Towards Efficient Re-encryption for Secure Client-side Deduplication in Public Clouds

Lei Lei[1,2,3], Quanwei Cai[1,2], Bo Chen[4,5⋆], and Jingqiang Lin[1,2,3]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
[4] Department of Computer Science, University of Memphis, Memphis, TN, USA
[5] Center for Information Assurance, University of Memphis, Memphis, TN, USA
bchen@mtu.edu

Abstract. By only storing a unique copy of duplicated data possessed by different users, data deduplication can significantly reduce storage cost, and is thus used extensively in cloud storage. When combining with confidentiality, deduplication will become problematic as encryption performed by different users may differentiate identical data. MLE (Message-Locked Encryption) is thus utilized to derive the same encryption key for the identical data. As keys may be leaked and users may be revoked, re-encrypting the outsourced data is of paramount importance to ensure continuous confidentiality. This problem is unfortunately not well addressed in deduplication-based encrypted cloud storage.

In this paper, we design SEDER, a SEcure client-side Deduplication system for cloud storage enabling Efficient Re-encryption. A salient advantage of SEDER is that it allows data owners to efficiently re-encrypt the data to ensure continuous data confidentiality for cloud storage using client-side deduplication, by smartly leveraging all-or-nothing transform, proofs of ownership as well as delegated re-encryption. Experimental evaluation validates the efficiency of SEDER.

Keywords: secure deduplication, client-side deduplication, re-encryption, cloud storage

1 Introduction

Cloud storage services are widely deployed nowadays. Popular services include Amazon S3 [1], Apple iCloud [5] and Microsoft Azure [6]. By using cloud services, data owners pay for the storage they use, eliminating the expensive cost of maintaining dedicated infrastructures.

As more and more users turn to clouds for storage, the amount of data stored in the clouds grows rapidly. Conventionally, the clouds simply store what have

⋆ Corresponding author.

been sent by the users. This unfortunately will lead to significant waste of storage space, as different users may upload identical data. A promising remediation is to perform data deduplication, in which the clouds only store a unique copy of duplicated data from different users to reduce the unnecessary waste of storage space. For example, recent research from Microsoft [34] showed that deduplication can achieve 50% and 90-95% storage savings in the standard file systems and backup systems, respectively. Almost all the existing popular file hosting services like Dropbox [4] and Box [2] perform data deduplication.

There are two popular data deduplication mechanisms: server-side deduplication and client-side deduplication. The main difference between them lies in the location of deduplication. In server-side deduplication, servers transparently perform deduplication on the data outsourced by the clients. In client-side deduplication, however, the servers and the clients cooperate to perform deduplication. Compared to the server-side deduplication, the client-side deduplication has a significant benefit that the clients do not need to upload the data which have already stored by the servers, significantly reducing bandwidth consumption. Therefore, the client-side deduplication has been used extensively in the public file hosting services [4, 2].

Encryption is necessary to protect confidentiality of sensitive data. However, it creates a severe obstacle for deduplication, as identical plain-texts may be encrypted into different cipher-texts by different users using different keys. Message-Locked Encryption (MLE) [13] is a cryptographic primitive which can resolve the aforementioned issue. MLE can derive encryption keys from messages being encrypted, such that different users are able to generate the same key for the identical data. Existing MLE schemes include CE [23], DupLESS [12], Duan Scheme [24], and LAP scheme [33].

To ensure continuous data confidentiality for encrypted cloud storage, re-encryption seems unavoidable, due to the potential key exposure [3, 8] or user revocation [41, 39]. Compared to conventional encrypted cloud storage, re-encryption in deduplication-based cloud storage is much more challenging, as it needs to be performed in such a manner that deduplication should not be disturbed. Li et al. proposed REED [32] to address the re-encryption problem for deduplication-based storage systems by smartly transforming the encrypted data such that they can be efficiently re-encrypted when revoking keys/users. REED however is specifically designed for *server-side deduplication*, which is not immediately applicable to the more beneficial *client-side deduplication*.

To design a secure client-side deduplication system which supports efficient re-encryption, we face several challenges: 1) To conform to the notion of storage outsourcing, we usually outsource both the data and the management of data [19, 16], such that once the data have been outsourced, the client will be involved as little as possible. It is thus challenging to allow re-encryption with least client intervention. 2) Different from server-side deduplication, in which the client will always upload the data being outsourced, in client-side deduplication, the client will not upload the data when he/she convinces the cloud server that he/she possesses the data which have already been stored in the cloud, and the keys

for decrypting these data will be disclosed to such clients after re-encryption. To ensure continuous confidentiality, we need a technique which can allow the cloud server to differentiate valid or invalid clients by efficiently verifying the possession of the file in the clients without being able to learn the plaintext of the file. 3) Considering the data stored in clouds are usually large in size, re-encrypting them may be prohibitively expensive. An efficient re-encryption approach is usually challenging.

In this paper, we propose SEDER, the first secure client-side deduplication system for cloud storage supporting efficient re-encryption. Our key insights are threefold: First, we leverage proofs of ownership (PoWs), by which we can ensure that after re-encryption, the new key is only disclosed to valid users who can prove they are the owners of the data. Second, we leverage all-or-nothing transform, by which it is possible to re-encrypt a file by only re-encrypting a small portion of it. Third, by observing that the existing proxy re-encryption can not be used in SEDER directly, we re-design a delegated re-encryption scheme, by which we can freely delegate the re-encryption to the cloud server without disclosing the plaintext data. This is advantageous as the client can be released from the burden of re-encryption and remains lightweight.

Comparison. Although both SEDER and REED [32] aim to address the re-encryption problem, they are different in multiple aspects: 1) REED resolves the re-encryption problem for server-side deduplication. However, SEDER resolves this problem for client-side deduplication, which has a more complicate design and hence a larger attack surface; 2) REED has very expensive computation/communication overhead for uploading files, as the client always needs to perform the expensive all-or-nothing transform and upload the file. In SEDER however, the amortized computation/communication overhead for uploading files is significantly reduced, as the clients do not need to perform the expensive all-or-nothing transform and upload the file if it exists in the cloud server; 3) In REED, the client is heavily involved in the re-encryption process which imposes significant burden on the client and contradicts with the notion of storage outsourcing. In SEDER however, the re-encryption is delegated to the cloud server who has rich computation resources, such that the client can always remain lightweight.

Contributions. We summarize our contributions in the following:

- We initiate the research of designing efficient re-encryption schemes for secure client-side deduplication in cloud storage.
- We design a delegated re-encryption scheme which can be used to delegate re-encryption to the untrusted third party. In addition, we design SEDER by smartly leveraging all-or-nothing transform (AONT), proofs of ownership (PoWs), and delegated re-encryption (DRE).
- We evaluate the performance of SEDER. Experimental results validate the efficiency of SEDER.

2 System and Adversarial Model

System model. We consider two entities: (1) *Cloud server* (CS). CS provides storage services and wants to perform client-side deduplication to reduce both storage and bandwidth cost; (2) *Cloud users* (U). The users outsource their data to the cloud. To maintain confidentiality of their outsourced data, they will encrypt the data before outsourcing them. Note that when the cloud user tries to upload a file that has been stored in the cloud server, CS will append this cloud user to the owner list of the corresponding file without requiring uploading the file again.

Adversarial model. We consider an honest-but-curious cloud server [41, 39]. CS will honestly store the encrypted data uploaded by the users, perform data deduplication, and respond the requests from the users. Moreover, CS will not disclose the data to any parties who fail to prove ownership of the data. However, it is curious and attempts to infer information about the encrypted users' data. In addition, there is a malicious entity (ME) who has obtained the key materials and tries to have access to the sensitive data.

Assumptions. We assume the MLE used in SEDER is secure[6]. All the communication channels among the CS and U are protected by SSL/TLS, so that any eavesdroppers cannot infer the messages being transmitted. Each entity (CS and U) has an asymmetric key pair, and the private key is well protected. We also assume that CS, U, and ME will not collude with each other.

3 Building Blocks

Message-Locked Encryption (MLE). MLE [13] is a scheme designed to derive encryption keys from messages being encrypted. In MLE, different users are able to generate the same key for the identical data. Existing MLE schemes include CE [23], DupLESS [12], Duan Scheme [24], and LAP scheme [33]. *A secure MLE scheme ensures that only the users who possess the same content can obtain the corresponding encryption key.* MLE usually uses symmetric encryption to encrypt messages.

All-or-Nothing Transform (AONT). AONT [35] is an unkeyed, invertible and randomized transformation. No one can succeed to perform the inverse transformation without knowing the entire output of the AONT. Specifically, given message m of s-blocks: $m = m_1||\dots||m_s$ where $||$ denotes block concatenation, AONT transforms m into message m' of t-blocks: $m' = m'_1||\dots||m'_t$ where $t \geq s+1$, and satisfies the following properties:

- Given m, $m' \leftarrow \mathsf{AONT}(m)$ can be computed efficiently. That is, the complexity of $\mathsf{AONT}(m)$ is polynomial to the length of m.
- Given m', $m \leftarrow \mathsf{AONT}^{-1}(m')$ can be computed efficiently.

[6] The MLE has been well investigated in the literature, and we believe a secure MLE can be found and directly applied here.

– Without knowing the entire m' (i.e., if one block is missing), the probability of recovering m is negligibly small.

In this paper, we instantiate AONTwith the package transform [35], which takes input an s-block message m and outputs an $(s+1)$-block message m'.

Proofs of Ownership (PoWs). PoWs [26] is a cryptographic protocol that allows the cloud server (as a verifier) to efficiently and securely validate that the data owner (as a prover), who wants to upload the data file that has been already stored in the server, really possesses that data file. Here the efficiency means that the communication is far less than the bandwidth of uploading the data file, and the security means that the data owner cannot cheat the server in non-negligible probability even if he/she possesses a large portion of the file and its metadata (e.g., hash value).

- witness $\leftarrow$ PoWs.Init(f) : Given a data file f, the verifier first preprocesses it and obtains some auxiliary data witness for the verification purpose:
  - The verifier uses an α-erasure-code EC to encode the data file f, where α denotes the erasure recovery capability.
  - The verifier computes the Merkle tree $MT_{H,b}(f)$ of the data file f, where H is a hash function used in computing Merkle tree and b is the size of a Merkle-tree leaf. The root value of Merkle-tree $r_{MT}(f)$ will be witness.
- challenge $\leftarrow$ PoWs.Challenge: When a prover declares that he/she owns a file f, the verifier chooses randomly x leaf indexes $l_1, l_2, ..., l_x$ and sends challenge $= (l_1, l_2, ..., l_x)$ to the prover, where ϵ is the soundness bound and x is the minimum integer satisfying $(1-\alpha)^x < \epsilon$.
- prof $\leftarrow$ PoWs.Prove($challenge, f$) : The prover builds the Merkle tree on top of data file f and returns the proof prof which consists of the sibling-paths of $l_1, l_2, ..., l_x$.
- $\{0,1\} \leftarrow$ PoWs.Verify(witness, challenge, prof): The verifier returns 1 if all the sibling-paths are valid with the Merkle tree root, and 0 otherwise.

4 SEDER

In this section, we first present a delegated re-encryption scheme (DRE) which allows to delegate re-encryption to an untrusted third party, and then elaborate the design of SEDER by leveraging DRE and other building blocks (Sec. 3).

4.1 Delegated Re-Encryption

Proxy re-encryption (PRE) [14, 27] allows a proxy to convert the ciphertext, which can only be decrypted by the delegator, into another ciphertext that can be decrypted by the delegatee, without leaking the plaintext to the proxy. Proxy re-encryption has been well studied and many promising features have been proposed, such as uni-direction, key privacy and no-interaction key generation. However, proxy re-encryption cannot be used here, because it cannot support

unlimited hops. Based on the scheme [11] which only supports single hop, we re-design a delegable re-encryption scheme supporting unlimited hops (DRE). The detail of DRE is as follows:

- $\mathsf{DRE.SetUp}(1^\ell)$: G is a multiplicative cyclic group of prime order q (q is an ℓ-bit system parameter, and ℓ is large enough). g is chosen from G at random and is known to all the parties.
- $\mathsf{DRE.KeyGen}(\mathsf{U}_i)$: Given user U_i, this algorithm generates the public key $\mathsf{pk}_i = \{g^{a_i}\}$ and $\mathsf{sk}_i = \{a_i\}$, where a_i is chosen at random from $\mathbb{Z}_q$.
- $\mathsf{DRE.Enc}(\mathsf{pk}_i, m)$: Message m is encrypted into $c_i = (c_{i_1}, c_{i_2}) = ((g^{a_i})^{k_i}, mg^{k_i})$, where k_i is chosen at random from $\mathbb{Z}_q$.
- $\mathsf{DRE.ReKeyGen}(\mathsf{sk}_i, \mathsf{pk}_j, c_{i_1})$: Given user U_i's private key sk_i, user U_j's public key pk_j (note that by running $\mathsf{PRE.KeyGen}(\mathsf{U}_j)$, U_j generates public key $\mathsf{pk}_j = \{g^{a_j}\}$ and $\mathsf{sk}_j = \{a_j\}$, where a_j is chosen at random from $\mathbb{Z}_q$) and c_{i_1}, the re-encryption key $rk_{i\to j}$ can be generated: $rk_{i\to j} = (rk_{i\to j_1}, rk_{i\to j_2}) = ((g^{a_j})^{k_j}, \frac{g^{k_j}}{(c_{i_1})^{1/a_i}})$, where k_j is randomly selected from $\mathbb{Z}_q$.
- $\mathsf{DRE.ReEnc}(rk_{i\to j}, c_i)$: Given the re-encryption key $rk_{i\to j} = (rk_{i\to j_1}, rk_{i\to j_2})$, the proxy can re-encrypt the ciphertext $c_i = (c_{i_1}, c_{i_2})$ to c_j by computing: $c_j = (c_{j_1}, c_{j_2}) = (rk_{i\to j_1}, c_{i_2} rk_{i\to j_2})$.
- $\mathsf{DRE.Dec}(\mathsf{sk}_j, c_j)$: Given the ciphertext $c_j = (c_{j_1}, c_{j_2})$, the user U_j decrypts it using $\mathsf{sk}_j = \{a_j\}$ by computing: $m = \frac{c_{j_2}}{(c_{j_1})^{1/a_j}}$.

4.2 Design Rational of SEDER

SEDER contains several key designs: First, we use AONT and DRE together to support efficient re-encryption of the outsourced file. Specifically, given a file, we apply MLE, obtaining the MLE ciphertext. MLE ensures that the same ciphertext will be generated from different users if the file content is the same. Then AONT is applied to MLE ciphertext, generating a set of data blocks. Note that without fetching all the data blocks, the MLE ciphertext cannot be recovered thanks to the interesting property of AONT. In this way, to re-encrypt a data file, the data owner only needs to re-encrypt one data block, rather than all the data blocks. In addition, by leveraging DRE, we can delegate the re-encryption process to the untrusted cloud server, without leaking the plaintext of the file. This is advantageous as we can eliminate the burden on the client who is supposed to be kept lightweight.

Second, to ensure only the valid data owners are able to decrypt the data being re-encrypted, we perform the following: 1) We leverage proofs of ownership (PoWs) to distinguish valid and invalid data owners. A valid data owner for a file should be able to prove his/her ownership as he/she possesses the file. When a data owner passes the verification, the cloud server will add him/her to the owner list of the file. 2) The cloud user who re-encrypts the file will compute new assisting information that is required to decode the file being re-encrypted. The new assisting information will only be disclosed to the valid data owners. The

malicious entity, even though have obtained the secret key, will not be able to pass the PoWs verification, and thus cannot obtain the new assisting information which is required to decode the re-encrypted file.

4.3 Design Details of SEDER

Let λ, γ and β be the security parameters. Let π_{DRE} be a delegated re-encryption scheme, such that $\pi_{\mathsf{DRE}} = (\pi_{\mathsf{DRE}}.\mathsf{SetUp}, \pi_{\mathsf{DRE}}.\mathsf{KeyGen}, \pi_{\mathsf{DRE}}.\mathsf{Enc}, \pi_{\mathsf{DRE}}.\mathsf{ReKeyGen}, \pi_{\mathsf{DRE}}.\mathsf{ReEnc}, \pi_{\mathsf{DRE}}.\mathsf{Dec})$. π_{sym} is a symmetric encryption scheme such that $\pi_{\mathsf{sym}} = (\pi_{\mathsf{sym}}.\mathsf{KeyGen}, \pi_{\mathsf{sym}}.\mathsf{Enc}, \pi_{\mathsf{sym}}.\mathsf{Dec})$, and π_{asym} is an asymmetric encryption scheme such that $\pi_{\mathsf{asym}} = (\pi_{\mathsf{asym}}.\mathsf{KeyGen}, \pi_{\mathsf{asym}}.\mathsf{Enc}, \pi_{\mathsf{asym}}.\mathsf{Dec})$. Let H_1 be a cryptographic hash function: $H_1 : \{0,1\}^* \rightarrow \{0,1\}^\lambda$. In the following, we describe the design details of SEDER, which contains six phases: SetUp, PreUpload, Upload, Update, Download and Delete.

SetUp: This is to bootstrap the system parameters, and initialize cryptographic parameters for cloud users and cloud server. The system runs $\pi_{\mathsf{DRE}}.\mathsf{SetUp}(1^\gamma)$ to initialize the system parameters. In addition,

- Cloud user U_i: He/She runs the key generation algorithm of asymmetric encryption scheme to generate the public/private key: $(\pi_{\mathsf{asym}}.\mathsf{pk}_{\mathsf{U}_i}, \pi_{\mathsf{asym}}.\mathsf{sk}_{\mathsf{U}_i}) \leftarrow \pi_{\mathsf{asym}}.\mathsf{KeyGen}(1^\beta)$.
- Cloud server: It runs the key generation algorithm of asymmetric encryption scheme to generate the public/private key: $(\pi_{\mathsf{asym}}.\mathsf{pk}_{\mathsf{CS}}, \pi_{\mathsf{asym}}.\mathsf{sk}_{\mathsf{CS}}) \leftarrow \pi_{\mathsf{asym}}.\mathsf{KeyGen}(1^\beta)$.

PreUpload: The PreUpload phase is run by the cloud user U_i before U_i uploads file f to the cloud. U_i uses MLE [13, 24, 23, 12] to obtain the file key k_f for file f. MLE can ensure that different users are able to generate the same key for the same file content.

Upload: The Upload phase is run by U_i to upload file f. Note that U_i has obtained the file key k_f during the PreUpload phase. U_i encrypts f by running $\mathsf{ct} = \pi_{\mathsf{sym}}.\mathsf{Enc}(k_f, f)$ (Note that this is part of MLE). U_i then computes a tag for f: $\mathsf{Tag}_f = H_1(\mathsf{ct})$, and sends Tag_f to cloud server CS. CS proceeds as follows:

Case 1: Tag_f does not exist in the cloud server: In this case, the cloud user conducts the following operations and uploads the corresponding file to the cloud:

- Given ct, U_i runs $\mathsf{PoWs.Init}(\mathsf{ct})$ to generate the witness.
- Assume that the encrypted file ct consists of s blocks: $\mathsf{ct} = \mathsf{ct}_1||\mathsf{ct}_2||...\ \mathsf{ct}_s$. U_i first applies all-or-nothing transform on ct, generating $s+1$ blocks, such that $\mathsf{ct}' \leftarrow \mathsf{AONT}(\mathsf{ct})$ where $\mathsf{ct}' = \mathsf{ct}'_1||\mathsf{ct}'_2||...||\mathsf{ct}'_s||\mathsf{ct}'_{s+1}$.
- U_i generates a pair of public/private key by applying $(\pi_{\mathsf{PRE}}.\mathsf{pk}_i, \pi_{\mathsf{PRE}}.\mathsf{sk}_i) \leftarrow \pi_{\mathsf{PRE}}.\mathsf{KeyGen}(\mathsf{U}_i)$.

- U_i randomly selects a data block ct'_z from $\mathsf{ct}'_1, \ldots, \mathsf{ct}'_{s+1}$. Then U_i applies the delegated re-encryption scheme π_{PRE} to encrypt ct'_z into c, such that $c = (c_1, c_2) = \pi_{\mathsf{DRE}}.\mathsf{Enc}(\pi_{\mathsf{DRE}}.\mathsf{pk}_i, \mathsf{ct}'_z)$. Therefore, the final ciphertext to be uploaded is: $\mathsf{ct}_{\mathsf{Upload}} = \mathsf{ct}'_1 || \ldots || \mathsf{ct}'_{z-1} || c || \mathsf{ct}'_{z+1} || \ldots || \mathsf{ct}'_{s+1}$.
- Using file key k_f, U_i encrypts $\pi_{\mathsf{DRE}}.\mathsf{sk}_i$ using $\pi_{\mathsf{sym}}.\mathsf{Enc}$ such that $\mathsf{ct}^*_{\mathsf{sym}} = \pi_{\mathsf{sym}}.\mathsf{Enc}(k_f, \pi_{\mathsf{DRE}}.\mathsf{sk}_i)$.
- Using the cloud server's public key $\pi_{\mathsf{asym}}.\mathsf{pk}_{\mathsf{CS}}$, U_i encrypts $\mathsf{ct}^*_{\mathsf{sym}}$ such that $\mathsf{ct}_{\mathsf{asym}} = \pi_{\mathsf{asym}}.\mathsf{Enc}(\mathsf{pk}_{\mathsf{CS}}, \mathsf{ct}^*_{\mathsf{sym}})$.
- U_i uploads $\mathsf{ct}_{\mathsf{Upload}}$, $\mathsf{witness}$, and $\mathsf{ct}_{\mathsf{asym}}$.
- After receiving the aforementioned information, CS organizes them in the format $< \mathsf{Tag}_f, \mathsf{ct}_{\mathsf{Upload}}, \mathsf{witness}, \mathsf{ct}_{\mathsf{asym}}$, user list $ul_{\mathsf{ct}_{\mathsf{Upload}}} >$. By decrypting $\mathsf{ct}_{\mathsf{asym}}$ using $\mathsf{sk}_{\mathsf{CS}}$, CS obtains the assisting information $\mathsf{ct}^*_{\mathsf{sym}}$, which will be distributed to valid cloud users in the following manner: encrypt $\mathsf{ct}^*_{\mathsf{sym}}$ using each user's public key and send the corresponding ciphertext to that user.

Case 2: Tag_f exists in the cloud server: To further confirm that U_i really possesses f, CS and U_i proceed as follows:

- CS runs $\mathsf{PoWs.Challenge}$ to generate a challenge which is sent to U_i.
- U_i computes a proof prof by running $\mathsf{PoWs.Prove}(\mathsf{challenge}, \mathsf{ct})$.
- CS further runs $\mathsf{PoWs.Verify}(\mathsf{witness}, \mathsf{challenge}, \mathsf{prof})$. If the output is 1, CS appends u_i to the user list $ul_{\mathsf{ct}_{\mathsf{Upload}}}$ and sends the assisting information of file f to U_i. Otherwise, CS terminates.

Update: When a data owner finds his file key is compromised, he needs to re-encrypt the corresponding file and makes sure that other data owners of the file can decrypt the latest ciphertext of the file. Thanks to AONT we only need to re-encrypt the encrypted block c rather than the entire outsourced file. Note that $c = (c_1, c_2)$, and c_1 is also sent to cloud users when CS distributes the assisting information $\mathsf{ct}^*_{\mathsf{sym}}$. The Upload phase is performed between cloud user U_j (who is on the user list of file f) and CS. The phase proceeds as:

- U_j runs $\mathsf{DRE.KeyGen}(\mathsf{U}_j)$ to generate a pair of public/private key, namely $(\mathsf{DRE.pk}_j, \mathsf{DRE.sk}_j) \leftarrow \mathsf{DRE.KeyGen}(\mathsf{U}_j)$.
- U_j decrypts $\mathsf{ct}^*_{\mathsf{sym}}$ using k_f, obtaining $\mathsf{DRE.sk}_i$.
- Using $\mathsf{DRE.sk}_i, c_1$ and $\mathsf{DRE.pk}_j$, U_j generates the delegable re-encryption key $rk_{i \to j} \leftarrow \mathsf{DRE.ReKeyGen}(\mathsf{DRE.sk}_i, \mathsf{DRE.pk}_j, c_1)$.
- Using k_f, U_j encrypts $\pi_{\mathsf{DRE}}.\mathsf{sk}_j$: $\mathsf{ct}^{\#}_{\mathsf{sym}} = \pi_{\mathsf{sym}}.\mathsf{Enc}(k_f, \pi_{\mathsf{DRE}}.\mathsf{sk}_j)$.
- Using $\pi_{\mathsf{asym}}.\mathsf{pk}_{\mathsf{CS}}$, U_j encrypts $\mathsf{ct}^{\#}_{\mathsf{sym}}$: $\mathsf{ct}'_{\mathsf{asym}} = \pi_{\mathsf{asym}}.\mathsf{Enc}(\mathsf{pk}_{\mathsf{CS}}, \mathsf{ct}^{\#}_{\mathsf{sym}})$.
- U_j sends $\mathsf{ct}'_{\mathsf{asym}}$ and $rk_{i \to j}$ to CS.
- CS runs $c' \leftarrow \pi_{\mathsf{DRE}}.\mathsf{ReEnc}(rk_{i \to j}, c)$ and replaces c with c'. In addition, CS replaces $\mathsf{ct}_{\mathsf{asym}}$ with $\mathsf{ct}'_{\mathsf{asym}}$, decrypts $\mathsf{ct}'_{\mathsf{asym}}$ obtaining $\mathsf{ct}^{\#}_{\mathsf{sym}}$, and distributes $\mathsf{ct}^{\#}_{\mathsf{sym}}$ to the users on the user list $ul_{\mathsf{ct}_{\mathsf{Upload}}}$.

Download: If user U_i wants to download $\mathsf{ct}_{\mathsf{Upload}}$ from the cloud server, U_i will send a download request $(Tag_f, download)$ to CS. When CS receives the request, CS

returns $\mathsf{ct}_{\mathsf{Upload}}$ to the requestor. U_i uses the file key and the assisting information to decode $\mathsf{ct}_{\mathsf{Upload}}$.

Delete: When CS receives a delete request $(Tag_f, delete)$ from user U_i, CS will delete U_i from $ul_{\mathsf{ct}_{\mathsf{Upload}}}$. If $ul_{\mathsf{ct}_{\mathsf{Upload}}}$ turns empty, CS will delete $\mathsf{ct}_{\mathsf{Upload}}$.

5 Security Analysis and Discussion

5.1 Security Analysis

Correctness and security of DRE. When receiving the ciphertext c_j, U_j can successfully decrypt it as follows:

$$\frac{c_{j_2}}{(c_{j_1})^{1/a_j}} = \frac{c_{i_2} rk_{i\to j_2}}{(rk_{i\to j_1})^{1/a_j}} = \frac{c_{i_2}(\frac{g^{k_j}}{(c_{i_1})^{1/a_i}})}{((g^{a_j})^{k_j})^{1/a_j}} = \frac{mg^{k_i}(\frac{g^{k_j}}{((g^{a_i})^{k_i})^{1/a_i}})}{g^{k_j}} = \frac{mg^{k_j}}{g^{k_j}} = m.$$

In addition, by knowing g, user U_i's public key g^{a_i} and user U_j's public key g^{a_j}, the proxy cannot learn anything about plaintext m by observing: 1) $c_i = (c_{i_1}, c_{i_2}) = ((g^{a_i})^{k_i}, mg^{k_i})$; and 2) $rk_{i\to j} = (rk_{i\to j_1}, rk_{i\to j_2}) = ((g^{a_j})^{k_j}, \frac{g^{k_j}}{(c_{i_1})^{1/a_i}}) = ((g^{a_j})^{k_j}, \frac{g^{k_j}}{g^{k_i}})$, due to the hardness of discrete logarithm problem.

Data confidentiality. In the following, we show that cloud server CS and the malicious entity ME are not able to learn the plaintext of the encrypted data.

CS possesses the following information: Tag_f, $\mathsf{witness}$, $\mathsf{ct}^*_{\mathsf{sym}}$, $\mathsf{ct}^{\#}_{\mathsf{sym}}$ and $\mathsf{ct}_{\mathsf{Upload}}$. By knowing Tag_f, CS usually cannot learn ct as H_1 is a cryptographic hash function. Even if CS can learn something about ct, without knowing the file key k_f (note that we assume CS, U, and ME will not collude with each other, and MLE ensures that only the valid cloud user U who possesses the plaintext of the file can obtain k_f), CS still cannot learn the plaintext of the file. Also, as $\mathsf{witness}$ is computed from ct, it cannot help to learn the plaintext of the file. In addition, by having access to $\mathsf{ct}^*_{\mathsf{sym}}$ or $\mathsf{ct}^{\#}_{\mathsf{sym}}$, CS cannot decrypt either of them as it cannot have access to the file key k_f. This can ensure that CS cannot decrypt either block c or c'. By knowing $\mathsf{ct}_{\mathsf{Upload}}$, CS is not able to obtain ct, as it is not able to decrypt either block c or c' (security of AONT), let alone the plaintext of the file.

ME, though have obtained the key materials (i.e., the old assisting information and the file key k_f), is not able to prove to CS the ownership of f, and is thus not able to obtain the new assisting information from CS. Upon having access to $\mathsf{ct}_{\mathsf{Upload}}$, by using the old assisting information, ME cannot decrypt the re-encrypted block c, and is thus not able to decode $\mathsf{ct}_{\mathsf{Upload}}$ to obtain ct (security of AONT). Therefore, even if he/she can have access to the file key k_f, he/she is not able to obtain f.

5.2 Discussion

Zero-day attack. SEDER is vulnerable to the zero-day attack, in which the key is leaked and the re-encryption has not been performed. During this period, the

adversary can have access to the original file using the obtained key materials. This seems to be unavoidable and we currently do not have a good solution for mitigating such a strong attack.

Supporting user revocation. Considering the scenario that each data owner has a few users, and the data owner wants to revoke a certain user, which requires re-encrypting the outsourced file. SEDER can be simply adapted to this scenario, but may face an additional attack: the malicious user can store the decrypted version of block c, and is always able to decode $\mathsf{ct}_{\mathsf{Upload}}$, even though he/she is not able to obtain the new assisting information. This attack can be mitigated by re-encrypting a randomly chosen block during each re-encryption process.

The nature of the storage being supported by SEDER. Currently, SEDER only supports archival storage [10, 22, 15, 20]. We will extend SEDER to support dynamic storage (i.e., supporting dynamic operations like insert, delete, modify, and append [25, 39, 17, 18]) in our future work.

6 Experimental Evaluation

6.1 Experimental Setup

We evaluated the overhead of each operation in SEDER. We choose security parameters as: $\gamma = 256$, $\beta = 1024$. The length of MLE file key is 128 bits. We used OpenSSLv1.0.0e [7] for data encryption/decryption and large number modular operations. The symmetric and asymmetric encryption/decryption function are instantiated by AES-128 and RSA-1024 respectively. Throughout the experiment, the client and the server both ran on local workstations with Intel i7-2600 (3.4 GHz) CPU and 10GB RAM. In our experiment, the added block implying the random key k for AONT is computed as $m'_{s'} = k \oplus m'_1 \oplus m'_2 \oplus ... \oplus m'_s$, which is a little different from [35].

The PreUpload phase simply relies on the existing MLE schemes. Therefore, we only focus on the performance overhead in Upload, Update, and Download. We evaluated the processing time for these phases with data size varying from 100 MB to 2GB. Results are averaged over 100 runs.

6.2 Evaluation Results

Upload. The computation overhead required for Upload phase is shown in Figure 1(a) and 1(b). In SEDER, the cloud user has to perform AONT which consists multiple AES encryption (determined by the size of the data) and XOR operations on the regular encrypted data. We observed that the computation overhead for AONT increases with the size of the processed data and is slightly larger than the regular data encryption.

When the cloud user (User 1) has accomplished the data encryption and AONT, it performs the asymmetric encryption on one block of the AONT transformed data. The user generates the asymmetric key pair, encrypts this block with the corresponding public key. He/she then encrypts the corresponding private key using the file key, and the resulting assisting information is further

encrypted by the public key of the cloud server (the process is denoted as $User1BlEncrypt$). From Figure 1(a), we observed that $User1BlEncrypt$ is the same for different data sizes, and is less than 0.5 ms, which is rather small. The cloud server decrypts the information sent by User 1, obtaining the assisting information. It then encrypts the assisting information with other user's public key (the process is denoted as $CloudReturnKey$). From Figure 1(b), we observed that $CloudReturnKey$ is the same for different data sizes, and is approximately 5.5 ms, which is also very small. Note that RSA decryption is usually slower than encryption, which explains why $CloudReturnKey$ is more expensive than $User1BlEncrypt$.

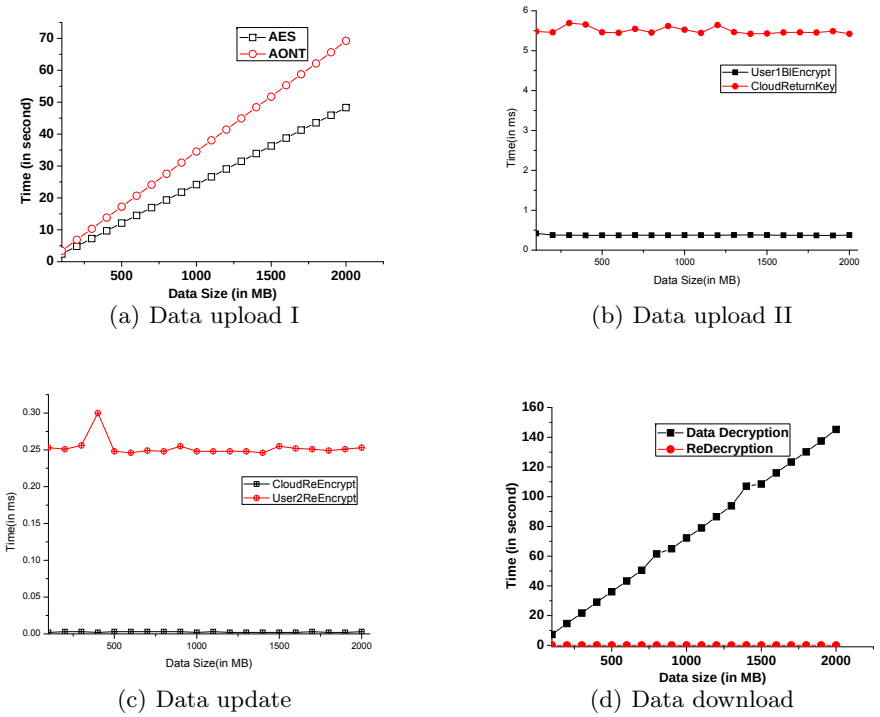
Update. When a user (User 2) wants to update the key, he/she requires the cloud server to collaborate in performing the re-encryption. The user generates the delegated re-encryption key $rk_{i\to j}$, encrypts $\pi_{\mathsf{DRE}}.\mathsf{sk}_j$ with the file key, and the resulting ciphertext is future encrypted with the cloud server's public key (the process is denoted as $User2ReEncrypt$ in Figure 1(c)). He/she then requires the cloud server to complete the re-encryption (this process is denoted as $CloudReEncrypt$ in Figure 1(c)) and re-distribute the new assisting information. We observed that the performance overhead for either $User2ReEncrypt$ or $CloudReEncrypt$ is independent of the size of data, and is very small (less than 0.5 ms) compared to the time for re-encrypting the entire data.

Download. To download the data, the user needs to decrypt the re-encrypted block (the process is denoted as $ReDecryption$ in Figure 1(d)). Then, the user performs the inverse operation of AONT and decrypts the resulting data to obtain the final plain-text (the process is denoted as $DataDecryption$). From Figure 1(d), we observed that the performance overhead caused by $ReDecryption$ is the same (less than 6 ms) for different data sizes, which is negligible compared to the cost of $DataDecryption$ (more than 7 seconds for 100 MB data). The processing time of $DataDecryption$ increases with the size of data, and is almost equal to the time required to perform data encryption plus AONT(Figure 1(a)).

7 Related Work

Bellare et al. [13] formalized a new cryptographic primitive "MLE" (Message-Locked Encryption) to derive encryption/decryption key from the message being encrypted/decrypted. This new primitive can facilitate performing deduplication over data encrypted by different users.

Douceur et al. [23] proposed convergent encryption (CE), the first MLE scheme in which the key used to encrypt a file is the hash value of the file, so that the same file possessed by different users can be encrypted by the same key. CE has been used in a few systems [21, 28, 37, 40, 31, 36, 30]. CE however, is vulnerable to an off-line dictionary attack as file data are usually from a predictable space [13]. Following CE, several MLE schemes were proposed. Bellare et al. proposed DupLESS [12] to mitigate the off-line dictionary attack using per-client rate limiting strategy. Specifically, they introduced a key server dur-



(a) Data upload I (b) Data upload II

(c) Data update (d) Data download

Fig. 1. SEDER performance

ing key derivation to restrict the number of signature requests allowed for a user during a fixed time interval. Duan [24] proposed another MLE scheme based on distributed oblivious key generation. Liu et al. [33] proposed a new MLE scheme by eliminating the additional independent servers. In their scheme, users use PAKE to exchange the file encryption key with the help of cloud servers. In order to prevent online dictionary attack, their scheme realizes a per-file rate limiting strategy (every user limits under the number of key agreement he/she takes part in).

REED [32] aimed at addressing the key revocation problem for secure server-side deduplication in cloud storage. In order to efficiently replace old keys and re-encrypt the data, REED introduced two special all-or-nothing transforms derived from CAONT (CANOT is a special case of all-or-nothing transforms in which the key used for AONT is the hash of message). ClearBox [9] is a transparent deduplication scheme, in which storage service providers can attest to users the number of owners of a file transparently, so that users can share the fee of storing the same file. Li et al. [29] proposed *SecCloud*$^+$ to achieve data integrity and deduplication simultaneously. Tang et al. [38] performed data deduplication on CP-ABE.

8 Conclusion

In this paper, we propose SEDER to address the re-encryption problem for secure client-side deduplication in cloud storage. Security analysis and experimental results show that our design brings in acceptable overhead in various phases while being able to ensure continuous confidentiality for encrypted cloud storage based on client-side deduplication.

Acknowledgments. This work was supported by National Program on Key Basic Research Project of China (973) (2014CB340603). The authors would like to thank the valuable discussion from Qingji Zheng. Bo Chen would also like to thank the support from Center for Information Assurance at the University of Memphis.

References

1. Amazon simple storage service, http://aws.amazon.com/cn/s3/
2. Box, https://www.box.com/
3. Debian security advisory, https://www.debian.org/security/2008/dsa-1571
4. Dropbox, https://www.dropbox.com/
5. Icloud, https://www.icloud.com/
6. Microsoft azure, http://www.windowsazure.cn/?fb=002
7. Openssl, https://www.openssl.org/
8. These are not the certs youre looking for, http://dankaminsky.com/2011/08/31/notnotar/
9. Armknecht, F., Bohli, J.M., Karame, G.O., Youssef, F.: Transparent data deduplication in the cloud. In: The ACM Sigsac Conference. pp. 886–900 (2015)
10. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 598–609. Acm (2007)
11. Ateniese, G., Fu, K., Green, M., Hohenberger, S.: Improved proxy re-encryption schemes with applications to secure distributed storage. Acm Transactions on Information and System Security 9(1), 1–30 (2006)
12. Bellare, M., Keelveedhi, S., Ristenpart, T.: DupLESS: Server-aided encryption for deduplicated storage. In: USENIX Conference on Security. pp. 179–194 (2013)
13. Bellare, M., Keelveedhi, S., Ristenpart, T.: Message-locked encryption and secure deduplication. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 296–312. Springer (2013)
14. Blaze, M., Bleumer, G., Strauss, M.: Divertible protocols and atomic proxy cryptography. In: EUROCRYPT '98. pp. 127–144. Springer (1998)
15. Bowers, K.D., Juels, A., Oprea, A.: Hail: a high-availability and integrity layer for cloud storage. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 187–198. ACM (2009)
16. Chen, B., Ammula, A.K., Curtmola, R.: Towards server-side repair for erasure coding-based distributed storage systems. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. pp. 281–288. ACM (2015)
17. Chen, B., Curtmola, R.: Robust dynamic provable data possession. In: 2012 32nd International Conference on Distributed Computing Systems Workshops. pp. 515–525. IEEE (2012)
18. Chen, B., Curtmola, R.: Robust dynamic remote data checking for public clouds. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 1043–1045. ACM (2012)
19. Chen, B., Curtmola, R.: Towards self-repairing replication-based storage systems using untrusted clouds. In: Proceedings of the third ACM conference on Data and application security and privacy. pp. 377–388. ACM (2013)

20. Chen, B., Curtmola, R., Ateniese, G., Burns, R.: Remote data checking for network coding-based distributed storage systems. In: Proceedings of the 2010 ACM workshop on Cloud computing security workshop. pp. 31–42. ACM (2010)
21. Cox, L.P., Murray, C.D., Noble, B.D.: Pastiche: making backup cheap and easy. Acm Sigops Operating Systems Review 36(SI), 285–298 (2002)
22. Curtmola, R., Khan, O., Burns, R., Ateniese, G.: Mr-pdp: Multiple-replica provable data possession. In: Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on. pp. 411–420. IEEE (2008)
23. Douceur, J.R., Adya, A., Bolosky, W.J., Dan, S., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: International Conference on Distributed Computing Systems. pp. 617–624 (2002)
24. Duan, Y.: Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In: CCSW. pp. 57–68 (2014)
25. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. ACM Transactions on Information and System Security (TISSEC) 17(4), 15 (2015)
26. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: ACM Conference on Computer and Communications Security. pp. 491–500. ACM (2011)
27. Ivan, A.A., Dodis, Y.: Proxy cryptography revisited. In: Network and Distributed System Security Symposium, NDSS 2003 (2003)
28. Killijian, M.O., Powell, D., Es, L.: A survey of cooperative backup mechanisms. Ubiquitous Computing (2006)
29. Li, J., Li, J., Xie, D., Cai, Z.: Secure auditing and deduplicating data in cloud. IEEE Transactions on Computers (1), 1–1 (2016)
30. Li, J., Chen, X., Li, M., Li, J., Lee, P.P.C., Lou, W.: Secure deduplication with efficient and reliable convergent key management. IEEE Transactions on Parallel and Distributed Systems 25(6), 1615–1625 (2014)
31. Li, J., Li, Y.K., Chen, X., Lee, P.P.C., Lou, W.: A hybrid cloud approach for secure authorized deduplication. IEEE Transactions on Parallel and Distributed Systems 26(5), 1206–1216 (2015)
32. Li, J., Qin, C., Lee, P.P.C., Li, J.: Rekeying for encrypted deduplication storage. In: IEEE/IFIP International Conference on Dependable Systems and Networks (2016)
33. Liu, J., Asokan, N., Pinkas, B.: Secure deduplication of encrypted data without additional independent servers. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 874–885 (2015)
34. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. ACM Transactions on Storage 7(4), 1–1 (2012)
35. Rivest, R.L.: All-or-nothing encryption and the package transform. In: International Workshop on Fast Software Encryption. pp. 210–218. Springer (1997)
36. Stanek, J., Sorniotti, A., Androulaki, E., Kencl, L.: A secure data deduplication scheme for cloud storage. In: FC2014. pp. 99–118. Springer Berlin Heidelberg (2014)
37. Storer, M.W., Greenan, K., Long, D.D.E., Miller, E.L.: Secure data deduplication. In: ACM Workshop on Storage Security and Survivability. pp. 1–10 (2008)
38. Tang, H., Cui, Y., Guan, C., Wu, J., Weng, J., Ren, K.: Enabling ciphertext deduplication for secure cloud storage and access control. In: ACM on Asia Conference on Computer and Communications Security (2016)
39. Wang, Q., Wang, C., Li, J., Ren, K., Lou, W.: Enabling public verifiability and data dynamics for storage security in cloud computing. In: European Conference on Research in Computer Security. pp. 355–370. Springer (2009)
40. Xu, J., Chang, E.C., Zhou, J.: Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In: ACM SIGSAC Symposium on Information, Computer and Communications Security. pp. 195–206 (2013)
41. Yu, S., Wang, C., Ren, K., Wenjing, L.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: INFOCOM 2010. pp. 1–9. IEEE (2010)