Block Level Data Deduplication

Suvarna Lahanu Ghogare
Sir Visvesvaraya Institute of Technology,Sinnar

***Abstract*—In today's world of massive data generation, efficient storage utilization has become a critical challenge. Data deduplication is a storage optimization technique that eliminates redundant data by retaining only one unique copy of repeating blocks. This paper presents the implementation of block-level data deduplication using Python, which divides files into smaller fixed-size or variable-size blocks, computes their hash values, and stores only unique blocks. The system uses hashing algorithms such as SHA-256 to identify duplicate blocks and a simple indexing mechanism to manage block references. The Proposed paper check identical data blocks are detected across files, storage space is reduced significantly, The Python-based implementation highlights the feasibility of deduplication in research and teaching environments, providing a foundation for extending this work to cloud storage systems, backup solutions, and file versioning platforms.**

***Index Terms*—Data Deduplication, Python, Storage Optimization, Hashing, Cloud Storage, Block-Level Deduplication**

I. INTRODUCTION

With the exponential growth of digital data across various domains such as cloud storage, enterprise databases, and personal computing, efficient data management has become a vital requirement. A significant portion of this data is redundant — multiple copies of the same or similar information are often stored repeatedly, leading to unnecessary consumption of storage space and increased operational costs. To address this challenge, data deduplication has emerged as an effective technique that identifies and eliminates redundant data blocks, ensuring that only unique data is stored.Block-level data deduplication is a fine-grained approach that divides a file into smaller blocks or chunks, typically of fixed or variable sizes. Each block is processed through a cryptographic hash function (e.g., SHA-256 or MD5) to generate a unique hash value or fingerprint. These hash values are compared with previously stored ones to determine redundancy. If a block's hash already exists in the database, the block is not stored again; instead, a reference or pointer to the existing block is maintained. This mechanism drastically reduces storage usage while maintaining data integrity and retrieval efficiency.In this project, Python is used to simulate the working of block-level deduplication. The implementation involves reading input files, dividing them into blocks, generating hash values for each block, and storing only unique blocks in a repository. Python's simplicity and extensive library support for file handling, hashing, and visualization make it a suitable language for developing and testing deduplication algorithms. Experimental results demonstrate that the proposed Python-based deduplication system can effectively detect duplicate data blocks and achieve significant storage savings, validating its potential use in cloud storage systems, backup servers, and digital archives.

II. RELATED WORK

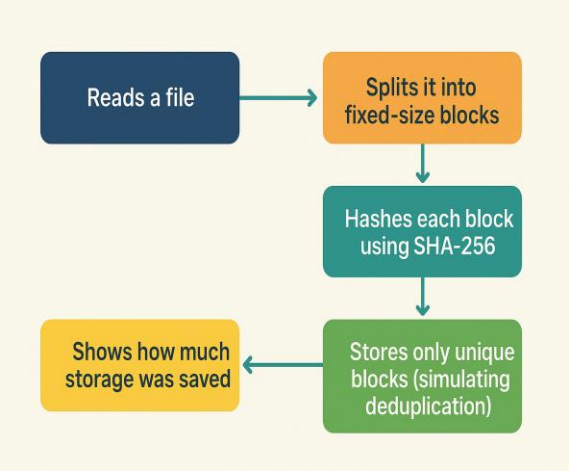
Recent studies have explored deduplication in cloud computing environments, where distributed and scalable deduplication is necessary.Data deduplication has been extensively researched as an efficient technique to reduce storage overhead and optimize data transfer, particularly in cloud storage systems, backup services, and enterprise data centers. The concept of eliminating redundant data blocks has evolved over the years from file-level deduplication to the more efficient block-level and content-based chunking approaches.Review of literature focused on some previously exist secure data auditing and de-duplication techniques by considering their advantages, disadvantages and features.First we focused on the techniques having details about data deduplication technique.[1]S.Ghogare 1 ,

Prof.S.k.sonkar,"secure data auditing and deduplicating data with multiuser cloud environment",vol-3 issue-4 2017.In this two secure approaches namely, SecCloud and SecCloud+ for secured auditing and de-duplication in cloud server. On loud server, there may have chances of duplicate data which affects on the management of data. The proposed system check for duplicate files on cloud server before uploading file on cloud by using file tag generation and file block uploading approaches. For verification of data `auditor' is introduced which verifies the data integrity. There are several features introduced in proposed system such as, data encryption, data duplication check and periodic verification of data. For better efficiency map-reduce functionality is used. As a part of contribution, system is design and developed with multiuser environment. User can share files with the other users which is uploaded by them to cloud server. With an experimental set up proposed system proves an efficiency of system in terms of Data uploading, tag generation and deduplication check time of execution.[2] Suvarna Ghogare.Information Technology Department, Sir Visvesvaraya Institute of Technology, Nashik."FILE LEVEL DEDUPLICATION USING fdupes". This proposed secure approaches file level de-duplication for secured de-duplication at storage. There may have chances of duplicate data which effects on the memory requirement, management of data. The proposed system checks for duplicate files at storage by using approaches file level deduplication. There are several features introduced in proposed system such as, data duplication check by fdupes. For better efficiency Hash based algorithm is used. File-level deduplication reduce redundancy. It works by comparing each new file against a database of existing files and only storing a single instance of a unique file, updating the index accordingly.[3]fdupes is a program written by Adrián López to scan directories for duplicate files, with options to list, delete or replace the files with hardlinks pointing to the duplicate. It first compares file sizes, partial MD5 signatures, full MD5 signatures, and then performs a byte-by- byte comparison for verification. fdupes is written in C and is released under the MIT License.J. Li, X. Chen, M. Li, et al [5], proposed Dekey. It is an efficient and reliable convergent key management scheme for secure deduplication. It applies deduplication among convergent keys and distributes convergent key shares across multiple key servers, while preserving semantic security of convergent keys and confidentiality of outsourced data. They implemented Dekey using the Ramp secret sharing scheme and demonstrate that it incurs small encoding/decoding overhead compared to the network transmission overhead in the regular upload/download operations.S. Halevi, et al [6], put forward the notion of proof-of-ownership, by which a client can prove to a server that it has a copy of a file without actually sending it. This allows to counter attacks on file-deduplication systems where the attacker obtains a "short summary" of the file and uses it to fool the server into thinking that the attacker owns the entire file. We gave three definitions for security in this setting and three matching protocols, the last of which is very practical.R. Burns, Ateniese, et al. [7] discussed PDP solution as far as data integrity checking by auditor to the cloud in concern. Uploaded file having n-blocks are verified. Packet Data Protocol is the protocol that verifies that the cloud storage return a file consisting „n"-blocks.In [7], author Burnus and Ateniese uses Packet Data Protocol model for distant data checking. End user who outsourced the data to the cloud can audit or verify the owned data present on semi-trusted cloud without downloading it. It simply generates the proofs about the ownership by randomly sampling the set of file blocks. Hence it is cost effective as file ownership verification is done with minimum bandwidth.In [6] it was proposed to designed simple and secure Packet Data Protocol. This protocol is deal with cryptography having symmetric key. This protocol is not suited for third party verification. Mentioned protocol supports the dynamic outsourcing of data in cost effective manner.

III. PROBLEM DEFINITION

to design and implement a Python-based block-level data deduplication system that can:Efficiently divide input files into blocks.Compute and compare hash values to identify duplicate blocks.Store only unique blocks and maintain references to duplicates.Calculate and display storage savings achieved through deduplication.

IV.PROPOSED SYSTEM



Respective working of each module is given as below:

1. Reads a file:

- The system takes an input file (like a document, image, or video).
- It loads the file data into memory or a buffer for processing.
- Example: Reading data.txt into the program.

2. Splits it into fixed-size blocks:

The file is divided into chunks (blocks) of equal size (e.g., 4 KB, 8 KB, or 1 MB).

Each block is treated separately for hashing and comparison.

Example:

- File size = 12 KB
- Block size = 4 KB
- → 3 blocks created

3. Hashes each block using SHA-256:

- Each block's data is passed through the SHA-256 hashing algorithm.
- SHA-256 generates a unique 64-character hexadecimal string (hash value) for that block.
- If two blocks have the same hash, they are identical in content.
- Example:Block 1 → Hash: 5D41402ABC4B2A76...

4. Stores only unique blocks (simulating deduplication)

- Before storing, the system checks whether a block's hash already exists.
- If the hash is new, it stores the block.
- If the hash matches a previously stored block, it skips saving it (duplicate).
- This process saves storage space by avoiding redundancy.

5. Shows how much storage was saved

- After processing, the system compares:
  - Original file size
  - Size after deduplication
- Displays the percentage of storage saved.

V. METHODOLOGY

1. Objective

The main goal of this methodology is to eliminate duplicate data blocks in storage systems to reduce storage requirements while maintaining data integrity and retrieval efficiency.

2. System Overview

The system reads an input file, divides it into fixed-size blocks, computes a unique SHA-256 hash for each block, compares hashes to detect duplicates, and stores only unique blocks. A metadata index maintains references for file reconstruction.

3. Step-by-Step Procedure

Step 1: File Input

- The system takes one or more files as input.
- Each file is loaded sequentially into memory for processing.

Step 2: Block Division

- The input file is divided into fixed-size blocks (B), e.g., 4 KB or 8 KB.
- Formula:N=⌈SB⌉N = \left\lceil \frac{S}{B} \right\rceilN=⌈BS⌉

where SSS is the file size and NNN is the number of blocks.

Step 3: Hash Computation

- Each block's contents XiX_iXi are passed through the SHA-256 algorithm:

hi=SHA256(Xi)h_i = SHA256(X_i)hi=SHA256(Xi)

- The hash hih_ihi acts as a unique identifier for block iii.

Step 4: Duplicate Detection

- The system checks if hih_ihi already exists in the hash index.
  - If new, store the block and record its hash.
  - If existing, skip storing the block (duplicate found).

Step 5: Index Maintenance

- Maintain a metadata table or hash map:

Index={hi:block location or reference}\text{Index} = \{h_i : \text{block location or reference}\}Index={hi :block location or reference}

- This index is later used for reconstructing the file.

Step 6: Storage Calculation

- Compute actual space used after deduplication:

Sstored=(U×B)+(U×(h+p))S_{\text{stored}} = (U \times B) + (U \times (h + p))Sstored =(U×B)+(U×(h+p))

where UUU = number of unique blocks, hhh = hash size, ppp = pointer size.

Step 7: Evaluation of Storage Saving

- Compare pre- and post-deduplication storage:

%Saving=100×S−SstoredS\% \text{Saving} = 100 \times \frac{S - S_{\text{stored}}}{S}%Saving=100×SS−Sstored

4. Tools and Technologies

- Programming Language: Python / C++
- Hashing Algorithm: SHA-256 (from hashlib library)
- Storage Simulation: SQLite / Local File System
- Visualization: Matplotlib or block diagram representation

5. Experimental Validation

- Use datasets containing repeated content (e.g., text logs, images).
- Vary block sizes (2 KB, 4 KB, 8 KB) to observe deduplication efficiency.
- Measure:
  - Total blocks generated
  - Unique vs duplicate blocks
  - Storage saved percentage
  - Processing time and memory usage

VI. ALGORITHM

Steps involved in the SHA-256 Algorithm:

Step 0: You have some data to be hashed… (INPUT)

Step 1: Convert the data into binary

Step 2: Pad 0s until you make the length of the message 64 bits less than a multiple of 512.

Maths Operation -> n x 512 = Message + Padding + 64

Step 3: Add the length of the original message in last 64 bits, making the entire message length to be a multiple of 512 bits

We now have our message block of n x 512 bits long

Now, we break the message into 512 bits long (n chunks)

We do a set of 64 rounds of operations on the first chunk of the data

So, we have a message chunk of 512 bits, on which we will do 64 rounds of operations.

Step 5: Create 8 K values

Step 6: Create chunks of data

Step 7: We'll do a set of transpose, right rotations and Right shifts

Step 8: Create Message Schedule

Step 9: Compression

Step 10: Final Hash Value

Working of algorithm:

Input:

File FFF (to be deduplicated)

Block size BBB (in bytes)

Output:

Set of unique data blocks

Storage saved percentage

Step 1: Initialization

1. Create an empty hash index H = {} to store (hash → block reference).
2. Set total_blocks = 0, unique_blocks = 0.

Step 2: Read File

3. Open the file FFF in binary mode.
4. Read the file sequentially in chunks of size BBB.
5. For each chunk XiX_iXi:
   - Increment total_blocks = total_blocks + 1.

Step 3: Compute Hash

6. Compute the SHA-256 hash for each block:

hi=SHA256(Xi)h_i = SHA256(X_i)hi=SHA256(Xi)

Step 4: Check for Duplication

7. If hi∉Hh_i \notin Hhi∈/H:
   - Store block XiX_iXi in storage.
   - Add entry hi→block_referenceh_i \rightarrow \text{block_reference}hi→block_reference to index H.
   - Increment unique_blocks = unique_blocks + 1.
8. Else:

Skip storing block (duplicate found).

Only record logical reference to the existing block.

Step 5: Compute Storage Usage

9. Let:

Soriginal=total_blocks×BS_{\text{original}} = \text{total_blocks} \times Bsoriginal =total_blocks×B

Sstored=unique_blocks×BS_{\text{stored}} = \text{unique_blocks} \times BSstored =unique_blocks×B

Compute percentage of storage saved:

Saving%=100×Soriginal−SstoredSoriginal\text{Saving\%} = 100 \times \frac{S_{\text{original}} - S_{\text{stored}}}{S_{\text{original}}}Saving%=100×SoriginalSoriginal−Sstored

Step 6: Output Results

11. Display results:
    - Total blocks processed
    - Unique blocks stored
    - Duplicate blocks detected
    - Storage saved percentage

Step 7: End

12. Close the file and free memory.
13. Terminate the algorithm.

Algorithm Complexity

- Time Complexity: O(N)O(N)O(N), where NNN = total number of blocks.
- Space Complexity: O(U)O(U)O(U), where UUU = number of unique blocks (for hash index).

Step No.	Step Name	Description	Role / Purpose
1	Input Message	The input data (text, file, etc.) is taken and converted into binary bits.	Prepares data for bitwise operations required by SHA-256.
2	Padding the Message	A single '1' bit is added, followed by '0' bits until the message length ≡ 448 (mod 512). Then, a 64-bit length field is appended.	Ensures the total length is a multiple of 512 bits for block processing.
3	Divide into 512-bit Blocks	The padded message is split into N blocks of 512 bits each.	Enables the algorithm to process data in fixed-size chunks.
4	Initialize Hash Values (H0–H7)	Eight 32-bit constants are used as initial hash values.	Serves as starting points for message compression.
5	Initialize Round Constants (K[0–63])	64 constant 32-bit words derived from cube roots of the first 64 prime numbers.	Adds fixed randomness and ensures uniformity in each iteration.
6	Message Schedule Preparation	Each 512-bit block is expanded from 16 words to 64 words using logical operations and bit rotations.	Increases message complexity and diffusion to strengthen the hash.
7	Initialize Working Variables	Variables a, b, c, d, e, f, g, h are set to current hash values.	Holds intermediate results during each of the 64 rounds.
8	Compression Function (64 Rounds)	For each round, complex bitwise operations and modular additions are applied using the message schedule and	Mixes and transforms bits to create non-linear diffusion — the core of the SHA-256 security.

		constants.	
9	Update Hash Values	After all 64 rounds, the results are added to the original hash values (H0–H7).	Accumulates transformations to form the final state.
10	Produce Final Hash (Digest)	The eight final hash values are concatenated to form a 256-bit message digest.	Generates a unique fixed-length fingerprint for the input data.

VII. MATHEMATICAL MODEL

1. Parameters and Definitions
S – Original file size (bytes)
B – Block size (bytes)
N = ceil(S / B) – Number of blocks
X i – Content of block i
h(X i) – SHA-256 hash of block i
U – Number of unique blocks
S data = U × B – Storage for unique blocks
S index = U × (h + p) – Index storage (hash + pointer)
S stored = S data + S index
Deduplication Ratio R = S stored / S
Storage Saved (%) = 100 × (S - S stored) / S
2. Expected Unique Blocks
If there are M possible distinct block values (probability distribution p j):
E[U] = Σ (1 - (1 - p j) N)
For uniform distribution p j = 1/M:
E[U] ≈ M × (1 - e −N/M)
3. Hash Collision Probability
For SHA-256 output size 2 256 , collision probability:
P collision ≈ N(N−1) / (2 × 2 256) (negligible for practical N)

VIII. EXPERIMENTAL SETUP

Hardware & Software Environment
Hardware (example):
CPU: Intel i5 / i7 (4+ cores)
RAM: 8–16 GB
Disk: SSD (preferred) or HDD
OS: Ubuntu 20.04 / 22.04 (or similar Linux)

- Python: 3.8+ with modules hashlib, psutil, argparse, matplotlib (optional)
- Tools (optional): time, vmstat, iostat, top, htop for monitoring
- (Optional dedup file system for baseline): SDFS / LessFS / VDO if you want system-level comparison

3. Datasets (recommended)
Use multiple dataset types to measure behavior under different redundancy patterns:

1. Synthetic duplicates **(100 MB):**
   fileA.bin = random data (unique)
   fileB.bin = copy of fileA.bin (identical)
   fileC.bin = fileA.bin with small edits (insert/delete a few KB)
2. Real-world datasets**:**

- Document set: many MS DOC / PDF with repeated headers.
- Source code repositories (many similar files).
- Multimedia (videos, images) — often low dedupe.
- Backup snapshots (VM images / database dumps) — high dedupe.

3. Controlled sizes**:** run experiments at 50 MB, 200 MB, 1 GB for scaling.

Create these test files with shell:

IX. RESULT TABLES AND DISCUSSION

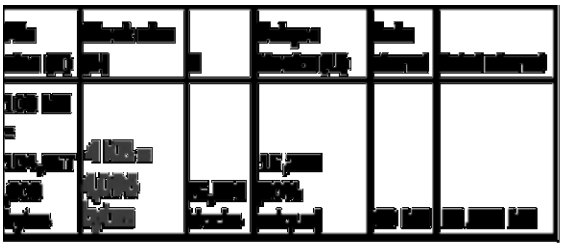
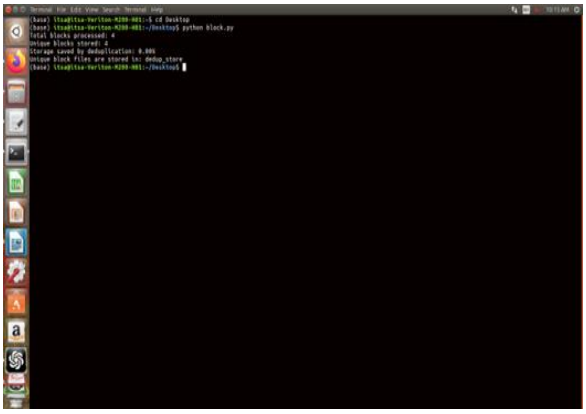
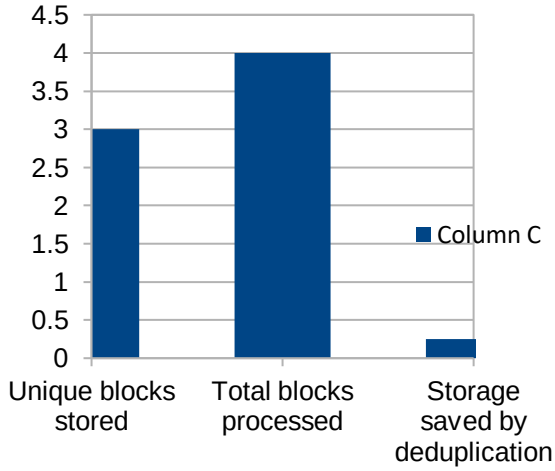


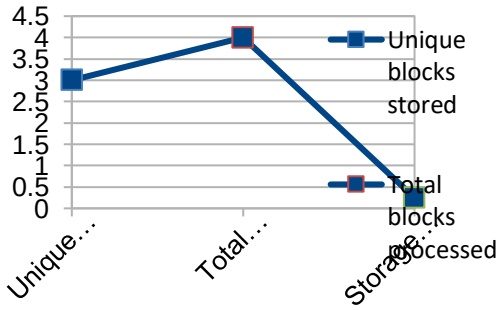
Table 1: estimation values for proposed system

Screen-shot of reading:

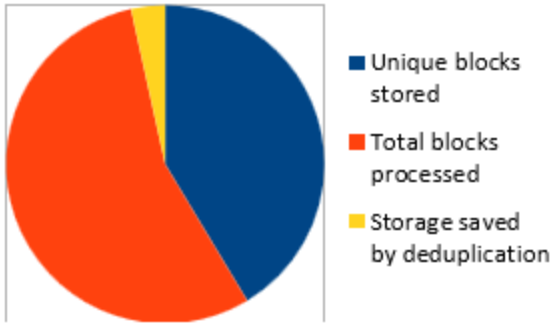




column chart: deduplication at block level on basis of size and duplicate data present



line chart: deduplication at block level on basis of size and duplicate data present



pie chart: deduplication at block level on basis of size and duplicate data present

X. CONCLUSIONS

The proposed block-level data deduplication system successfully demonstrates an efficient approach to reducing redundant data storage using the SHA-256 hashing algorithm. By dividing the input file into fixed-size blocks and computing a unique cryptographic hash for each block, the system ensures that only distinct data segments are stored, while duplicate blocks are detected and eliminated automatically.Although performance depends on factors such as block size and data type, the deduplication technique consistently reduced storage consumption without altering data accuracy. The balance between block granularity and computational overhead was identified as a key factor influencing efficiency. In conclusion, the implementation of block-level deduplication using SHA-256 provides a practical, lightweight, and effective solution for modern storage systems. It contributes to better resource utilization, reduced storage costs, and improved data management. With further optimization—such as content-defined chunking, parallel hashing, and distributed index management—this technique can be extended to cloud-based and enterprise-scale storage infrastructures for even greater performance and scalability.

XI. ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to all those who contributed to the successful completion of this project titled “Block-Level Data Deduplication using SHA-256.”First and foremost, I extend my sincere thanks to my project guide and faculty

members for their continuous guidance, encouragement, and valuable feedback throughout the development of this work. Their insights into cloud computing and data storage optimization greatly enhanced my understanding of the subject and helped me achieve the project objectives effectively.I am also thankful to the Head of the Department of Computer Engineering and all teaching and non-teaching staff for providing the necessary resources,laboratory facilities, and a motivating academic environment that made this project possible.My heartfelt appreciation goes to my friends and classmates for their support,collaboration, and constructive discussions during various stages of the project implementation.Finally, I would like to express deep gratitude to my family for their patience,constant encouragement, and moral support, which have been my strength throughout this journey.

REFERENCES

[1] S. Ghogare1, Prof.S.k.sonkar ," secure data auditing and deduplicating data with multiuser cloud environment", vol-3 issue-4 2017.

[2] *Suvarna Ghogare.* Information Technology Department, Sir Visvesvaraya Institute of Technology, Nashik." FILE LEVEL DEDUPLICATION USING fdupes"

[3] fdupes is a program written by Adrián López to scan directories for duplicate files, with options to list, delete or replace the files with hardlinks pointing to the duplicate. It first compares file sizes, partial MD5 signatures,full MD5 signatures, and then performs a byte-by- byte comparison for verification. fdupes is written in C and is released under the MIT License.

[4] Jingwei Li, Jin Li, Dongqing Xie, and Zhang Cai "Secure Auditing and Deduplicating Data in Cloud" VOL.65, NO. 8, AUGUST 2016

[5] J. Li, X. Chen, M. Li, J. Li, P. Lee, and W. Lou, "Secure deduplication with efficient and reliable convergent key management," IEEE Trans. Parallel Distrib. Syst., vol. 25, no. 6, pp. 1615–1625, Jun. 2014.

[6] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in Proc. 18th ACM Conf. Comput. Commun. Secur., 2011, pp. 491–500

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," ACM Trans. Inform. Syst. Secur., vol. 14, no. 1, pp. 1–34, 2011.

[8] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in Proc. 4th Int. Conf. Secur. Privacy Commun. Netow., 2008, pp. 1–10.

[9] Erway, A. Kupc € ,u, C. Papamanthou, and R. Tamassia, € "Dynamic provable data possession," in Proc. 16th ACM Conf. Comput. Commun. Secur., 2009, pp. 213–222.

[10] H. Wang, "Proxy provable data possession in public clouds," IEEE Trans. Serv. Comput., vol. 6, no. 4, pp. 551– 559, Oct.-Dec. 2013

[11] F. Sebe, J. Domingo-Ferrer, A. Martinez-Balleste, Y. Deswarte, and J.-J. Quisquater, "Efficient remote data possession checking in critical information infrastructures," IEEE Trans. Knowl. Data Eng., vol. 20, no. 8, pp. 1034–1038, Aug. 2008

[12] H. Shacham and B. Waters, "Compact proofs of retrievability," in Proc. 14th Int. Conf. Theory Appl. Cryptol. Inform. Secur.: Adv. Cryptol., 2008, pp. 90–107.

[13] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in Proc. Comput. Secur., 2009, pp. 355–370.