A Project Report

On

Randomized Encryption Mechanism for Deduplicated Cloud Storage Systems

BY

PRITHVI HEGDE

2020A7PS1718H

Under the supervision of

JAY KAMLESH DAVE

SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS OF

CS F366 (LABORATORY PROJECT)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(MAY 2023)

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to Dr Jay Dave, for his valuable guidance and mentorship during the design and development of this project. His insights, expertise, and patience helped me navigate through the various stages, challenges and complexities of this project, until completion.

I would like to extend my special thanks to the director, Dr G Sundar, and the IC of the course, Dr Jabez Christopher, for providing me with the resources, facilities and the opportunity to carry out this project and for allowing me to pursue this research topic.

I would also like to thank Dr Jay Dave for taking the time to mentor me and providing me with constructive feedback throughout the project. His expertise and guidance have been crucial in helping me refine my research methodology and develop my analytical skills.

I am truly honoured to have had the opportunity to work under his mentorship, and I am grateful for the knowledge and experience gained during this project. Thank you once again for the continued support and encouragement.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

Certificate

This is to certify that the project report entitled "**Randomized Encryption Mechanism for Deduplicated Cloud Storage Systems**" submitted by Mr. PRITHVI HEGDE (ID No. 2020A7PS1718H) in partial fulfilment of the requirements of the course CS F366 (LABORATORY PROJECT), embodies the work done by him under my supervision and guidance.

Date: 11/05/2023 **(JAY KAMLESH DAVE)**

BITS- Pilani, Hyderabad Campus

ABSTRACT

The practice of encrypting data on the client-side before uploading it to a cloud storage service is an important step in protecting user privacy, and cannot be avoided. However, most of these encryption methods typically use randomized encryption techniques, which conflict with standard deduplication practices for reducing storage and communication costs, as the ciphertext generated by the same file uploaded multiple times will be unique each time.

Researchers are actively working to reconcile these two approaches, and the most popular method is called Convergent Encryption. However, this too has it's shortcomings, as convergent encryption uses deterministic encryption methods, and is thus vulnerable to offline brute force attacks.

With RESIST, we have proposed a novel deduplication scheme that supports client-side randomised encryption without requiring additional independent servers. We attempt to perform a proof-of-concept implementation of the scheme in a realistic scenario demonstrating its efficiency and effectiveness.

CONTENTS

1. Introduction

The International Data Corporation (IDC) predicts that by 2025, there will be 181 ZB of data worldwide. Much of this data is stored on large cloud servers. Protecting these storage servers against malicious attacks, while ensuring speed and efficiency of data retrieval is a crucial problem that many researchers are working on. Deduplication is a technique employed by cloud storage providers to identify and remove redundant data on a cloud server. This method can significantly reduce storage costs by eliminating the need to store multiple copies of the same data. Cloud storage providers typically implement deduplication using either inline or post-process methods.

From the point of view of when deduplication occurs, there are two kinds of deduplication:

1. Inline deduplication eliminates duplicate data as it is ingested into the storage system. This is also called source-based deduplication, ie, the user must send tags, which help identify the ciphertext/ verify ownership/ verify deduplication. Our paper is a source-based implementation.
2. Post-process deduplication identifies and removes duplicates after the data has been stored. The choice of method usually depends on the specific use case and storage requirements of the cloud storage provider. This is also called target-based deduplication.

Similarly, based on how the granularity of data, deduplication can be classified into two kinds:

1. File-level: we ensure that only one copy of a redundant file is stored, i.e. deduplication is done at the file level
2. Block-level: Files are divided into blocks or chunks. These blocks are then checked for redundancy. This is more granular than file-level deduplication and is more extensive. There can be many ways of deciding block size, number of blocks, etc.

Client-side encryption is at odds with deduplication, as encryption on the client side leads to the creation of unique ciphertexts, which will not allow for deduplication. Even if the underlying files are identical, they will have different ciphers, and due to this, are seen as different files from the perspective of deduplication. One of the few ways of combining client-side encryption with data deduplication is Convergent Encryption. Convergent encryption is used in cloud storage systems to enable efficient deduplication. With convergent encryption, data is encrypted using a unique encryption key that is derived from the content of the data itself, rather than a randomly generated key. This means that identical pieces of data will always produce the same encryption key, regardless of who encrypted the data.

The major problem with convergent encryption though, is that it is susceptible to offline brute-force attacks.

- Offline brute-force attack: If we use a deterministic (nonrandom) encryption mechanism, as is generally used in convergent encryption methods, the adversary can simply try all possible ways to reconstruct the files offline, and then target the deduplication storage server.

2. Related work

Bellare et al. [3]: In this paper, a primitive known as message-locked encryption (short form: MLE) was defined. MLE has been used successfully by various systems. Douceur et al. [1] proposed convergent encryption (CE) to deal with the problem of deduplication of encrypted data. An important feature of CE is that it allows for encrypting deduplicated data easily. However, both MLE and CE are still vulnerable to bruteforce attacks. This led to a focus on creating a system resistant to such attacks, and Bellare et al. [10] formulated a scheme called DepLESS. DepLESS stores a system secret locally, in a fully trusted key-server. It is from this server through which any host/user can obtain keys. However, such a key server which is fully trusted, and has no risk of exposure/attacks etc exists only in theory. If such a server is actually compromised, it can be made vulnerable to bruteforce attacks. This problem, referred to as "single point of failure" problem, was dealt with in two ways:

- Schemes such as [13,14] which require a single server, similar to DepLESS, but require a certain number of hosts to be online for the scheme to work
- Zhang et al. [11]: requires multiple servers, which are "semi-trusted", and share the system secret

A problem with data deduplication is the problem of ownership, ie. finding the data owner without requiring the owner to provide ciphertexts. Some papers which combat this proof of ownership problem are:

- Halevi et al. [18]: A Merkle Tree was used to determine the owner of the data. But, this methodology requires a fully-trusted server too
- Hur et al. [20]: This paper proposes a methodology with ownership being determined dynamically. However, it has a lot of overhead, as it must recalculate tags, and the encryption of the cipher, to prevent a previous owner from decrypting the text.
- Similar to the previous paper, Kan et al. [24]: proposed a re-encryption-based scheme, which uses achieve dynamic ownership via an identity-based proxy.

Li et al. [21]: Designed a deduplication based storage system called REED. REED allows for fast rekeying and is based on a nonrandom version of AONT. However, a problem with REED is that it is susceptible to DNS attacks. [22, 23] build upon REED, and deal with the problem of DNS attacks.

Some other methods of deduplication are:

- Stanek et al. [2], which introduced "data popularity". Data popularity is determined by the number of users/hosts that have copies of the data, and more popular data is considered as not requiring the same level of security as unpopular data. This was further expanded upon in Stanek et al. [16], which Proposed a threshold scheme. It encrypts unpopular data twice, which supports the deduplication server in checking the correctness of the ciphertext of the data.
- Ha et al. [17], which uses an encryption based proof of ownership, to prevent attacks on data popularity.

3. Proposed work

We are pleased to present the following contributions to address the aforementioned issues:

- Firstly, we propose a single-server cross-user deduplication scheme. Our scheme offers client-side secure encryption, which helps significantly improve data integrity.
- Secondly, our scheme effectively blocks both online and offline brute-force attacks by compromised clients or servers, without relying on an identity server.
- Lastly, we have implemented the suggested approach practically and thoroughly evaluated its effectiveness. Our findings confirm that the proposed approach is highly efficient and offers robust protection against data theft and other security threats.

The implementation of RESIST uses the following classes and algorithms:

Classes:

1. **Merkle Tree**: This class implements the Merkle Tree data structure, a sha-256 hash-based binary tree. Some important subfunctions in this class are:
    a. *__buildTreeRec(self, values:* list[str]): To build the tree recursively
    b. *getLeaves(self) -> list[Node]:* returns the list of all leaf nodes of the tree
    c. *findPath(self, root: Node, x: Node, arr: list = []):* Finds the path from the root node to a given Node object x, and returns True if the path is found. The path is stored in a list arr.
    d. *siblingPath(self, path: list[Node]):* Finds the sibling path of a given path, and is used in conjunction with the findPath function. The sibling path is a list of sibling Node objects for each node in the path, starting from the root and ending with a leaf node.
    e. *findNode(self, node: Node, val: str):* Finds a Node object with a given value val in the Merkle Tree, starting from the given Node object node. If the Node object is found, it is returned, else None.
2. **Node**: Simple recursive node class, for the Merkle Tree. The Node class has a constructor that initialises the instance variables. It also has a static method hashFile that takes a string as input, hashes it using the SHA256 algorithm, and returns the hash value as a string. This method is used to calculate the hash value of the content of a node.

Algorithms:

1. **EncryptKey**: This function takes a securely generated random key, and a file, and encrypts the file using the key. It does this by creating a Merkle Tree from the file, and randomly selects few of the leaf nodes. For each selected leaf, we compute the sibling path of the leaf node in the Merkle Tree. Then the XOR of the sibling values is computed and XORd with the random key, to get CKey. This is finally returned.
2. **LearnKey**: The learnKey function first reads the file and constructs a Merkle tree from its blocks. It then loops over the CKeys list, and for each encrypted key, it computes the XOR of the Merkle tree nodes corresponding to the indices given in the corresponding sub-list of Ls. This gives a new key that is used to decrypt the file. After decrypting the file, the function computes its file tag and compares it with the expected file tag from the FTags list.

If they match, the function returns the key that was used to decrypt the file. Otherwise, it continues with the next encrypted key in the list.

3. **FileDownload**: If any owner requests for the file he/she uploaded, we use the DTag, CFile, CKey, {L} values to extract and decrypt the original file and then this file is downloaded to the local storage of the user.

Implementation Procedure:

Initially, we establish a connection with the database, by means of the firebase python API. The first step of the RESIST algorithm is to check of presence of a file on the server. This is done by the CheckifDTagisPresent function.

First Upload is the sequence that takes place when the file is not present in the storage server. We generate a DTag by encrypting the file using SHA256 and generating a 254-bit key. We then generate CKey and CFile using the EncryptKey algorithm described above. The CFile, along with DTag, FTag, CKey and {L} is then uploaded to the Firebase storage server.

Subsequent upload is the sequence that takes place when the file is present in the storage server. The CheckifDTagisPresent function returns (CKeys, FTags, Ls, clients, R). We call the LearnKey function to check if the CFile matches the corresponding CFile value, and we add the new user as an owner of the file.

However, in subsequent uploads, if the file was modified in any way, we have to upload a new file. This is done in a manner similar to the first upload procedure.

4. Performance analysis

Hardware(s) Used:

Client Device: HP Omen16 Laptop

CPU Details: 11th Gen Intel(R) i7-11800H @2.30 GHz

RAM: 16 GB

Software Details:

Client-Side Code Implementation: Python 3.11 64-bit

Server-Side Code Implementation: Firebase, Firestore Database

Libraries Used: PyCryptodome, pyrebase4, firebase_admin (API to connect to firebase), random, secrets, hashlib

After the complete implementation of the algorithm as shown above, we were able to obtain results, that validated our algorithm successfully.

For a sample file of size 15MB, here is the time taken by various functions in the algorithm:

First Upload:

VerifyDedupTime: 0.0s

File Encryption Time: 0.2864654064178467s

Key Encryption Time: 2.045449733734131s

File uploaded successfully

Total time 9.248428106307983s

Subsequent Upload:

VerifyDedupTime: 0.7681612968444824s

LearnKey Time: 1.5956411361694336s

Total Time: 2.87326979637146s

In both first and subsequent upload, Total Time is dependent on bandwidth while uploading/downloading our file. Among the various algorithms, Verifying Deduplication, i.e. the VerifyDedup() algorithm is dependent on bandwidth, as it has to access the server to fetch records. All other algorithms are not dependent on bandwidth.

We notice that the key encryption algorithm takes the most time. This is seen to be consistent with the implementation of the algorithm, as key encryption involves the creation of a Merkle Tree, which may be time-consuming, depending on the size of the file.

Further, when the bandwidth usage of RESIST is examined, we see that, for a 15MB file, the sizes of:

DTag:	254 bits
FTag:	256 bits
CFile:	15361 KB
CKey:	256 bits
{L}:	1107 B.

CFile, the encrypted file, is the largest among the various files and tags we are uploading to the server and is seen to be approximately equal to the size of the file we are uploading (15MB). This is consistent with the SHA-256 encryption algorithm used to encrypt the file. We can also note that CKey and FTag are both 256-bit, which is to be expected and is independent of the size or any other properties of the file being uploaded. Similarly, DTag is seen to be 254-bit.

Conclusion

In this project, we dealt with the clash between client-side encryption of data, to ensure it's security and deduplication of data, to prevent storage of redundant data. We designed a randomized encryption mechanism that enables protects the data against brute-force attacks. Based on this methodology, we developed a unique single-server scheme that enables the deduplication of client-side encrypted data in a cross-user fashion. We implement the proposed approach in a realistic scenario where Firebase is used as a cloud service provider. The main functions/algorithms we implemented as a part of RESIST are:

1.*Merkle Tree:* The merkle tree was implemented as two classes, a Merkle Tree class, and a Node class, in line with the algorithm as depicted in RESIST.

3. *EncryptKey:* This function constructs a Merkle Tree, using randomly selected leaf nodes of the tree, and by performing an XOR on the sibling paths of these nodes, computes CKey, and returns it.

2. *LearnKey:* The learnKey function helps a recipient of the file to recover the secret key, given some information shared by the file owner. This is done by reconstructing the Merkle Tree, by performing a tree traversal, using the nodes given by L, and their sibling paths.

As mentioned previously, we have used AES-256 for hashing, and UTF-8-encoded SHA-256 for encrypting the file.

The results obtained as part of RESIST are consistent with the theoretical analysis of the algorithm, and there is a notable reduction in the time taken for subsequent upload. This improvement in performance is even more pronounced with files of size >100MB and reflects the major strength of the RESIST algorithm.

References

1. J. R. Douceur, A. Adya,W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In Proceedings 22nd International Conference on Distributed Computing Systems, pages 617-624, 2002.
2. J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In 18th International Conference on Financial Cryptography and Data Security, pages 99–118, 2014.
3. M. Bellare, S. Keelveedhi, and T. Ristenpart. Message Locked encryption and secure deduplication. In 32nd Annual IACR Eurocrypt International Conference on the Theory and Applications of Cryptographic Techniques, pages 296–312, 2013.
4. P. Puzio, R. Molva, M. Onen, and S. Loureiro. ClouDedup: Secure deduplication with encrypted data for cloud storage. In 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, pages 363–370, 2013.
5. X. Liu, W. Sun, W. Lou, Q. Pei, and Y. Zhang. One-tag checker: Message-locked integrity auditing on encrypted cloud deduplication storage. In IEEE INFOCOM 2017 IEEE Conference on Computer Communications, pages 1–9, 2017.
6. R. Chen, Y. Mu, G. Yang, and F. Guo. BL-MLE: block-level message-locked encryption for secure large file deduplication. IEEE Trans. Inf. Forensic Secur, 10(12):2643–2652, 2015.
7. P. Anderson and L. Zhang. Fast and secure laptop backups with encrypted deduplication. In Proceedings of the 24th International Conference on Large Installation System Administration, pages 1–8, 2010.
8. M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In Annual Cryptology Conference, pages 374–391, 2013.
9. M. Miao, J.Wang, H. Li, and X. Chen. Secure multi-server-aided data deduplication in cloud computing. Pervasive and Mobile Computing, 24:129–137, 2015.
10. M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. Usenix Conference on Security, 2013.
11. Y. Zhang, C. Xu, H. Li, K. Yang, J. Zhou, and X. Lin. Healthdep: An efficient and secure deduplication scheme for cloud-assisted ehealth systems. IEEE Transactions on Industrial Informatics, 14(9):4101–4112, 2018.
12. D. L. Vo, F. Zhang, and K. Kim. A new threshold blind signature scheme from pairings. In The 2003 Symposium on Cryptography and Information Security, pages 699–702, 2003.
13. J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 874–885, 2015.
14. C. Yu. POSTER: Efficient cross-user chunk-level client-side data deduplication with symmetrically encrypted two-party interactions. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1763–1765, 2016.
15. P. Puzio, R. Molva, M. Onen, and S. Loureiro. PerfectDedup: Secure data deduplication. In 10th Data Privacy Management International Workshop / 4th International Workshop in Quantitative Aspects in Security Assurance, pages 150–166, 2016.
16. J. Stanek and L. Kencl. Enhanced secure thresholded data deduplication scheme for cloud storage. IEEE Transactions on Dependable and Secure Computing, 15(4):694–707, 2018.

17. G. Ha, H. Chen, C. Jia, R. Li, and Q. Jia. A secure deduplication scheme based on data popularity with fully random tags. In 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications, 2021.
18. S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In ACM Conference on Computer & Communications Security, pages 491-500, 2011.
19. J. Xu, E. C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pages 195-206, 2013.
20. J. Hur, D. Koo, Y. Shin and K. Kang. Secure Data Deduplication with Dynamic Ownership Management in Cloud Storage. IEEE Transactions on Knowledge and Data Engineering, pages 69-70, 2018.
21. J. Li, C. Qin, PPC. Lee and L. Jin. Rekeying for Encrypted Deduplication Storage. IEEE/IFIP International Conference on Dependable Systems & Networks, 2016.
22. H. Yuan, X. Chen, J. Li, T. Jiang, J.Wang and R. Deng. Secure Cloud Data Deduplication with Efficient Re-Encryption. IEEE Transactions on Services Computing, 15(1): 442-456, 2022.
23. Y. Wang, M. Miao, J. Wang and X. Zhang. Secure deduplication with efficient user revocation in cloud storage. Computer Standards & Interfaces, 2021.
24. G. Kan, C. Jin, H. Zhu, Y. Xu and N. Liu. An identity-based proxy re-encryption for data deduplication in cloud. Journal of Systems Architecture, 2021.
25. S. Jiang, T. Jiang and L. Wang. Secure and Efficient Cloud Data Deduplication with Ownership Management. IEEE Transactions on Services Computing, 13(6): 1152-1165, 2020.