generates a unique key $k = \{0,1\}^{l_k}$ of length l_k, for the message m. Encryption function E is applied to encrypt m using key k to generate C_1.

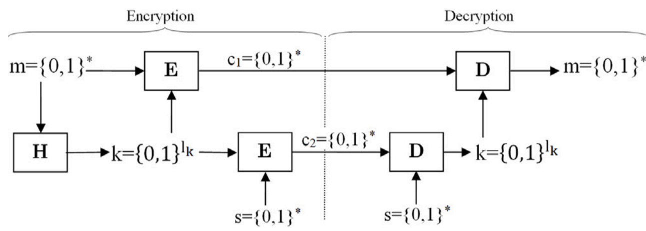


Fig. 3. Convergent Encryption and Decryption Process.

(b) Further, k is encrypted with E that takes secret key s of the user to generate the ciphertext C_2 of k.

Similarly, the decryption process consists of the following two steps:

(a) User decrypts C_2 with decryption function D that uses the secret key s to generate back the key k.
(b) Further, D decrypts C_1 with the key k to generate the original message m.

CSPs like Dropbox, Google, Bitcasa, and Amazon have adopted the CE concept [4,18]. However, CE is not semantically secure [4] and also vulnerable to brute force and statistical type of attacks due to the deterministic property of content hashing [25, 26].

(2) *Duplicateless Encryption for Simple Storage*
Duplicateless Encryption for Simple Storage (DupLESS) [6] concept is used to enable secure data deduplication, which uses rate-limiting approach to provide the resistance against brute-force attack. Rate limiting approach ensures that a server handles a limited number of requests for a particular client during a fixed interval of time. This technique uses a key server to generate an identical key k_x for all clients with similar data content x, by using its hash value H_x. DupLESS key server uses a key generation protocol that uses an oblivious pseudorandom function (OPRF) [27] to generate key k_x. The $OPRF$ ensures that the key generation process does not reveal any file information and learn the KS secret key. Fig. 4 shows a scenario where multiple users want to upload the same data x. Initially, with the help of DupLESS key server, they generate the identical keys k_x, and then encrypt x with k_x to generate the identical encrypted files C_x. CSS receives multiple identical copies, C_x, and stores only a single copy, thus achieving deduplication.

3.1.2. Reliability

Major CSPs claim reliability as one of the important aspects of their services. Reliability ensures that the services offered to the users are minimally affected even if a certain number of servers failures or data loss. Reliability is achieved by distributing the fragments across the multiple storage servers to enhance data recoverability and availability. Erasure Coding (EC) is a widely used technique for creating reliable data fragments.

Erasure Coding: EC [10,11] technique splits the actual data into the number of coded blocks as per the predefined threshold. This threshold, say T, has two components, i.e., α and β. The first component, α, denotes a total number of data fragments and β is the total number of parity fragments. The sum of these two components represent the total number of coded blocks generated by EC, say γ. EC can reconstruct the actual data from any α number of coded blocks and it can tolerate the loss up to β number of coded blocks. This reconstruction provides reliability. The encoding rate of this technique, i.e., $\frac{\alpha}{\gamma}$ must be less than one to guarantee reliability. It is obvious that if the encoding rate is equal to one, then the technique cannot tolerate any number of lost coded blocks, which leads towards the loss of reliability. In this technique, the storage cost is increased by a factor of $\frac{\gamma}{\alpha}$ [28]. However, it is better than other techniques providing reliability, such as N-way replication technique [13], in terms of storage cost and performance [10].

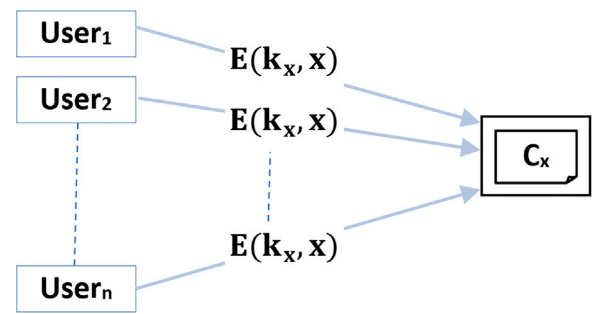


Fig. 4. DupLESS Scheme with Deduplication.

3.2. Related work

In this subsection, we review related works that focus on secure-deduplication and reliability. We classify them into two categories: (i) Reliability Solutions, and (ii) Secure-deduplication Solutions.

3.2.1. Reliability solutions

There exist several related works [10–16] that focus to achieve the reliability in cloud storage. Zhang et al. [12] proposed an efficient Cauchy coding approach for data storage that generates series of schedules to select the optimal schedule using heuristic approach. The authors used Hadoop File System (HDFS) for their implementation. Li et al. [10] implemented Collective Reconstruction Read for improving the read operation performance. They reduced the read latency of EC. They applied *read* and *computation* operations in parallel and improved the system availability. Xu et al. [15] proposed a decentralized encoding framework for EC in cloud storage. They achieved better read/write performance and low network traffic using an incremental encoding. The authors in both [10,15] used HDFS-RAID system for their implementation.

To provide secure data forwarding and system recovery, Lin et al. [11] proposed a secure distributed storage system which applies the proxy re-encryption scheme. To achieve efficient I/O performance and significant stability, Yin et al. [13] introduced ASSembling chain of Erasure coding and Replication(ASSER) storage scheme that combined N-way replication and EC to store each object entirely as well as in segments. They proposed multiversional parity logging mechanism for handling efficient read/write operations.

All of the schemes discussed above have achieved reliability but failed to address the deduplication problem. On the contrary, the proposed framework aims to provides both deduplication and reliability.

3.2.2. Secure-deduplication solutions

There exist several related works [4,6–9] to achieve secure-deduplication. All these works have used the CE concept to avoid redundancies in storage. To address the data security and space efficiency problem, Storer et al. [7] developed two models: authenticated and anonymous model. To achieve deduplication, the client encrypted the file chunks using CE before transferring to the storage. These models hide the users' identities and permit only authorized users. However, their model can lead to information leakage due to lack of key security.

To provide a secure and efficient storage service, Puzio et al. [8] introduced ClouDedup, which assured block-level deduplication and data confidentiality simultaneously. They applied an additional encryption layer to provide privacy and confidentiality from malicious

service providers. Bellare et al. [4] analyzed the security of MLE scheme family and justified with proofs in the random-oracle-model practically. Further, theoretically, they addressed the issue of finding a standard model MLE scheme using Extract-Hash-Check and Sample-Extract-Encrypt procedure. Few other authors [29–31] support data deduplication over encrypted data in their works; however, they are not explored towards providing data reliability to deal with data corruption or data unavailability.

Kaaniche et al. [9] proposed client-side deduplication solution for data outsourcing. They ensured the data confidentiality from an unauthorized user by using CE and metadata (containing access rights). The authors used OpenStack Swift for their implementation.

All of these schemes achieved secure-deduplication but failed to discuss the reliability. On the contrary, the proposed framework aims to provide both deduplication and reliability.

We further review the secure-deduplication solutions in three categories: (i) Secure-deduplication with Key Reliability, (ii) Secure-deduplication with Data Reliability, and (iii) Secure-deduplication with Client Side Overhead.

- *Secure-deduplication with Key Reliability*
  A huge number of keys are created during the operation of secure-deduplication that introduces new challenges regarding the key management. The works in [18,19] store keys to the storage server to provide key reliability. Li et al. [18] and Zhou et al. [19] proposed the Dekey approach and Multi-Level Key (MLK) management approach, respectively, to eliminate the difficulty of key management by using Ramp Secret Sharing Scheme (RSSS) [20]. These works achieved key level reliability but failed to provide data reliability. However, the proposed work provides both key and data reliability.
- *Secure-deduplication with Data Reliability*
  Li et al. [17] proposed a distributed deduplication system that distributed the data chunks across the storage servers to achieve data reliability. It used RSSS, message authentication code and Tag generation algorithm to ensure confidentiality, integrity and tag consistency. Douceur et al. [3] introduced SelfArranging, Lossy, Associative Database (SALAD) to maintain the file records in a decentralized manner, and stored redundant copies to enable reliability. These approaches achieved data reliability but failed to discuss key reliability. However, the proposed work provides both key and data reliability.
- *Secure-deduplication with Client Side Overhead*
  Several related works like [6,17–19] provided both inside-user and cross-user deduplication. SecDep [19] is a cross-user file-level and inside-user chunk-level deduplication method with User Aware Convergent Encryption. It is a variant of CE scheme to defend against brute force attacks. The server-aided method creates the file-level CE keys to ensure the security of cross-user deduplication. The user-aided method is used to create chunk-level keys with lower computational overheads. Since all operations are performed on the client side, it increases the client overhead. However, the proposed approach in this work eliminates the client space overhead.

To the best of our knowledge, none of the existing works provide secure-deduplication with both key and data reliability. The proposed framework in this paper aims to achieve reliability (for both key and data) along with deduplication by distributing the keys and data fragments in a secured manner.

4. Threat model and design goals

Generally, the Cloud Storage Servers (CSS), Trusted Third Party (TTP) and Key Server (KS) claims that they are not involved in any malicious activity, such as disclosing the information. However, the user may not be able to recognize such activity in case they carry out some malicious actions. There are two types of adversaries in a system, i.e., insider adversary and outsider adversary. An insider adversary may be referred to as a CSS or CSS agent who can outsource some sensitive information stored in the cloud to the untrusted environment. Outsider adversary may be a user or any malicious entity who is not aware of the internal architecture of the cloud and still tries to compromise the system to retrieve the sensitive information stored on the system. Hence, the system should be designed in such a way that it is secure from both insider and outsider adversaries. Thus, we propose a system *dualDup*, which achieves the following design goals:

1. *Compromise Resilience:* It refers to the situation where an adversary may control a certain number of components of the system, but still is unable to decrypt the stored encrypted information. In the proposed architecture, both dupless key and file contents are encrypted and distributed among the CSS to semantically secure the dupless key.
2. *Brute-force Attack Resilience:* It refers to the scenario where an enemy attempts to decrypt the ciphertext by continuously sending the requests to the KS to retrieve the key. The proposed framework uses DupLESS scheme, that limits the number of requests made by the user in the fixed interval of time, to set the key from KS, and resist the brute force attack.
3. *Reliability:* It refers to the scenario where uploading and downloading activities of the users should be minimally influenced due to some server failures or data losses. The proposed framework applies the EC technique to design a fault-tolerant system that provides both key and data reliability.
4. *Secure-deduplication:* In the proposed system, CSP does not store multiple ciphertext forms of the same data. It applies the file level deduplication on the encrypted data to reduce the extra computation and space overhead at $compute$ servers.
5. *Key Security and Management:* A key should be secure, and it must not be decryptable even if an adversary compromises the key servers. In the proposed architecture, a large number of keys are created during the uploading of a file by the cloud users. To secure the keys, proposed framework encrypts and stores them in a distributed fashion. The key management is achieved by tracking the address of each fragmented key stored at distributed servers.

5. Proposed $dualDup$ framework

This section presents an overview of our work and system model followed by a detailed description of the scheme.

5.1. Overview

Our proposed scheme $dualDup$ provides secure-deduplication and reliability for both data and key. It consists of three operations, namely, file uploading, downloading and deletion. File uploading scenario is discussed in the multitenant environment, in which multiple clients can upload the identical file to the datacenters. Compute Server receives the files to be uploaded in encrypted form. Since all computations are performed on encrypted data, it enables fully homomorphic encryption [32], achieving confidentiality and privacy. Deduplication problem is handled by both TTP and Compute Server, in different scenarios. If an identical file is uploaded multiple times by the same client, then TTP handles deduplication. Moreover, if an identical file is uploaded multiple times by different clients, then Compute Server manages deduplication.

The proposed scheme uses the EC technique, which partitions data into γ number of fragments. EC encoding module divides these γ fragments into α number of data fragments and β number of parity fragments, i.e., $\gamma = \alpha + \beta$. Compute Server stores all the γ fragments

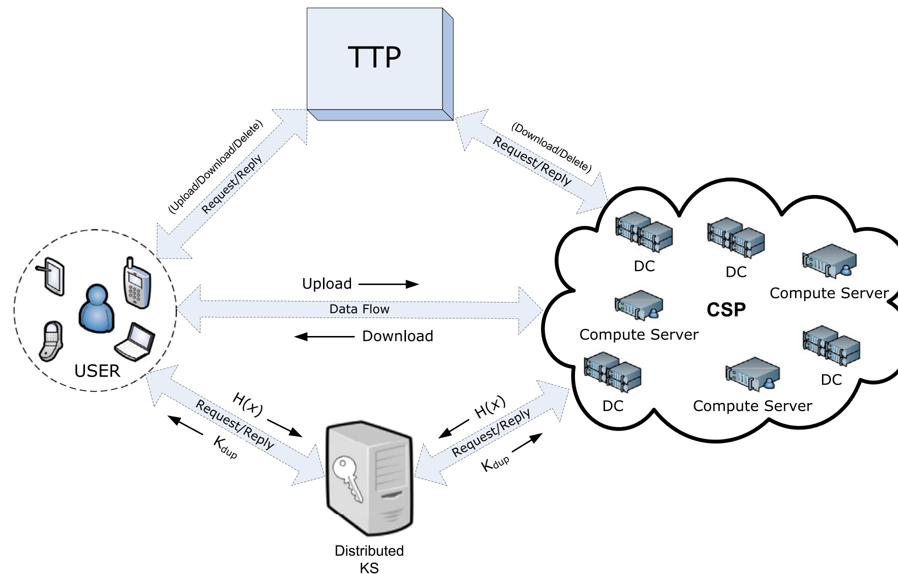


Fig. 5. The System Model Including Various Entities.

to distinct storage servers during uploading operation. However, to download the file, Compute Server requires any α number of data fragments, i.e., it needs any α number of active storage servers. Even if any β number of servers fail to provide the service, still EC decoding module can reconstruct the actual data from any α number of the data fragments. This reconstruction of the data even in the failure of β number of servers provides reliability to the proposed scheme.

5.2. System components

A representative system model of the proposed scheme is illustrated in Fig. 5. The model consists of the following entities:

- *Client/User*: an entity that can perform three operations: file upload, file download, and file delete in the cloud. It can either be an individual client or an enterprise. In this paper, we use the words *client* and *user* interchangeably.
- *Key Server (KS)*: an entity, that generates a dupless key k_{dup} corresponding to the received hash value of the data.
- *Trusted Third Party (TTP)*: a trusted entity, that stores the client file's information, handles deduplication from the same client, and forwards the download and delete requests from the user to the Compute Server.
- *Cloud Datacenters (DC)*: an entity, which is operated by CSP to store/retrieve the user data.
- *Compute Node/Server (CN)*: an entity, that uploads the client's file to the DC's, and download/delete the file from the DC's, as per the user request.

In our system model, CN can access the CSP database as per the granted permission, and TTP Node can access the TTP database as per the granted permission. Both user and CN communicate with KS to generate their corresponding dupless keys by sending the hash value of their input. The user sends its upload, download, or delete request to the TTP, which replies with the server id (S_{id}) to the client. User uploads the file to the corresponding server (S_{id}). Further, TTP forwards the download or delete request to the CN, which handles the delete request, and sends the corresponding file to the client for the download request.

Our work considers the following assumptions. First, we assume that all the connections between the entities use the secure protocol (such as IPSec or SSL/TLS) to ensure secure communication between entities. Second, our work uses the secure cryptographic hash algorithm in the generation of the data and key locators. Hence, we assume that the used hash function is strongly collision-resistant and produces a longer output which is harder to break. Further, we discuss all the operations in detail in the next subsections.

5.3. Upload procedure

In the file upload procedure, four entities are primarily involved which are Client, TTP, CN, and DC. Initially, the client is registered with TTP to get a particular cloud service where TTP has a list of CSP and corresponding *compute* servers. Fig. 6 describes the file uploading operations, which are divided into two phases, as discussed below.

5.3.1. Phase I: Client side operations

In this phase, the client calls the upload function $Up_f(\eta)$ which takes file, η, as an input, and performs the following operations:

1. The function retrieves the information: Client Id (U_{id}), TTP Id (T_{id}), Server Id (S_{id}) and secret key (s), from the Client Node.
2. It calls the function $R_f(\eta)$ to extract the file content, x, from the file, η.
3. Then it calls the function $g_d(x)$ which interacts with the Key Server KS to obtain a dupless key k_{dup}. For this computation, $g_d(x)$ performs the following steps:
   (a) It computes the hash value, H_x by using Secure Hash Algorithm (SHA-256), and sends it to the KS.
   (b) The KS generates a dupless key k_{dup} by using DupLESS protocol defined in [6].
4. The function encrypts x using the key k_{dup} to produces ciphertext C_d.
5. It further encrypts k_{dup} using the key s into ciphertext C_k.
6. It calls the algorithm $TagGen$ which takes C_d as an input and generates the corresponding locator l_d.
7. It again calls the algorithm $TagGen$ with C_k as an input and gives the locator l_k as an output.

The operations in the steps 4 *and* 5 are executed in parallel. Similarly, the operations in the steps 6 *and* 7 are executed in parallel. After performing the above operations, the function $Up_f(\eta)$ generates C_d, C_k, l_d, and l_k as outputs. The client creates a tuple $\langle U_{id}, \eta, S_{id}, l_d, l_k \rangle$, and sends it to TTP. A client does not store any tuple into its DB, it forward all the tuples to the TTP, hence the client is space-efficient.

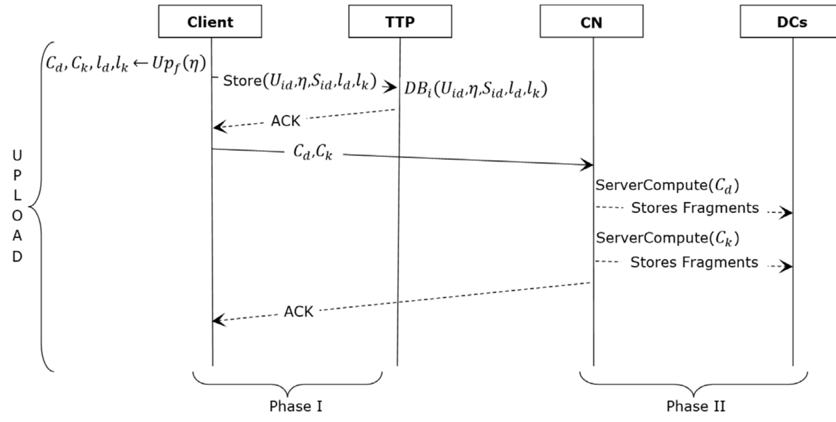


Fig. 6. File Uploading Operations.

Now, TTP checks whether the received tuple exists in its DB or not. If the tuple exists, then it updates the file counter value in its DB corresponding to the received tuple, and informs the client that the corresponding file has been stored on the cloud. Otherwise, TTP inserts the received tuple into its DB, and informs the client to send the tuple $\langle C_d, C_k \rangle$ to the CN. CN also acknowledges the client by sending an ACK, and further processes the received ciphertexts C_d and C_k in Phase II.

The proposed framework achieves the inside-user deduplication at TTP level if the same client attempts to upload an identical file more than once. In this case, TTP updates the corresponding file counter value in its DB to save the unnecessary processing overhead. Moreover, it also achieves confidentiality as the client sends the data and the key in encrypted form to the CN.

5.3.2. Phase II: Server side operations

In this phase, the CN fragments the received ciphertexts C_k and C_d using EC technique and stores them to the storage servers. For this computation, the CN calls the function $ServerCompute()$. The function is executed for both C_k and C_d parallelly, and is defined in Algorithm 1.

Algorithm 1 takes ciphertext C as an input and calls the algorithm $TagGen$ that computes locator l for C. If l exists already in the CN's database (DB), then there is no need to store the content to the Cloud Storage Server (CSS). In such a case, the algorithm increments the file counter n_0 by 1 in the DB, that represents the number of clients which have the same file contents.

If l does not exist in the DB, then the algorithm performs the following operations:

1. It generates a unique file name for the received ciphertext C, using $Base64$ or the preferred encoding. Let η_c and η_k be the file names generated for C_d and C_k, respectively.
2. It calls the function f_{ks} that computes the hash value of contents in C, i.e., H_C, by using Secure Hash Algorithm (SHA-256), and sends it to the KS.
3. The KS generates a dupless key k by using DupLESS protocol defined in [6]. It further applies EC technique to partition k into equal sized fragments and stores them in a set δ_k.
4. Subsequently, it invokes the function f_{ds} to partition C into equal sized fragments using EC technique. These fragments are stored in a set δ_d.
5. Now, the function $f_a(dss)$ is called to get the list of available servers, which is stored in a set A^1 and known as set of Data Storage Servers (DSS).
6. It encrypts each data fragment δ_{d_i}, where $\delta_{d_i} \in \delta_d$, with dupless key k to get the resultant ciphertext C_i.

Algorithm 1: Compute Node Stores the Fragments to the Datacenters.

```
   Input : Ciphertext C
   Output: Success/Failure ACK
1  begin
2      l ⟵ TagGen(C)
3      if l ∈ DB then
           /* Handle cross-user deduplication                          */
4          DB_u (n_0 = n_0 + 1) corresponding to l
5          return Success
6      end
7      Compute η_c, η_k
8      δ_k, k ⟵ f_ks(C)
9      δ_d ⟵ f_ds(C)
10     A^1 ⟵ f_a(dss)
       /* Data fragment storage procedure starts                       */
11     Let N_1 denote the number of element in δ_d
12     for i = 1 to N_1 do
13         C_i ⟵ E(k, δ_{d_i})
14         Put_f^{A^1_i} (η_c, C_i) where A^1_i ∈ A^1
15         S^d.append(A^1_i)
16         A^1.remove(A^1_i)
17     end
       /* Data fragment storage procedure ends                         */
18     A^2 ⟵ f_a(kss)
       /* Key fragment storage procedure starts                        */
19     Let N_2 denote the number of element in δ_k
20     for j = 1 to N_2 do
21         C_j ⟵ E(P_k, δ_{k_j})
22         Put_f^{A^2_j} (η_k, C_j) where A^2_j ∈ A^2
23         S^k.append(A^2_j)
24         A^2.remove(A^2_j)
25     end
       /* Key fragment storage procedure ends                          */
       /* Operations in the steps 12 − 17 and 20 − 25 are executed in
          parallel                                                     */
26     DB_i(l, S^d, S^k, n_0 = 1)
       /* S^d and S^k represent the list of servers where data and
          key fragments are stored, respectively                       */
       /* The algorithm returns a success status in case of no
          error occurred during the execution of the operations */
27     return status
28 end
```

7. It further calls the function $Put_f^{A^1_i}$ which stores each C_i to one of the available server from the set A^1. Then it appends the server address A^1_i to the server list S^d and removes A^1_i from the set A^1.
8. The algorithm calls the function $f_a(kss)$ to get a separate list of available servers for key fragments stored in set A^2 such that $A^1 \cap A^2 = \phi$. The set A^2 is called as a set of Key Storage Servers (KSS).

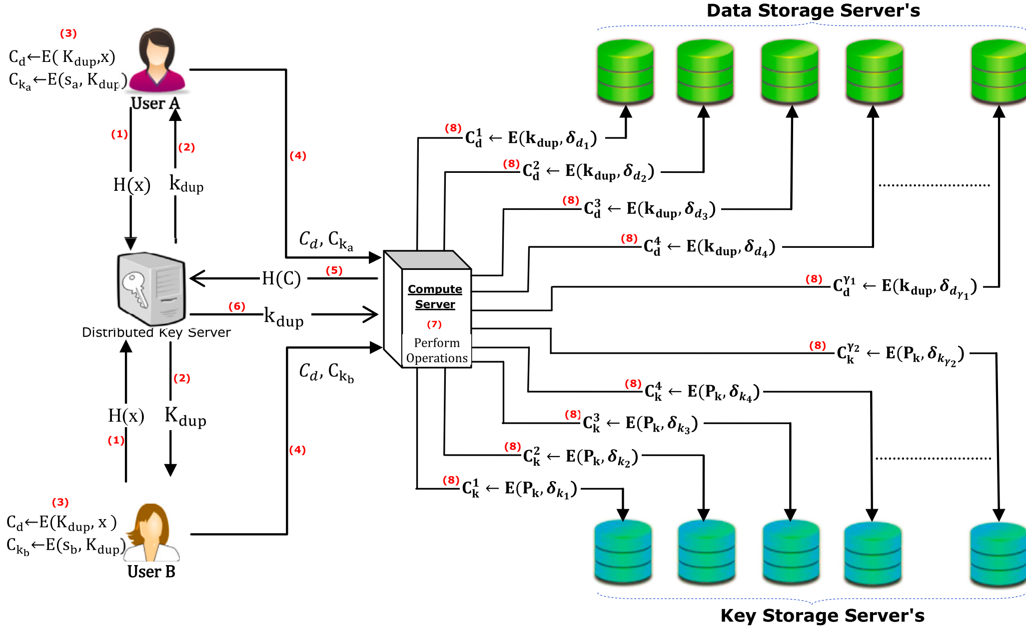


Fig. 7. Scenario of Uploading Same File Content x from $User\ A$ and $User\ B$, where Number in Red Represents the Sequence of Operations.

9. It encrypts each key fragment δ_{k_j}, where $\delta_{k_j} \in \delta_k$, with private key of CN, i.e., P_k, to get the resultant ciphertext C_j.
10. It further calls the function $Put_f^{A_j^2}$ which uploads each C_j to one of the available server from the set A^2. Then it appends the server address A_j^2 to the server list S^k and removes A_j^2 from the set A^2.
11. The algorithm finally inserts a tuple $\langle l, S^d, S^k, n_0\rangle$ into the database of CN and returns. S^d and S^k represent the list of servers where data and key fragments are stored, respectively.

If the data already exists in cloud storage, then CN updates the file counter entry, corresponding to the received locator, in its DB, to achieve cross-user deduplication at the server level. Otherwise, it encrypts each fragment and stores them to the distinct γ number of storage servers to achieve confidentiality and reliability.

5.3.3. File uploading scenario

The file uploading scenario from multiple clients simultaneously to datacenters is illustrated in Fig. 7. Suppose more than one user, say, user A and user B, wants to upload the same file content x to the cloud. As discussed in Section 5.3.1, both of the client nodes interact with KS to get the dupless key k_{dup}, and encrypt x with k_{dup} to generate the identical ciphertext, C_d. Both A and B further encrypt k_{dup} with their own secret keys to generate C_{k_a} and C_{k_b} respectively.

CN receives the tuple $\langle C_d, C_{k_a}\rangle$ from user A and $\langle C_d, C_{k_b}\rangle$ from user B. All the unique ciphertexts, i.e., C_d, C_{k_a}, and C_{k_b} are given as an input, separately, to the algorithm 1, which partitions the received data into fragments. All the fragments of the set δ_d : $\{C_d^1, C_d^2, C_d^3, \dots, C_d^{\gamma_1}\}$ are stored in the distinct available DSS. Similarly, all the fragments of the set δ_k : $\{C_k^1, C_k^2, C_k^3, \dots, C_k^{\gamma_2}\}$ are stored in the distinct available KSS.

5.4. Download procedure

In the file download procedure, three entities are primarily involved which are Client, TTP, and CN. Fig. 8 describes the file download operations. Suppose that a client wants to download a file η from the cloud. The following steps are involved in the download procedure:

1. Client retrieves its Client Id (U_{id}), TTP Id (T_{id}), and secret key (s) from its DB, and calls the function $G_{fl}(U_{id})$ that sends U_{id} to TTP.
2. TTP returns all the file names, along with their locators, that have been uploaded by the client. The tuple $\langle \eta, l_d, l_k\rangle$ represents the filename (η), data locator (l_d), and key locator (l_k). It sends a list of all such tuples corresponding to U_{id}.
3. Out of all the tuples received, client selects the required download tuple. Let the selected i^{th} tuple be $\langle \eta_i, l_{d_i}, l_{k_i}\rangle$. It further inserts $\langle \eta_i, l_{d_i}\rangle$ into the request queue Q_R.
4. It sends the tuple $\langle \eta_i, l_{d_i}, l_{k_i}\rangle$ to the TTP, which finds the S_{id} of the CN corresponding to the tuple $\langle U_{id}, \eta_i, l_{d_i}, l_{k_i}\rangle$, and sends an ACK back to the client. If the received tuple is not found in TTP's DB, then it cannot process the client's download request, and further notifies the client.
5. TTP further sends the tuple $\langle U_{id}, l_{d_i}, l_{k_i}\rangle$ to the CN, to get the corresponding ciphertexts of both data and key, which are being referred by locators l_{d_i} and l_{k_i} respectively. CN also acknowledges the receipt of the tuple by sending an ACK back to the TTP.
6. For each locator $l \in \{l_{d_i}, l_{k_i}\}$, CN calls the function $Compute(l)$ that computes the ciphertexts C_d and C_k respectively. CN further communicates the tuple $\langle C_d, C_k\rangle$ to the Client Node. Algorithm 2 describes the working of $Compute(l)$ function which will be discussed in Section 5.4.1.
7. Client calls the algorithm $TagGen(C_d)$ to generate the corresponding locator l_d.
8. It searches l_d in Q_R to get the corresponding filename η and removes this tuple from Q_R.
9. It decrypts C_k with secret key s to obtain the dupless key k.
10. It further decrypts C_d with dupless key k to obtain the file content x.
11. Finally, it calls the write function $W_f(\eta, x)$ that writes the file content x into the file named η.

5.4.1. Compute ciphertext from datacenters

Algorithm 2 takes locator l as an input, and returns the corresponding ciphertext stored across the cloud DCs'. The algorithm performs the following operations:

1. Initially, it retrieves the list of servers, S^d and S^k, where data and key fragments are stored, respectively, corresponding to locator l.

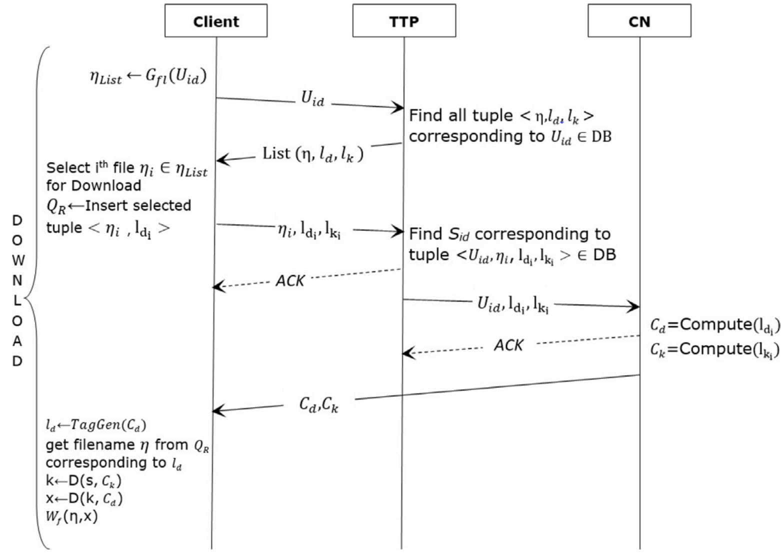


Fig. 8. File Downloading Operations.

Algorithm 2: Compute Ciphertext from Datacenters.

Input : *locator* l
Output: *ciphertext* C

1 **begin**
2 *Find* S^d, S^k *corresponding to* $l \in DB$
3 *Compute* η_c, η_k
4 $count = 0$
/* Key computation procedure starts */
5 *Let* N_1 *denote the number of element in* S^k
6 **for** $i = 1$ *to* N_1 **do**
/* Run parallelly for any α_1 out of γ_1 number of servers */
7 **if** $count < \alpha_1$ **then**
8 $C_{\delta_i} \longleftarrow Get_f(\eta_k)$
9 $\delta_{k_i} \longleftarrow D(P_k, C_{\delta_i})$
10 $\delta_k.append(\delta_{k_i})$
11 $count = count + 1$
12 **end**
13 **else**
14 *break*
15 **end**
16 **end**
17 $k \longleftarrow decode(\delta_k)$
/* Key computation procedure ends */
18 $count = 0$
/* Ciphertext computation procedure starts */
19 *Let* N_2 *denote the number of element in* S^d
20 **for** $i = 1$ *to* N_2 **do**
/* Run parallelly for any α_2 out of γ_2 number of servers */
21 **if** $count < \alpha_2$ **then**
22 $C_{\delta_i} \longleftarrow Get_f(\eta_c)$
23 $\delta_{d_i} \longleftarrow D(k, C_{\delta_i})$
24 $\delta_d.append(\delta_{d_i})$
25 $count = count + 1$
26 **end**
27 **else**
28 *break*
29 **end**
30 **end**
31 $C \longleftarrow decode(\delta_d)$
/* Ciphertext computation procedure ends */
32 *return* C
33 **end**

2. Then it generates the unique filename for the stored ciphertext C using Base64 encoding. Let η_c and η_k be the filenames for data fragment and key fragment, respectively, for a particular locator l.
3. Let γ_1 be the total number of servers where all the key fragments are stored, and let α_1 be the minimum number of servers required to regenerate the dupless key k. Note that each key fragment is stored at distinct KSS. The algorithm performs the following steps, in parallel, at α_1 number of servers:
   (a) It calls the function $Get_f(\eta_k)$ to get the encrypted fragment C_{δ_i}.
   (b) It further decrypts C_{δ_i} with private key P_k of the CN to get the key fragment δ_{k_i} and append this fragment to the key fragment set δ_k. Hence, we will have α_1 number of key fragments available at the end of this step.
4. Decoding module of the EC technique is applied on δ_k to regenerate the dupless key k.
5. Let γ_2 be the total number of servers where all the data fragments are stored and let α_2 be the minimum number of servers required to regenerate the stored ciphertext C. Note that each encrypted data fragment is stored at distinct DSS. The algorithm performs the following steps, in parallel, at α_2 number of servers:
   (a) It calls the function $Get_f(\eta_c)$ to get the encrypted fragment C_{δ_i}.
   (b) It further decrypts C_{δ_i} with dupless key k to get the encrypted data fragment δ_{d_i} and append this fragment to the data fragment set δ_d. . Hence, we will have α_2 number of encrypted data fragments available at the end of this step.
6. Decoding module of EC technique is applied on δ_d to regenerate the ciphertext C, and returns C.

CN retrieves the stored fragments from any α number of storage servers to reconstruct the actual data, to achieve reliability. Further, it can tolerate up to $\beta = \gamma - \alpha$ number of server failures or fragments loss.

5.5. Delete procedure

In the file delete procedure, three entities are primarily involved: Client, TTP, and CN. Fig. 9 describes the file delete operations. Suppose that a client wants to delete a file η from the cloud. The following steps are required in the deleting process:

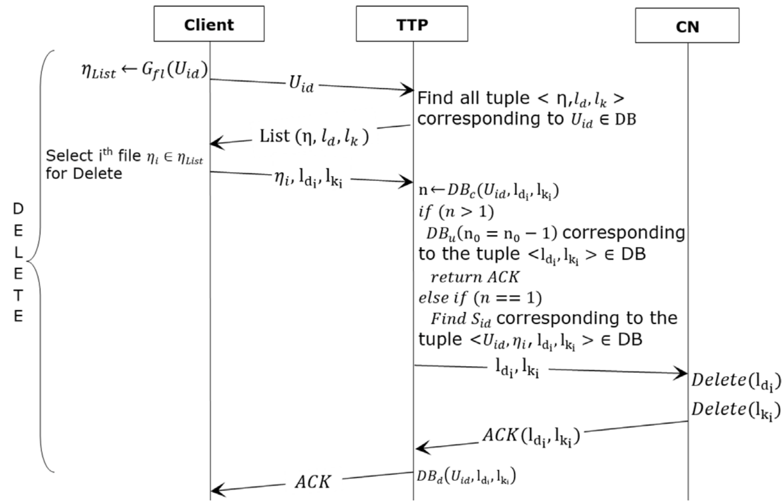


Fig. 9. File Deletion Operations.

1. Client retrieves its Client Id (U_{id}) and TTP Id (T_{id}) from its DB, and calls the function $G_{fl}\left(U_{id}\right)$ that sends U_{id} to TTP.
2. TTP returns all the file names, along with their locators, that have been uploaded by the client. The tuple $\langle \eta, l_d, l_k \rangle$ represents the filename (η), data locator (l_d), and key locator (l_k). It sends a list of all such tuples corresponding to U_{id}.
3. Out of all the tuples received, client selects the file to be deleted. Let the selected i^{th} tuple be $\langle \eta_i, l_{d_i}, l_{k_i} \rangle$.
4. It sends the tuple $\langle \eta_i, l_{d_i}, l_{k_i} \rangle$ to the TTP. Further, TTP calls the function $DB_c\left(U_{id}, l_{d_i}, l_{k_i}\right)$ to retrieve the stored file counter value n, which represents the number of copies of the file referred by locators l_{d_i} and l_{k_i} that were uploaded by client U_{id}. If $n > 1$, then TTP decrements the corresponding file counter value by 1, updates it in the DB, and sends a file deletion ACK back to the client. Otherwise, the delete operation proceeds with the following steps.
5. TTP finds the S_{id} of the CN corresponding to the tuple $\langle U_{id}, \eta_i, l_{d_i}, l_{k_i} \rangle$. It sends the tuple $\langle l_{d_i}, l_{k_i} \rangle$ to the CN to delete the corresponding ciphertexts of both data and key. If the received tuple is not found in TTP's DB, then it cannot process the client's delete request, and further notifies the client.
6. For each locator $l \in \{l_{d_i}, l_{k_i}\}$, CN calls the function $Delete(l)$ that deletes both the ciphertexts C_d and C_k. Algorithm 3 describes the working of $Delete(l)$ function, which will be discussed in Section 5.5.1.
7. After deleting C_d and C_k, CN sends an ACK back to the TTP, along with tuple $\langle l_{d_i}, l_{k_i} \rangle$.
8. TTP removes the corresponding file entry from the DB and sends a file deletion ACK back to the client.

The proposed framework handles the scenario of deleting the file at the TTP level if the file is uploaded multiple times by the same client. In this case, TTP decrements the corresponding file counter entry, n, by 1 in its DB, if $n > 1$, to save the unnecessary processing overhead. TTP checks the entry of the received file tuple in their database (i.e., step 4 in the download procedure and step 5 in the delete procedure) to find the identity of the storage server corresponding to the received tuple. The following are some of the scenarios that could occur: (i) If a client forwards the incorrect file locators to the TTP, then the selected file locator is not found in the TTP database. (ii) Suppose a client saves some file tuples and later uses them to initiate a file download or delete request. If the file has already been deleted, it will not be found in the TTP database.

TTP will only process a user request to download or delete a file if the file already exists in the system. Therefore, the TTP checks the file locator entry in their database before processing the client's request.

Algorithm 3: Delete File from Datacenters.

Input : *locator* l
Output: ACK

1 **begin**
2 $\quad n_0 \longleftarrow DB_c(l)$
3 $\quad$ **if** $(n_0 > 1)$ **then**
4 $\quad\quad DB_u(n_0 = n_0 - 1)$ *corresponding to* $l \in DB$
5 $\quad\quad$ return ACK
6 $\quad$ **end**
7 $\quad$ *Find* S^d, S^k *corresponding to* $l \in DB$
8 $\quad$ *Compute* η_c, η_k
$\quad$ /* Key fragment deletion procedure starts */
9 $\quad$ *Let* N_1 *denote the number of element in* S^k
10 $\quad$ **for** $i = 1$ *to* N_1 **do**
11 $\quad\quad Del_f^{S_i^k}(\eta_k)$
12 $\quad$ **end**
$\quad$ /* Key fragment deletion procedure ends */
$\quad$ /* Data fragment deletion procedure starts */
13 $\quad$ *Let* N_2 *denote the number of element in* S^d
14 $\quad$ **for** $j = 1$ *to* N_2 **do**
15 $\quad\quad Del_f^{S_j^d}(\eta_c)$
16 $\quad$ **end**
$\quad$ /* Data fragment deletion procedure ends */
$\quad$ /* Operations in the steps 10 − 12 and 14 − 16 are executed in parallel */
17 $\quad DB_d(l)$ *delete from DB*
18 $\quad$ *return* $ACK\ (l_d, l_k)$
19 **end**

5.5.1. Delete file from datacenters

The Algorithm 3 takes locator l as an input and deletes all the fragments, referred by l, from the corresponding DC. The algorithm performs the following operations:

1. First it calls the function $DB_c(l)$ to get the file counter n_0, which represents the number of copies of the file referred by a locator l.
2. If $n_0 > 1$, then n_0 is decremented by 1 and the algorithm returns. Else, the algorithm proceeds with the following steps.
3. It retrieves the list of servers, S^d and S^k, where data and key fragments are stored respectively corresponding to locator l.
4. Then it generates the unique filename for the stored ciphertext C using Base64 encoding. Let η_c and η_k be the filenames for data fragment and key fragment respectively for a particular locator l.
5. It calls the function $Del_f^{S_i^k}(\eta_k)$ which deletes all the key fragments that are stored at the servers, existing in the list S^k.

6. It calls the function $Del_f^{S_i^d}(\eta_c)$ which deletes all the data fragments that are stored at the servers, existing in the list S^d.
7. Finally, it deletes the entries corresponding to the locator l from the DB, and returns an ACK with the tuple $\langle l_d, l_k \rangle$ to the TTP.

CN updates its DB only if more than one copy of the data exists in its DB, corresponding to the given file locator. This process avoids unnecessary storage access.

6. Performance evaluation

We, first, analyze the security aspects of the proposed scheme as per the goals discussed in Section 4 to evade such attacks. Later, the implementation setup with details related to packages and other libraries. Next, we discuss the performance analysis of experiments conducted and their corresponding results. Finally, we conclude by providing the complexity analysis of different operations defined in the framework and comparative analysis of various deduplication schemes.

6.1. Security analysis

The proposed framework uses AES (or any other strong encryption algorithm alike) with a dupless key generated by the KS based on the hash value of the data and $OPRF$ protocol to encrypt the data. We presume that AES is secure [33,34]. Therefore, the proposed framework is secure and achieves confidentiality as long as the encryption algorithm works fine. Besides, our scheme uses $OPRF$ protocol between KS and $Users$. It ensures that KS learns nothing about the user data and the user learns nothing about the key. However, only authentic users can access the KS thus the external attackers cannot compromise the KS for user data. The Rate Limiting Protocol slows down the brute-force attacks from compromised clients. So we can argue that our framework provides compromise resilience to KS.

The insider and outsider adversaries try to recover the plaintext data from the cloud storage. In this section, we analyze the security aspects of the proposed scheme as per the goals discussed in Section 4 to evade such attacks.

6.1.1. Security from insider adversaries/CSP

Suppose CSP or other admin insiders want to retrieve user sensitive information from the stored data in the cloud. The adversary fails to retrieve the information in plaintext because it receives the data in ciphertext which is encrypted with k_{dup} using DupLESS [6] algorithm. It is a brute-force resistance algorithm and denies online attacks by adopting Rate Limiting Protocol [27]. Moreover, suppose it performs some illegal activity like data leakage or misuse, etc. The encrypted form of users' data and key $\{C_d, C_k\}$ ensures the privacy of the users' data in spite of the data is leaked to an untrusted party.

6.1.2. Security from malicious users or outsider adversaries

The proposed framework stores the encrypted data in fragments at distributed sites, and it is unpredictable for any outsider to locate the fragments. Such encrypted data copies are semantically secure and provide privacy against chosen distribution attacks [4]. Suppose that any outsider gets track of all the data fragments $\{C_d^1, C_d^2, C_d^3, \dots, C_d^{\gamma_1}\}$ from the storage servers to recover the C_d. The adversary needs the corresponding key C_k to recover the dupless key K_{dup}. Since the key C_k is also fragmented and stored at distributed locations, it is difficult for the adversary to extract C_k. Even if the adversary gets all the key fragments $\{C_k^1, C_k^2, C_k^3, \dots, C_k^{\gamma_2}\}$ to recover C_k, it is not possible to decipher them as it requires user's secret key. Even if the attackers compromise the external entities, i.e., KS & TTP, they cannot gain any access to the user data as the KS deals with an only hash value of the user data and the TTP deals with only the data and key locators. Further, suppose that any malicious user compromises the TTP to initiate a false download request with its id as U_{id} to get C_d & C_k from the storage servers. Malicious users fail to decipher them as they are in encrypted form.

Therefore, we argue that the proposed system is secure from both insider and outsider adversaries.

6.2. Performance analysis

The core of the implementation of *dualDup* prototype is based on implementation of DupLESS and cryptographic techniques [6,33,35].

Experimental Setup: The testing environment in the implementation framework comprises a system with Intel® $Core^{TM}$ i7-3770 CPU @ 3.40 GHz processor having 8 cores, 8 GB RAM, Ubuntu 14.04 64-bit operating system. The software tools include Python2.7-dev package and various Python-based libraries for each entity, MySQL for the database operations, SHA-256 for generating cryptographic hash value, PyECLib-1.2.0 library with liberasurecode-dev for generating erasure codes, and AES for symmetric-key encryption/decryption algorithms via Python's crypto library. Any customizable data center can be used for storing and retrieving the files. Our implementation consists of multiple modules including client, TTP, KS, and CN. Several Python scripts are developed to implement these modules i.e., Client, KS, TTP, and CN.

Implementation: Our implementation adopts Dropbox as a datacenter. Multiple Dropbox accounts are used to emulate the storage servers. Out of those accounts, some are used as data storage servers to store data fragments and the remaining accounts as storage servers to store key fragments.

The client sends the commands from the developed command-line interface for uploading, downloading, deleting and accessing the stored files. For instances, the client uses $upload < filename >$ to upload a file to the Dropbox, $download < filename >$ to download a file from the Dropbox, ls command to see the list of stored file names in the Dropbox, and $delete < filename >$ to remove a file from the Dropbox. An app created for each Dropbox account generates corresponding APP_KEY and APP_SECRET for accessing permissions, eventually to access the files from Dropbox. Further, we generate self-signed certificate via OpenSSL to authenticate the key server and other entities.

We consider a dataset S of various files of different sizes generated using a script for analyzing the execution time during different operations on the top of the Dropbox datacenter as shown in Eq. (1).

$$S = \sum_{i=0}^{max} 2^{2i} \text{ Kilobytes.} \tag{1}$$

In the experiments, we took $max = 9$ leading to generation of files with random contents of size 2^{2i} KB, $0 \le i < 10$, i.e., different files of size 1 KB to 256MB. Further, we create multiple copies of these datasets for testing the deduplication. In our experiments, tests are performed by continuously sending requests to perform the operations such as upload, and download to the datacenter via TTP. Python *timeit* module is used to evaluate the computation results of various operations.

In the next few subsections, we analyze the performance of our proposed framework for the time taken for encryption/decryption of key and data, for uploading and downloading of files, for the encoding of data and key, and the generation of keys of various files of different sizes. We also analyze the space overhead for each distinct file.

6.2.1. Encryption and decryption cost analysis

The execution cost of cryptographic operations including encryption and decryption are analyzed by repeating the experiments 1000 times on input datasets defined in Eq. (1). There must be an impact of variation of distinct files of different sizes on encryption and decryption operations. This study aims to figure out the minimum, mean, maximum, and standard deviation (Std. Dev.) of execution times due to variation of file sizes. Fig. 10 shows the execution time of encryption and decryption operations for both data and their corresponding keys. Encryption and decryption of data, shown in Fig. 10(a) and 10(c),

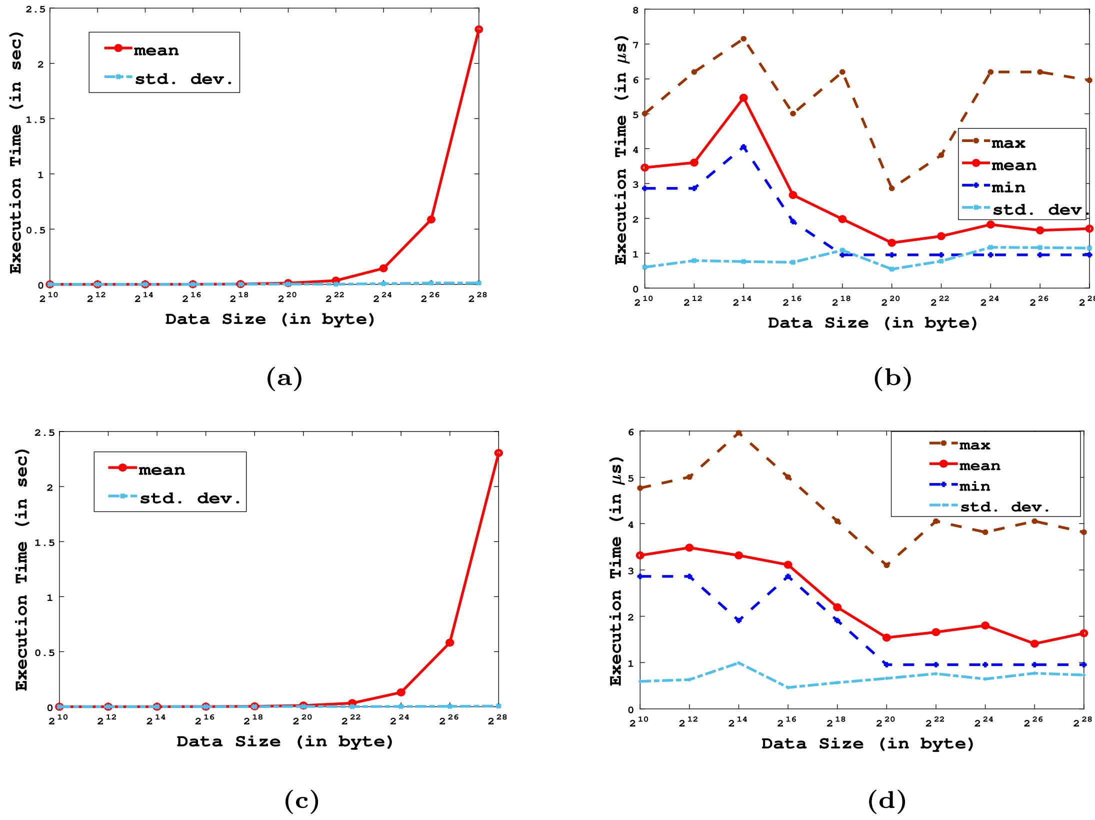


Fig. 10. Execution Time of Cryptographic Operations of Various File sizes. (a) Execution Time of Encryption Operation over Data. (b) Execution Time of Encryption Operation over Key. (c) Execution Time of Decryption Operation over Data. (d) Execution Time of Decryption Operation over Key.

takes constant time (< 0.1 s) for data size up to 2^{22} bytes, and then the time increases exponentially with the increase in file size. Since the size of the key is very small and independent of data, encryption and decryption of keys are measured in microseconds, as shown in Fig. 10(b) and 10(d). These figures show that the average execution time for encrypting and decrypting the keys lies between 1 to 6 μs, and between 1 to 4 μs, respectively. As can be seen in the figures, the standard deviation of these cryptographic operations is nearly zero for data and less than one for the key, which represents that the variation in execution time of these operations for different file sizes is low.

6.2.2. Upload and download time analysis

We evaluate and discuss the execution time of Upload and Download operations for the following possible deduplication circumstances. Suppose distinct files uploaded by m users to the cloud takes equivalent execution time as uploading an individual file from each user. When the existing user (inside-user deduplication) attempts to upload the previously uploaded file to the storage, as a result, no need to execute uploading operation. We can say that it takes constant time to upload the file. If any other users (cross-user deduplication) uploads the file to the server, the CN receives the file and subsequently checks if the file already exists in the storage. Then, CN updates a file counter with the user id. In any case, users need to upload file to the cloud storage. We measure upload and download operation time for the datasets S using *timeit* module. We repeat these operations 100 times to evaluate minimum, mean, maximum and standard deviation of the operations time on distinct files of different sizes. Fig. 11 shows the execution time of uploading and downloading operations for both data and their corresponding keys. Fig. 11(a) and 11(c) show time required to upload and download the dataset S. Both graphs show the linear operation time up to data size of 2^{22} bytes, then increases exponentially. Uploading and downloading time of the keys corresponding to their data are shown in Fig. 11(b) and 11(d). These figures show that the average time for uploading and downloading the keys lies between 0.5 to 1.2 s, and between 0.4 to 1.2 s, respectively for the given data inputs. As can be seen in the figures, the standard deviation of upload time for data and key is nearly zero and less than one, respectively, which represents that the variation in upload time for different file sizes is low. Similarly, the standard deviation of download time for the key is less than one. However, the standard deviation of download time for data increases exponentially after 16MB file size.

6.2.3. Encoding and key generation time analysis

Figs. 12 (a) and 12 (b) show the time required for encoding the dataset S and its key respectively, using Reed–Solomon, $RS\,(\alpha = 3, \beta = 2)$ encoding scheme. We repeat these encoding operations 1000 times to evaluate minimum, mean, maximum and standard deviation of the encoding time. Since the key size is small, its average key encoding time lies between 7 to 12 μs. As can be seen in the figures, the standard deviation of encoding time for data and key is nearly zero and less than five respectively which represents the variation in encoding time for different file sizes is low.

Fig. 13 shows the time required to generate the keys corresponding to the input data of various sizes. We measure this time by excluding the network overheads. As can be seen in the figure, the average key generation time increases exponentially after 16MB file size. Standard deviation is nearly zero, which represents that the variation in key generation time for different file sizes is low.

6.2.4. Space and time analysis of RS code

Table 2 summarizes the space overhead $\left(S_{oh}\right)$ of various input file sizes ranging from 1KB to 256 MB. We analyze the results corresponding to the *Reed–Solomon*, $RS\,(\alpha = 3, \beta = 2)$ codes which can tolerate maximum of $\beta = 2$ missing fragments, hence increasing the data reliability.

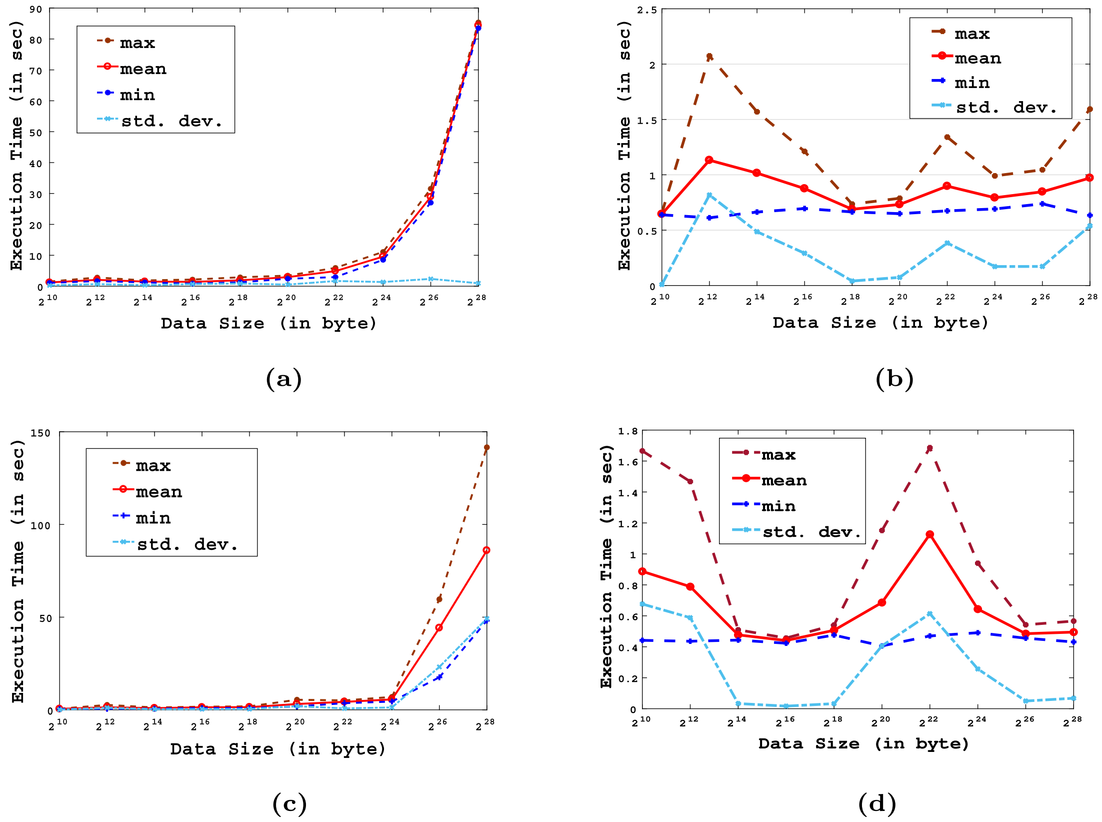


Fig. 11. Execution Time for Upload and Download Operation of Various File sizes. (a) Execution Time for Uploading Data. (b) Execution Time for Uploading Key. (c) Execution Time for Downloading Data. (d) Execution Time for Downloading Key.

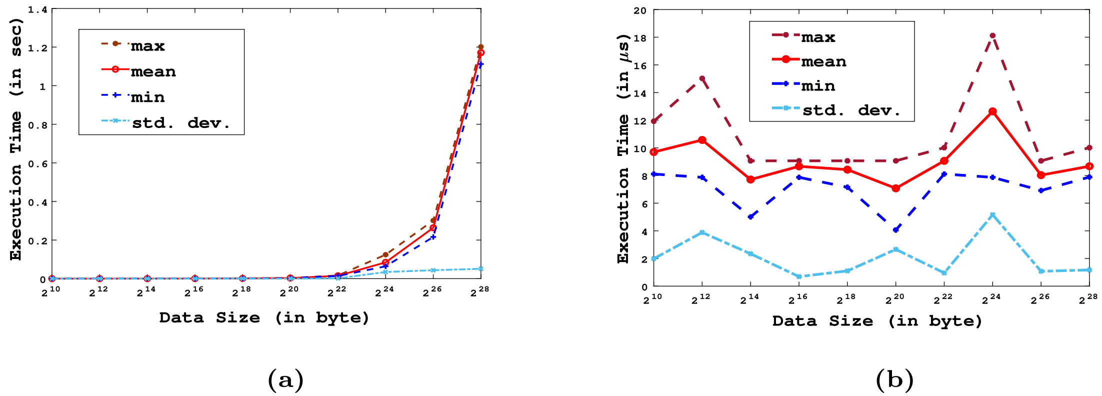


Fig. 12. (a) Running Time of Data Encoding. (b) Running Time of Key Encoding.

Average space overhead $\left(\overline{S_{oh}}\right)$ is calculated, as given in Eq. (2).

$$\overline{S_{oh}} = \left(\frac{\ell_{\delta_i} * (\alpha + \beta)}{Data\ Size} - 1\right) \times 100\ \% \tag{2}$$

where ℓ_{δ_i} represents a single fragment size in bytes, α represents the number of data fragments, and β represents the number of parity fragments. The space overhead is maximum for the smallest 1KB file, and it is almost the same when file size increases from 256KB to 256MB. The encoding library adds 80 byte header to each fragment. These overhead bytes significantly increase the size of the small files, whereas, it does not affect the large files. Therefore, small files have extra space overheads as compared to the large files. RS(3,2) coding generates an average space overhead of 66.66% which is better than N-way replication technique which generates space overhead of $O(N)$, where N denotes the number of replicas.

Average fragment size $\left(\ell_{\delta_{Avg}}\right)$ for different data size corresponding to RS codes is computed, as given in Eq. (3).

$$\ell_{\delta_{Avg}} \approx \frac{Filesize \times \left(1 + \frac{\overline{S_{oh}}}{100}\right)}{\alpha + \beta}\ \text{byte} \tag{3}$$

The execution time of operations is reduced at compute server due to the parallelism of the proposed architecture which depends on an adapted EC scheme. For example, suppose the compute server receives 3 MB file from the client, then it computes fragments of size nearly equivalent to 1 MB if we use $RS(3,2)$ codes. The time complexity of this 3 MB file is shown in Fig. 14 with and without EC. Fig. 14(a) and Fig. 14(b) compares the encryption/decryption time, and upload/download time for a 3 MB file with RS(3,2) encoding scheme. It can also be seen from the figure that time required for all

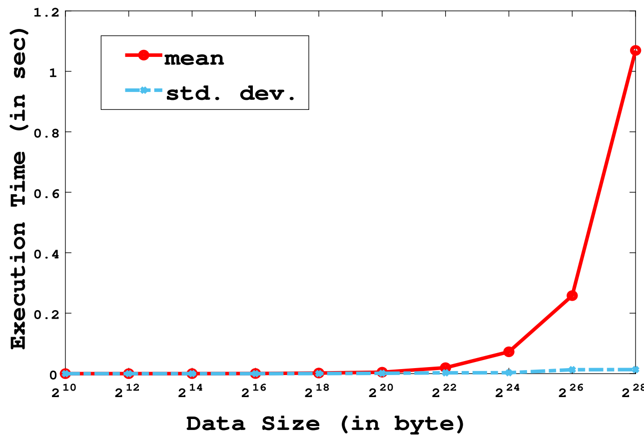


Fig. 13. Key Generation Time for Various File Flavors.

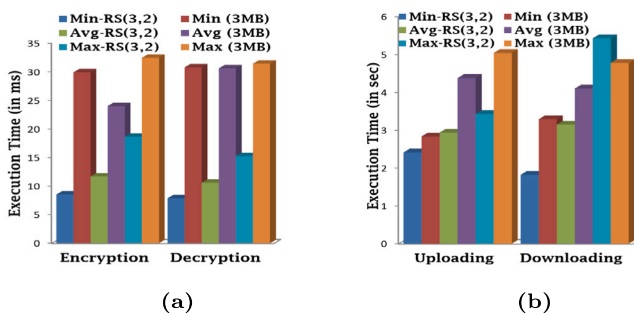


Fig. 14. Comparison of Normal Operation Time with Execution Time of RS(3,2) Codes for a 3 MB File.

these operations gets reduced if RS(3,2) encoding scheme is used for fragmenting the data.

6.2.5. Complexity analysis

In our implementation, when the compute server receives a unique file from the client, it makes an entry of the file's locator in its DB. If the locator does not exist in the DB, then it uploads the file. The adapted encoding scheme fragments the data, and data uploading time depends on the fragment size. Our proposed model adapts RS encoding that generates the set of encrypted fragments of data, and uploads each fragment in parallel. Reduction in uploading/downloading operation time depends on the adapted RS coding scheme. If the file locator already exists in the DB, then the proposed approach does not perform operations like evaluating encrypted file fragments, uploading, etc. It only increments the file counter entry (corresponding to an existing file locator) in the DB by 1. It takes constant time to upload any file and ensures deduplication at the datacenter. We use a hash index storage engine to perform the lookup, insert, delete, and update operation in constant time.

The complexity of the operations is discussed in Table 3. Suppose that all m users upload the distinct files (each of size n) in parallel, then the upload operation takes an equivalent time as the time required to upload a single file, i.e., $O(n)$. Space complexity of such a scenario is $O(mn)$. Inside-user deduplication detected by the TTP takes constant time and space and hence saves the server space of $O(n)$ if the file locator exists in the DB. Otherwise, it takes $O(n)$ time to upload the file and constant space to update its DB. The cross-user deduplication checks the file locator entry in its DB. If the corresponding entry exists, then the file storing operation takes constant time and space, hence saving $O(n)$ space. Otherwise, it takes $O(n)$ time and space to store the file. In the delete operation, inside-user deduplication is checked

Table 2
Space overheads and fragment sizes for various input file size corresponding to $RS(3,2)$ codes.

File size	ℓ_{δ_i} (bytes)	S_{oh} %
1 KB	422	106.0546875
4 KB	1446	76.51367188
16 KB	5542	69.12841797
64 KB	21926	67.28210449
256 KB	87462	66.82052612
1 MB	349606	66.70513153
4 MB	1398182	66.67628288
16 MB	5592486	66.66907072
64 MB	22369702	66.66726768
256 MB	89478566	66.66681692

by TTP. If the file exists in TTP's DB, then it takes constant time to delete the file because TTP has to decrement the counter value by one from its DB. Further, cross-user deduplication is checked using the CSP database. If the file counter is equal to one, then it takes $O(n)$ time in file deletion because it removes the whole file from the storage; else, if the file counter value > 1, then deletion operation is done in constant time because it decrements the file counter entry in the database.

6.2.6. Comparative analysis of various deduplication schemes

Table 4 compares the proposed approach with the existing secure-deduplication schemes in the aspects of deduplication, attack mitigation, key security and management, reliability, and various QoS features. The deduplication managing entity prevents the data redundancies in storage. In all other works, either client or server or both act as managing entity. Whereas, in our proposed work, TTP and CSP act as managing entity to defend against inside-user deduplication and cross-user deduplication respectively.

All the existing works except the one in [8] incur client storage overheads. None of them except [8] can achieve all QoS features of data confidentiality, compromise resilience, and reduction in bandwidth. However, the work in [8] fails to address the reliability. The strength of the proposed approach lies in achieving reliability without incurring client overheads. Moreover, the proposed framework achieves all QoS features of confidentiality, compromise resilience, and reduction in bandwidth.

None of the exciting works, except [6,17], can mitigate against both brute force and side-channel attacks. However, both related works [6, 17] fail to achieve key reliability. On the contrary, our *dualDup* framework mitigates against both brute force and side-channel attacks. Additionally, none of the works can target both reliability and key security together. The proposed work achieves key security and management in addition to reliability. Hence, we can conclude that the proposed approach is superior over other related works.

7. Conclusion and future work

This paper proposed a novel *dualDup* framework combining techniques such as DupLESS, Erasure Coding, and other cryptographic primitives to provide secure-deduplication, privacy, and reliability together unlike other models discussed in the literature. The proposed *dualDup* framework optimizes the storage by eliminating the duplicate encrypted data in the storage servers by extending the DupLESS concept and further achieved privacy and reliability for both data and key by securely storing their fragments in a distributed fashion based on the agreed Erasure Coding (EC) scheme. Using the EC concept, the proposed model prevents data loss due to server failures and software or hardware faults. The proposed system is designed in such a way that it is secure from both insider and outsider adversaries as well as to meets specific design goals such as compromise resilience, brute-force attack resilience, reliability, secure-deduplication, key security, and management. Here we summarize some of the important findings

Table 3
Complexities of various operations.

	Operations	Time	Space	Save space
1	Distinct file uploaded by m users (assume each file of size n)	O(n)	O(mn)	–
2	At TTP, Inside-user deduplication check and file upload			
	(a) If file locator exists in TTP's database	O(1)	O(1)	O(n)
	(b) If file locator does not exists in TTP's database	O(n)	O(1)	–
3	At CSP, cross-user deduplication check and stores the file			
	(a) If file locator exists in CSP's database	O(1)	O(1)	O(n)
	(b) If file locator does not exists in CSP's database	O(n)	O(n)	–
4	Inside-user deduplication check before file deletion			
	(a) If file exists in TTP's database	O(1)	–	–
5	Cross-user deduplication check and deletes the file			
	(a) If file exists in CSP's database & file counter value = 1	O(n)	–	–
	(b) If file exists in CSP's database & file counter value > 1	O(1)	–	–

Table 4
Comparison of proposed approach with existing deduplication schemes.

			[8]	[6]	[18]	[9]	[19]	[17]	Proposed scheme
Deduplication	Secure-deduplication		Y	Y	Y	Y	Y	Y	Y
	Managing Entity		Manager	Server	Client	Client	Client	Client & Server	TTP & CSP
	Inside-user		–	Y	Y	Y	Y	Y	Y
	Cross-user		Y	Y	Y	–	Y	Y	Y
QoS Features	Client Storage Overhead Reduced		Y	N	N	–	N	N	Y
	Save Bandwidth		Y	N	Y	Y	–	Y	Y
	Data Confidenciality		Y	Y	Y	Y	Y	Y	Y
	Compromise Resilience		Y	Y	Y	Y	Y	–	Y
Attack Mitigated	Side Channel	(Outsider/User)	Y	Y	Y	Y	–	Y	Y
		(Insider/CSP)	Y	Y	N	Y	Y	Y	Y
	Brute Force		–	Y	–	–	Y	Y	Y
Key	Security		Y	Y	Y	–	Y	–	Y
	Management		Y	N	Y	N	Y	–	Y
Reliability	Key		N	N	Y	N	Y	N	Y
	Data		N	N	N	N	N	Y	Y
	Scheme		–	–	RSSS	–	RSSS	RSSS	EC

of the research. It is observed that the data encryption and decryption operations execute in constant time (< 0.1 s) up to 4 MB, and then the time increases exponentially with the increase in file size. However, the average execution time for encrypting and decrypting the keys lies between 1 to 6 μs, and between 1 to 4 μs respectively. Besides, the key size is small thus the average key encoding time lies between 7 to 12 μs. Since our experiment uses $RS(3,2)$ encoding for testing purposes, it limits the max space overhead up to 66.66% compared to the N-way replication technique, which generates space overhead of $O(N)$, where N denotes the number of replicas.

In future work, we aim to enhance the proposed framework with verification techniques to provide data integrity and user trust. We believe that any cloud system is trusted and productive if it provides an effective verification methodology. The verification strategy ensures that the data and information stored in the cloud are complete, accurate, consistent, and accessible when needed. Because data is stored in an encoded and encrypted form across cloud datacenters, proper management and storage correctness assurance strategies are required. In addition, we intend to implement a disaster recovery plan with storage correctness assurance. It will automatically trigger the data recovery module if any data inconsistency or corruption is detected. As a result, integrating a trustable storage correctness checking mechanism into the future development of the *dualDup* architecture helps to improve customer satisfaction levels and promote business continuity.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Mell P, Grance T. The NIST definition of cloud computing. 2019, [Accessed 12 May 2019].
[2] Aljahdali H, Albatli A, Garraghan P, Townend P, Lau L, Xu J. Multi-tenancy in cloud computing. In: 2014 IEEE 8th international symposium on service oriented system engineering. 2014, p. 344–51.
[3] Douceur JR, Adya A, Bolosky WJ, Simon P, Theimer M. Reclaiming space from duplicate files in a serverless distributed file system. In: Proceedings 22nd international conference on distributed computing systems. 2002, p. 617–24.
[4] Bellare M, Keelveedhi S, Ristenpart T. Message-locked encryption and secure deduplication. In: Johansson T, Nguyen PQ, editors. Advances in cryptology – EUROCRYPT 2013: 32nd annual international conference on the theory and applications of cryptographic techniques, Athens, Greece, May 26-30, 2013. proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg; 2013, p. 296–312.
[5] Abadi M, Boneh D, Mironov I, Raghunathan A, Segev G. Message-locked encryption for lock-dependent messages. In: Advances in cryptology – CRYPTO 2013: 33rd annual cryptology conference, Santa Barbara, CA, USA, August 18-22, 2013. proceedings, Part I. Berlin, Heidelberg: Springer Berlin Heidelberg; 2013, p. 374–91.
[6] Bellare M, Keelveedhi S, Ristenpart T. DupLESS: Server-aided encryption for deduplicated storage. In: Proceedings of the 22Nd USENIX conference on security. SEC'13, Berkeley, CA, USA: USENIX Association; 2013, p. 179–94.
[7] Storer MW, Greenan K, Long DD, Miller EL. Secure data deduplication. In: Proceedings of the 4th ACM international workshop on storage security and survivability. StorageSS '08, New York, NY, USA: ACM; 2008, p. 1–10.
[8] Puzio P, Molva R, Önen M, Loureiro S. ClouDedup: Secure deduplication with encrypted data for cloud storage. In: 2013 IEEE 5th international conference on cloud computing technology and science, Vol. 1. 2013, p. 363–70.
[9] Kaaniche N, Laurent M. A secure client side deduplication scheme in cloud storage environments. In: 2014 6th International conference on new technologies, mobility and security. 2014, p. 1–7.
[10] Li P, Jin X, Stones RJ, Wang G, Li Z, Liu X, et al. Parallelizing degraded read for erasure coded cloud storage systems using collective communications. In: 2016 IEEE Trustcom/BigDataSE/ISPA. 2016, p. 1272–9.
[11] Lin HY, Tzeng WG. A secure erasure code-based cloud storage system with secure data forwarding. IEEE Trans Parallel Distrib Syst 2012;23(6):995–1003.
[12] Zhang G, Wu G, Wang S, Shu J, Zheng W, Li K. CaCo: An efficient Cauchy coding approach for cloud storage systems. IEEE Trans Comput 2016;65(2):435–47.

[13] Yin J, Tang Y, Deng S, Li Y, Lo W, Dong K, et al. ASSER: An efficient, reliable, and cost-effective storage scheme for object-based cloud storage systems. IEEE Trans Comput 2017;66(8):1326–40.
[14] Nachiappan R, Javadi B, Calheiros RN, Matawie KM. Cloud storage reliability for big data applications: A state of the art survey. J Netw Comput Appl 2017;97:35–47.
[15] Xu F, Wang Y, Ma X. Incremental encoding for erasure-coded cross-datacenters cloud storage. Future Gener Comput Syst 2018;87:527–37.
[16] Hasan M, Goraya MS. Fault tolerance in cloud computing environment: A systematic survey. Comput Ind 2018;99:156–72.
[17] Li J, Chen X, Huang X, Tang S, Xiang Y, Hassan MM, et al. Secure distributed deduplication systems with improved reliability. IEEE Trans Comput 2015;64(12):3569–79.
[18] Li J, Chen X, Li M, Li J, Lee PPC, Lou W. Secure deduplication with efficient and reliable convergent key management. IEEE Trans Parallel Distrib Syst 2014;25(6):1615–25.
[19] Zhou Y, Feng D, Xia W, Fu M, Huang F, Zhang Y, et al. SecDep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In: 2015 31st Symposium on mass storage systems and technologies. 2015, p. 1–14.
[20] De Santis A, Masucci B. Multiple ramp schemes. IEEE Trans Inform Theory 1999;45(5):1720–8.
[21] Meyer DT, Bolosky WJ. A study of practical deduplication. ACM Trans Storage (TOS) 2012;7(4):14:1–20.
[22] Harnik D, Pinkas B, Shulman-Peleg A. Side channels in cloud services: Deduplication in cloud storage. IEEE Secur Priv 2010;8(6):40–7.
[23] Meister D, Brinkmann A. Multi-level comparison of data deduplication in a backup scenario. In: Proceedings of SYSTOR 2009: the israeli experimental systems conference. SYSTOR '09, New York, NY, USA: ACM; 2009, p. 8:1–8:12.
[24] Qin C, Li J, Lee PPC. The design and implementation of a rekeying-aware encrypted deduplication storage system. ACM Trans Storage 2017;13(1):9:1–30.
[25] Bellare M, Boldyreva A, O'Neill A. Deterministic and efficiently searchable encryption. In: Menezes A, editor. Proceedings of the 27th annual international cryptology conference on advances in cryptology. CRYPTO'07, Berlin, Heidelberg: Springer-Verlag; 2007, p. 535–52.
[26] Rogaway P, Shrimpton T. A provable-security treatment of the key-wrap problem. In: Advances in cryptology - EUROCRYPT 2006: 24th annual international conference on the theory and applications of cryptographic techniques, St. Petersburg, Russia, May 28 - June 1, 2006. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg; 2006, p. 373–90.
[27] Naor M, Reingold O. Number-theoretic constructions of efficient pseudo-random functions. J ACM 2004;51(2):231–62.
[28] Weatherspoon H, Kubiatowicz J. Erasure coding Vs. Replication: A quantitative comparison. In: Revised papers from the first international workshop on peer-to-peer systems. IPTPS '01, London, UK, UK: Springer-Verlag; 2002, p. 328–38.
[29] Yan Z, Ding W, Zhu H. A scheme to manage encrypted data storage with deduplication in cloud. In: International conference on algorithms and architectures for parallel processing. Springer; 2015, p. 547–61.
[30] Miao M, Wang J, Li H, Chen X. Secure multi-server-aided data deduplication in cloud computing. Pervasive Mob Comput 2015;24:129–37.
[31] Fan C-I, Huang S-Y, Hsu W-C. Encrypted data deduplication in cloud storage. In: 2015 10th Asia joint conference on information security. IEEE; 2015, p. 18–25.
[32] Van Dijk M, Gentry C, Halevi S, Vaikuntanathan V. Fully homomorphic encryption over the integers. In: Proceedings of the 29th Annual international conference on theory and applications of cryptographic techniques. EUROCRYPT'10, Berlin, Heidelberg: Springer-Verlag; 2010, p. 24–43.
[33] J. Dworkin M, B. Barker E, R. Nechvatal J, Foti J, E. Bassham L, Roback E, et al. National institute of standards, and technology. FIPS PUB 197: Advanced encryption standard. 2001, URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
[34] Biryukov A, Khovratovich D. Related-key cryptanalysis of the full AES-192 and AES-256. In: Proceedings of the 15th international conference on the theory and application of cryptology and information security: advances in cryptology. ASIACRYPT '09, Berlin, Heidelberg: Springer-Verlag; 2009, p. 1–18.
[35] H. Dang Q. National institute of standards, and technology. FIPS PUB 180-4: secure hash standard. 2015, URL https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.