Rekeying for Encrypted Deduplication Storage

Jingwei Li[1,*], Chuan Qin[1], Patrick P. C. Lee[1], and Jin Li[2]
[1]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[2]School of Computer Science and Educational Software, Guangzhou University
lijw1987@gmail.com, {cqin,pclee}@cse.cuhk.edu.hk, lijin@gzhu.edu.cn

***Abstract*—Rekeying refers to an operation of replacing an existing key with a new key for encryption. It renews security protection, so as to protect against key compromise and enable dynamic access control in cryptographic storage. However, it is non-trivial to realize efficient rekeying in encrypted deduplication storage systems, which use deterministic content-derived encryption keys to allow deduplication on ciphertexts. We design and implement REED, a rekeying-aware encrypted deduplication storage system. REED builds on a deterministic version of all-or-nothing transform (AONT), such that it enables secure and lightweight rekeying, while preserving the deduplication capability. We propose two REED encryption schemes that trade between performance and security, and extend REED for dynamic access control. We implement a REED prototype with various performance optimization techniques. Our trace-driven testbed evaluation shows that our REED prototype maintains high performance and storage efficiency.**

I. Introduction

Data explosion has raised a scalability challenge to cloud storage management. For example, Aberdeen Research [25] reports that the average size of backup data for a medium-size enterprise is 285TB, and meanwhile, faces an annual growth rate of about 24-27%. *Deduplication* is one plausible solution that makes storage management scalable. Its idea is to eliminate the storage of redundant messages that have identical content, by keeping only one message copy and referring other redundant messages to the copy through small-size pointers. Deduplication is shown to effectively reduce storage space for some workloads, such as backup data [54]. It has also been deployed in today's commercial cloud storage services (e.g., Dropbox, Google Drive, Bitcasa, Mozy, and Memopal) for saving maintenance costs [34].

To protect against content leakage of outsourced data, cloud users often want to store encrypted data in the cloud. Traditional symmetric encryption is incompatible with deduplication: it assumes that users encrypt messages with their own distinct keys, and hence identical messages of different users will lead to distinct ciphertexts and prohibit deduplication. To enable *encrypted ~~duplication~~ storage* (i.e., encrypting the stored data while preserving the deduplication capability), Bellare *et al.* [18] define a cryptographic primitive called *message-locked encryption (MLE)*, which derives the encryption key from the message itself through a uniform derivation function, so that the same message deterministically returns the same ciphertext through symmetric encryption. One well-known instantiation of MLE is *convergent encryption (CE)* [28], which uses the cryptographic hash of the message content as the derivation function. In fact, implementations of MLE and CE have been extensively deployed and evaluated in encrypted deduplication storage systems (e.g., [9], [10], [17], [24], [52], [57]).

However, existing encrypted deduplication storage systems do not address *rekeying*, an operation that replaces an existing key with a new key so as to renew security protection. Rekeying is critical not only for protecting against key compromise that has been witnessed in real-life accidents [26], [37], [53], but also for enabling dynamic access control to revoke unauthorized users from accessing data in cryptographic storage [14], [30], [36], [43]. However, realizing efficient rekeying in encrypted deduplication storage is challenging. Since the encryption key of each message in MLE is obtained from a deterministic derivation function (e.g., a hash function), if we renew the key by renewing the derivation function, then any newly stored message encrypted by the new key can no longer be deduplicated with the existing identical message; if we re-encrypt all existing messages with the new key obtained from the renewed derivation function, then there will be tremendous performance overheads for processing large quantities of messages.

This paper presents *REED*, a rekeying-aware encrypted deduplication storage system that aims for secure and lightweight rekeying, while preserving content similarity for deduplication. REED augments MLE with the idea of *all-or-nothing transform (AONT)* [47], which transforms a secret into a package, such that the secret cannot be recovered without knowing the entire package. REED encrypts a small part of the package with a key that is subject to rekeying, while the remaining large part of the package is generated from a deterministic variant of AONT [38] to preserve content similarity. We show that this approach enables secure and lightweight rekeying, and simultaneously, allows deduplication. The contributions of this paper are summarized as follows.

First, we propose two encryption schemes for REED, namely basic and enhanced, that trade between performance and security. Both schemes enable lightweight rekeying, while the enhanced scheme is resilient against key leakage through a more expensive encryption than the basic scheme.

Second, we extend REED with dynamic access control. We demonstrate how REED integrates existing primitives, namely ciphertext-policy attribute-based encryption (CP-ABE) [19] and key regression [30], so as to control the access privileges to different files.

Third, we implement a proof-of-concept REED prototype. Our REED prototype leverages various performance optimization techniques, such as batching, caching, and parallelization, to mitigate computational and I/O overheads.

Finally, we conduct extensive trace-driven evaluation on

*Jingwei Li is now with Center for Cyber Security, University of Electronic Science and Technology of China. This work was done when he was with the Chinese University of Hong Kong.

our REED prototype in a LAN testbed. REED shows lightweight rekeying. For example, it only takes 3.4s to re-encrypt an 8GB file with a new key (in active revocation). It also maintains a storage saving of 98.6% in a real-world dataset.

The remainder of the paper proceeds as follows. Section II motivates the need of rekeying for encrypted deduplication storage. Section III defines our threat model and security goals. Section IV presents the design of REED, and Section V presents its implementation details. Section VI presents our testbed experimental results. Section VII reviews related work, and finally, Section VIII concludes the paper.

II. Background and Motivation

A. Encrypted Deduplication Storage

Deduplication exploits content similarity to achieve storage efficiency. Each message is identified by a *fingerprint*, computed as a cryptographic hash of the content of the message. We assume that two messages are identical (distinct) if their fingerprints are identical (distinct), and that the fingerprint collision of two distinct messages has a negligible probability in practice [20]. Deduplication stores only one copy of identical messages, and refers any other identical message to the copy using a small-size pointer. This paper focuses on *chunk-level deduplication*, which divides file data into fixed-size or variable-size chunks, and removes duplicates at the granularity of chunks. In this paper, we use the terms "messages" and "chunks" interchangeably to refer to the data units operated by deduplication.

Message-locked encryption (MLE) [18] is a cryptographic primitive that provides confidentiality guarantees for deduplication storage. It applies symmetric encryption to encrypt a message with a key called the *MLE key* that is derived from the message itself, so as to produce a deterministic ciphertext. Two identical (distinct) messages will lead to identical (distinct) ciphertexts, so deduplication remains plausible. A special case of MLE is *convergent encryption (CE)* [28], which directly uses the message's fingerprint as the MLE key.

However, MLE (including CE) is inherently vulnerable to brute-force attacks, and achieves security only for *unpredictable* messages [18]. Specifically, suppose that a target message is known to be drawn from a finite space. Then an adversary can sample all messages, derive the MLE key of each message, and compute the corresponding ciphertexts. If one of the computed ciphertexts equals the ciphertext of the target message, then the adversary can deduce the target message.

To address the unpredictability assumption, DupLESS [17] implements *server-aided MLE*. It uses a dedicated key manager to generate an MLE key for a message based on the message's fingerprint and additionally a system-wide secret that is independent of the message content. If the key manager is secure, then the ciphertexts appear to be derived from a random key space, and this provides confidentiality guarantees even for predictable messages. Even if the key manager is compromised, DupLESS still preserves confidentiality for unpredictable messages. To make MLE key generation robust, DupLESS introduces two mechanisms. First, it uses the oblivious pseudo-random function (OPRF) [31] to "blind" a fingerprint that is to be processed by the key manager, such that the key manager can return the MLE key without knowing the original fingerprint. Second, the key manager rate-limits the key generation requests to protect against brute-force attacks.

In this work, we focus on encrypted deduplication storage based on MLE. Like DupLESS, we maintain a dedicated key manager that is responsible for MLE key generation, thereby resisting brute-force attacks.

B. Rekeying

We define *rekeying* as the generic process of updating an existing old key to a new key in encrypted storage, such that the old key will be revoked, and all subsequently stored files will be encrypted by the new key. We argue that rekeying is critical for renewing security protection for encrypted deduplication storage in two aspects: *key protection* and *access revocation*.

Key protection: There have been real-life cases that indicate how adversaries make key compromise plausible through various system vulnerabilities, such as design flaws [26], [37], [50] and programming errors [53]. These threats also apply to storage systems, since adversaries can compromise file encryption keys and recover all encrypted files. In addition to key compromise, every cryptographic key in use is associated with a lifetime, and required to be replaced, once the key reaches the end of its lifetime [15]. Rekeying is thus critical to immediately update the compromised or expired keys, so that the stored files remain protected by the new keys.

Since deduplication implies the sharing of data across multiple files and users, rekeying in encrypted deduplication storage is more critical than traditional encrypted storage without deduplication. In particular, the security of a message depends on its MLE key. The leakage of the MLE key may imply the compromise of multiple files that share the message.

Access revocation: Organizations increasingly outsource large-scale projects to cloud storage providers for efficient management. We consider a special case in genome research. Genome researchers increasingly leverage cloud services for genome data storage due to the huge volume of genome datasets [51]. Some cloud services, such as Google Genomics [2] and Amazon [5], have also set up specific platforms for organizing and analyzing genome information. With deduplication, the storage of genome data can be significantly reduced, for example, by 83% in real deployment [4]. However, some genome datasets, such as those produced by disease sequencing projects, are potentially identifiable and must be protected. Thus, dataset owners must properly protect the deduplicated genome data with encryption and multiple dimensions of access control [41]. When a researcher leaves a genome project, it is necessary to revoke the researcher's access privilege to the genome data.

Rekeying can be used to revoke users' access rights by re-encrypting ciphertexts (e.g., the genome data in the previous example) with new keys and making old keys inactive. There are two revocation approaches for existing stored data [14]: (i) *lazy revocation*, in which re-encryption of a stored file is deferred until the next update to the file, and (ii) *active revocation*, in which the stored files are immediately re-encrypted with the new key for up-to-date protection, at the expense of incurring additional performance overheads.

C. Challenges

Enabling rekeying in encrypted deduplication storage is a non-trivial issue. MLE keys are often derived from messages via a global *key derivation function*, such as a hash function in CE [28] or a keyed pseudo-random function in DupLESS [17]. A straightforward rekeying approach is to update the key derivation function directly. However, this approach compromises deduplication. Specifically, a new message cannot be deduplicated with the existing identical message, because the messages are now encrypted with different MLE keys derived from different derivation functions. If we re-encrypt all existing messages with new MLE keys, it incurs significant overheads.

There are possible rekeying approaches, but we argue that they have limitations. One approach is based on *layered* encryption [10]. Each deduplicated message is first encrypted with its MLE key, and the MLE key is further encrypted with a master key associated with each user. The security now builds on the master key. Rekeying can simply be done by updating the master key, and re-encrypting the MLE key with the new master key. This approach does not change the MLE key, so any new message can be deduplicated with the existing identical message. Its drawback is that every ciphertext remains encrypted by the same MLE key. If an MLE key is leaked, then the corresponding message can be identified. Another approach is *proxy re-encryption* [13], which transforms a ciphertext encrypted with an old key into another ciphertext encrypted with a new key. However, proxy re-encryption is a public-key primitive and is inefficient when encrypting large-size messages. To this end, we pose the following question: *how to enable secure and lightweight rekeying, while preserving the deduplication capability?*

III. Overview

REED is a rekeying-aware encrypted deduplication storage system designed for a single enterprise or organization in which multiple users want to outsource storage to a remote third-party cloud provider. We target the workloads that have high content similarity, such as backup or genome data (see Section II-B), so that deduplication can effectively remove duplicates and improve storage efficiency.

REED aims to achieve secure and lightweight rekeying, while preserving deduplication. In particular, it enables dynamic access control by controlling which group of users can access a file. It supports both lazy and active revocations (see Section II-B); for the latter, the stored files can be re-encrypted with low overheads.

A. Architecture

Figure 1 presents an overview of the architecture of REED. REED follows a client-server architecture. In each user machine, we deploy a *REED client* (or *client* for short) as a software layer that provides a secure interface for a user to access and manage files in remote storage. To perform uploads, the client takes a file (e.g., a snapshot of a file system folder) as an input from its co-located user machine. It divides the file data into chunks, encrypts them, and uploads the encrypted chunks to the cloud. We assume that the file has a sufficiently large size (e.g., GB scale), and can be divided into a large number of chunks of small sizes (e.g., KB scale).

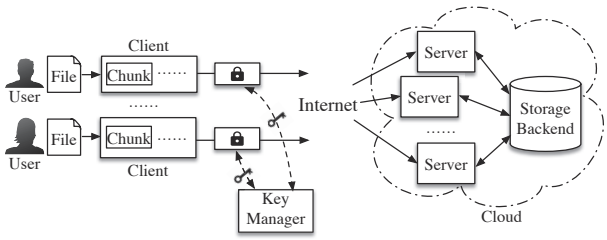


Fig. 1. REED architecture.

As in DupLESS [17], REED deploys a *key manager*, which provides an interface for a client to access MLE keys for encrypted storage. Each client communicates with the key manager to perform necessary cryptographic operations. We implement server-aided MLE as in DupLESS to protect all chunks, including predictable and unpredictable ones (see Section II-A). This work considers a single key manager, while our design can be generalized for multiple key managers for improved availability [29].

REED performs server-side deduplication. In the cloud, we deploy a *REED server* (or *server* for short) for storage management. The server maintains a fingerprint index that keeps track of all chunks that have been uploaded to the cloud. For a given received chunk, the server checks by fingerprint if the chunk has already been uploaded by the same or a different client. If the chunk is new, it stores the chunk and inserts the chunk fingerprint to the index. Finally, the server stores the encrypted chunks and metadata in the *storage backend* of the cloud. For example, if we choose Amazon's cloud services, we can rent an EC2 virtual machine for a REED server, and use S3 as the storage backend.

B. Threat Model

We consider an *honest-but-curious* adversary that aims to learn the content of the files in outsourced storage. The adversary can take the following actions. First, it can compromise the cloud (including any hosted server and the storage backend) to have full access to all stored chunks and keys. Also, it can collude with a subset of unauthorized or revoked clients, and attempt to learn the files that are beyond the access scope of the colluded clients. Furthermore, it can monitor the activities of the clients, identify the MLE keys returned by the key manager, and extract the files owned by the monitored clients.

Our threat model makes the following assumptions. We assume the communication between a client and the key manager is encrypted and authenticated (e.g., using SSL/TLS), so as to defend against any eavesdropping activity in the network. Each client and the key manager adopt oblivious key generation [17], so that the key manager cannot infer the fingerprint information and learn the message content. We also assume that the key manager is deployed in a fully protected zone, and an adversary cannot compromise or gain access to the key manager.

We do not consider the threat in which an adversary launches on-line brute force attacks from a compromised client against the key manager, since the key manager can rate-limit the query rate of each client [17]. REED can be deployed in conjunction with remote data checking [12], [35] to efficiently

check the integrity of outsourced files against malicious corruptions. Since REED performs server-side deduplication, it does not introduce any side channel in deduplication [33], [34].

To this end, REED focuses on two main security goals. First, REED ensures *confidentiality*, such that chunk contents are kept secret against any honest-but-curious adversary (e.g., any unauthorized user or cloud). In addition, REED prevents revoked users from accessing any new file or update. Second, REED ensures *integrity*, such that when a client downloads a chunk, it can check if it is intact or corrupted.

IV. REED Design

A. Main Idea

REED builds security simultaneously on two types of symmetric keys: a file-level secret key per file (or *file key* for short) and a chunk-level MLE key for each chunk (or *MLE key* for short). During rekeying, REED only needs to renew the file key, while the MLE keys of all chunks remain unchanged. We argue that this rekeying approach achieves our security goals (see Section III-B), while preserving deduplication effectiveness and allowing lightweight re-encryption in active revocation (see Section IV-C).

REED uses *all-or-nothing transform* (AONT) [47] as the underlying cryptographic primitive. AONT is an unkeyed, randomized encryption mode that transforms a message into a ciphertext called the *package*, which has the property that it is computationally infeasible to be reverted back to the original message without knowing the entire package. The original AONT design prohibits deduplication, since its transformation takes a random key as an input to construct a package. Thus, REED uses *convergent AONT (CAONT)* [38], which replaces the random key with a deterministic message-derived key to construct a package. This ensures that identical messages always lead to the same package.

REED augments CAONT to enable rekeying. Our insight is to achieve security by sacrificing a slight degradation of storage efficiency. The idea of REED is based on AONT-based secure deletion [42], which makes the entire package unrecoverable by securely removing a small part of a package. REED extends the idea to make it applicable for rekeying. Specifically, REED generates a CAONT package with the MLE key as an input, and encrypts a small part of the package, called the *stub* [42], with the file key. Thus, the entire package is now protected by both the file key and the MLE key. Since the stub size is small (e.g., 64 bytes, or 0.78%, for an 8KB chunk in our implementation), we mitigate the re-encryption overhead. In addition, we can still apply deduplication to the remaining large part of the package, called the *trimmed package*, so as to maintain storage efficiency.

In the following, we first design two rekeying-aware encryption schemes on a per-chunk basis (see Section IV-B), followed by enabling REED with dynamic access control on a per-file basis.

B. Encryption Schemes

We propose the basic and enhanced encryption schemes for REED. The basic scheme is more efficient, but is vulnerable to the leakage of an MLE key. On the other hand, the enhanced scheme protects against the leakage of an MLE key, while introducing an additional encryption step. In the following, we first explain the basics of AONT [47] and its variant CAONT [38], followed by how the basic and enhanced encryption schemes build on CAONT.

All-or-nothing transform (AONT): AONT [47] works as follows. It transforms a message M to a package denoted by (C, t), where C and t are called the *head* and *tail*, respectively. Specifically, it first selects a random encryption key K and generates a pseudo-random mask $\mathsf{G}(K) = \mathsf{E}(K, S)$, where $\mathsf{E}(\cdot)$ denotes a symmetric key encryption function (e.g., AES-256) and S is a publicly known block with the same size as M. It then computes $C = M \oplus \mathsf{G}(K)$, where '$\oplus$' is the XOR operator, and also computes $t = \mathsf{H}(C) \oplus K$, where $\mathsf{H}(\cdot)$ is the hash function (e.g., SHA-256). Note that the resulting package has a larger size than the original message M by the size of t. To recover the original message M, suppose that the whole package (C, t) is known. We first compute $K = \mathsf{H}(C) \oplus t$, followed by computing $M = C \oplus \mathsf{E}(K, S)$.

CAONT [38] follows the same paradigm of AONT, but replaces the random encryption key K by a deterministic cryptographic hash $h = \mathsf{H}(M)$ derived from the message M. This preserves content similarity of packages for identical messages, thereby making deduplication plausible. Another feature of CAONT is that it allows integrity checking without padding. Specifically, after the package is reverted, the integrity can be verified by computing the hash value of M and checking if it equals h.

Basic encryption: The basic encryption scheme leverages CAONT [38] to generate both the trimmed package and the stub, as shown in Figure 2. In particular, we make two modifications to CAONT. The first modification is to replace the cryptographic hash key in CAONT [38] by the corresponding MLE key K_M generated by the key manager. The rationale is that we use the MLE key to achieve security even for predictable chunks through server-aided MLE [17] (see Section II-A). However, we now cannot use the hash key for integrity checking as in CAONT. Thus, the second modification is to append a publicly known, fixed-size canary c to M [46] for CAONT, so that the integrity of M can be checked.

The basic encryption scheme is detailed as follows. We first concatenate an input chunk M with the canary c to form $(M||c)$, and compute the pseudo-random mask $\mathsf{G}(K_M) = \mathsf{E}(K_M, S)$, where K_M is the MLE key obtained from the key manager and S is the publicly known block with the same size of $(M||c)$. We compute the package head $C = (M||c) \oplus \mathsf{G}(K_M)$, where $\oplus$ is the XOR operator, and the package tail $t = K_M \oplus \mathsf{H}(C)$. We generate the stub by trimming the last few bytes (e.g., 64 bytes) from the package (C, t), and leave the remaining part as the trimmed package. Finally, we encrypt the stub with the file key. Reconstruction of a message works reversely, and we omit details here.

Enhanced encryption: One limitation of the basic encryption scheme is that it is vulnerable to the compromise of the MLE key. Specifically, an adversary can monitor the MLE keys generated by the key manager at a compromised client (see Section III-B). If an MLE key is revealed, the adversary can recover the pseudo-random mask and XOR the mask with the trimmed package to extract a majority part of the chunk.

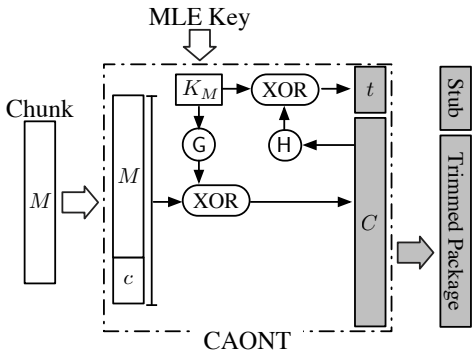


Fig. 2. Basic encryption of REED.

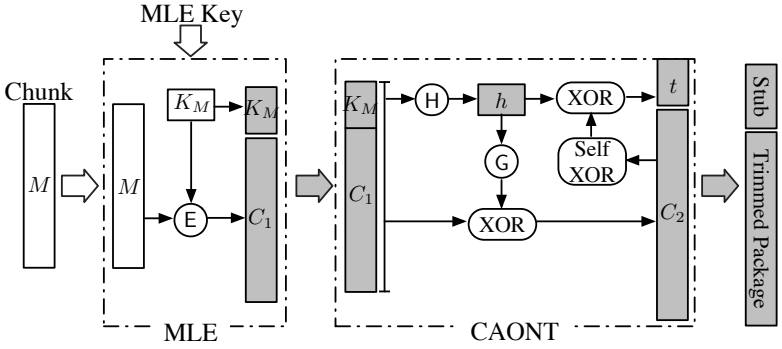


Fig. 3. Enhanced encryption of REED.

We propose the enhanced encryption scheme, which protects against the compromise of its MLE key. Figure 3 shows the workflow of the enhanced encryption, which first applies MLE to form a ciphertext, followed by applying CAONT [38] to the MLE ciphertext. The rationale is that even if an adversary obtains the MLE key, it still cannot recover original chunk because the MLE ciphertext is now protected by CAONT.

The enhanced encryption scheme is detailed as follows. First, we encrypt an input chunk M with the MLE key K_M as in traditional MLE, and obtain the ciphertext C_1. We then transform the concatenation $C_1||K_M$ based on the original CAONT [38]. We can now use the hash key $h = \mathsf{H}(C_1||K_M)$, instead of the MLE key used in the basic encryption scheme, to transform the package. This eliminates the security dependence on the MLE key. Formally, we compute the hash key $h = \mathsf{H}(C_1||K_M)$ and the pseudo-random mask $\mathsf{G}(h) = \mathsf{E}(h, S)$, where S is a publicly known block with the same size as $C_1||K_M$, and computes the package head $C_2 = (C_1||K_M) \oplus \mathsf{G}(h)$.

Since the hash key h allows integrity checking [38], we can generate the tail t with a *self-XOR* operation for efficiency [42], instead of using the cryptographic hash as in the basic encryption scheme (see Figure 2). Specifically, we evenly divide C_2 into a set of fixed-size pieces, each with the same size as h. We then XOR all the pieces as well as h to compute the tail t. Note that the self-XOR result cannot be predicted without knowing the entire content of C_2. Finally, we obtain the trimmed package and the stub from (C_2, t).

To reconstruct M, we first reconstruct (C_2, t) from the trimmed package and the stub. We evenly divide C_2 into fixed-size pieces, each with the same size as t, and compute h by XOR-ing the pieces and t. We then recover $C_1||K_M = C_2 \oplus \mathsf{G}(h)$, and check the integrity by comparing $\mathsf{H}(C_1||K_M)$ and h. We finally compute $M = \mathsf{D}(K_M, C_1)$, where $\mathsf{D}(\cdot)$ is the decryption function.

C. Dynamic Access Control

REED supports dynamic access control by associating each file with a *policy*, which provides a specification of which users are authorized or revoked to access the file. Our policy-based design builds on two well-known cryptographic primitives: *ciphertext policy attribute-based encryption (CP-ABE)* [19] and *key regression* [30]. REED integrates both primitives to generate the corresponding file key, as shown in Figure 4. We elaborate the details as follows.

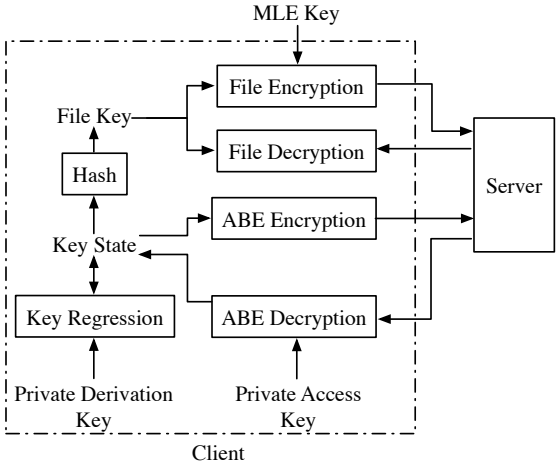


Fig. 4. REED generates a file key from the hash of a key state. The key state is derived from key regression. Its access is protected by CP-ABE.

Access control: REED defines policies based on CP-APE [19]. In CP-ABE, each policy is represented in the form of an *access tree*, in which each non-leaf node represents a Boolean gate (e.g., AND or OR), while each leaf node represents an *attribute* that defines or classifies some user property (e.g., the department that a user belongs to, the employee rank, the contract duration, etc.). Each user is given a private key that corresponds to a set of attributes. If a user's attributes satisfy the access tree, his private key can decrypt the ciphertext.

Our current design of REED treats each attribute as a unique identifier for each user. We issue each user with a CP-ABE private key, called the *private access key*, related to the identifier. We define the policy of each file as an access tree that connects the identifiers of all authorized users with an OR gate. Thus, any authorized user can decrypt the ciphertext, which we use to protect the file key (see the rekeying discussion below). Note that we can define more attributes and a more sophisticated access tree structure for better access control.

Rekeying: REED supports both lazy and active revocations for rekeying. In lazy revocation, REED builds on key regression [30], which is a serial key derivation scheme for generating different versions of keys. Specifically, key regression introduces

a sequence of *key states*, such that the current key state can derive the previous key states, but it cannot derive any future key state. Thus, an authorized user can access all previous key states, and the corresponding files, by using only the current key state; meanwhile, a user revoked from the current key state cannot access any new file that is protected by a future key state. Key regression is designed for lazy revocation (which defers file re-encryption until the next update), since it allows an authorized user to access the not-yet-updated files through the previous key states.

REED implements lazy revocation using the RSA-based key regression scheme [30]. We assign each user with a unique pair of public-private keys called the *derivation keys*, such that the private derivation key is used to generate new key states for the files owned by the user, while the public derivation key is used to derive the previous key states. The file key will be obtained by generating a cryptographic hash of the current key state. Each key state refers to a policy, and it will be encrypted by CP-ABE associated with the authorized users. In other words, any authorized user can retrieve the current key state, and hence the file key, with his private access key.

REED implements active revocation following the same paradigm as in lazy revocation, except that the files affected by active revocation are immediately re-encrypted with the new file key.

D. Operations

We now summarize the interactions among a client, a server, the key manager, and the storage backend in REED operations. We focus on three basic operations, including upload, download, and rekeying.

Upload: To upload a file F, the client first picks a random key state S_F and hashes it into a (symmetric) file key κ_F. It splits F into a set of chunks $\{M\}$, computes their fingerprints, and runs the OPRF protocol [17] with the key manager to obtain the MLE keys $\{K_M\}$ of these chunks. For each M, it uses K_M to transform a chunk into a trimmed package and a stub, using either the basic or enhanced encryption scheme (see Section IV-B). The stub will be further encrypted by the file key κ_F. In addition, the client generates a *file recipe*, which includes the file information such as the file pathname, file size, and the total number of chunks. Furthermore, the client encrypts S_F using CP-ABE based on the policy of the file. Finally, the client uploads the following information to the REED server[1]: (i) the trimmed packages and encrypted stubs for all chunks, (ii) file recipe, and (iii) the encrypted key state S_F and the metadata that includes the policy information. The server performs deduplication on the received trimmed packages. All information will be stored at the storage backend.

Download: To download a file F, the client first retrieves the encrypted key state S_F and decrypts it with the private access key. It then hashes S_F to recover the file key κ_F. In addition, it downloads all trimmed packages and encrypted stubs from the storage backend, with the help of the REED server and the file recipe. It decrypts the stubs via κ_F, and finally reconstructs all chunks for F. Note that if the client detects any tampered chunk, the reconstruction operation will abort.

[1]Note that we do not need to upload MLE keys, as they are not used in decryption (see Section IV-B).

Rekeying: To rekey F with new access privileges, the client (on behalf of the owner of F) retrieves S_F and its metadata, and decrypts S_F with the private access key. It then generates a new key state S'_F based on key regression (see Section IV-C). It encrypts S'_F via CP-ABE based on a new policy (e.g., with a new group of users). It finally uploads the encrypted S'_F as well as its metadata that describes the new policy information. For active revocation, the client also downloads the stubs of F, re-encrypts them with a new file key obtained by hashing S'_F, and finally uploads the re-encrypted stubs.

Discussion: Our current design and implementation focus on the encryption and rekeying of file chunks, while we do not address those of file metadata. We can obfuscate sensitive metadata information, such as the file pathname, by encoding it via a salted hash function. In addition, we perform rekeying on a per-file basis, yet we can generalize rekeying for a group of files. We pose these issues as future work.

E. Security Analysis

We now analyze the security of REED based on our security goals (see Section III-B).

Confidentiality: We show how REED achieves confidentiality at three levels. First, an adversary can access all trimmed packages, encrypted stubs, and encrypted key states from a compromised server. Since the adversary cannot compromise any private access key and private derivation key, all trimmed packages and encrypted stubs cannot be reverted. Thus, REED achieves the same level of confidentiality like DupLESS [17] (see Section II-A).

Second, an adversary can collude with revoked or unauthorized clients, through which the adversary can learn a set of private derivation keys and private access keys. Due to the protection of CP-ABE and key regression, these compromised private keys cannot be used to decrypt the file key ciphertexts beyond their access scopes. Without proper file keys, the adversary cannot infer anything about the underlying chunks. One special note is that a client may keep the MLE key (in basic encryption) or the hash key (in enhanced encryption) of a chunk in CAONT (see Figures 2 and 3, respectively) to make the chunk accessible even after being revoked. However, if the chunk is updated, the revoked client cannot learn any information from the updated chunk because CAONT will use a new MLE key or hash key to transform the updated chunk, making the old one useless.

Finally, an adversary can monitor a subset of clients and identify the MLE keys requested by them. The enhanced encryption scheme of REED ensures confidentiality for unpredictable chunks, even though the victim clients are authorized to access these chunks. Specifically, the enhanced encryption scheme builds an additional security layer with the file key. Although the MLE key is compromised, as long as the file key remains secure, the adversary cannot access the stub and infer any useful information.

Integrity: Both the basic and enhanced encryption schemes of REED ensure chunk-level integrity, such that any modification of the trimmed package or the stub of a chunk can be detected. In the basic encryption scheme, the MLE key can be reverted as $K_M = \mathsf{H}(C) \oplus t$ (see Section IV-B). Since $\mathsf{H}(C)$ depends

on every bit of C [56], the modification of any part of the package will lead to an incorrect K_M. Thus, the client can easily detect the modification by checking the canary padded with the reverted chunk.

Using similar reasonings, the enhanced encryption scheme also ensures the integrity of a chunk, such that a client performs integrity checking by comparing if $\mathsf{H}(C_1||K_M)$ equals h (see Section IV-B). One special note regarding the enhanced scheme is that its use of the self-XOR operation may return a correct hash key h even if the package is tampered. For example, an intelligent adversary can divide C_2 into fixed-size pieces and flip the same bit position for an even number of the pieces. On the other hand, a tampered package will be reverted to a wrong input even with the correct hash key, and its integrity violation can be caught by comparing it with h.

V. Implementation

We implement a REED prototype in C++. We extend the open-source system CDStore [38] to support our rekeying design. We also use OpenSSL 1.0.2a [3] and CP-ABE toolkit [1] to implement the cryptographic operations in REED.

A. Entities

In the following, we describe the implementation details of each entity in the REED architecture (see Figure 1).

Client: A client divides an input file into chunks in file uploads. We support both fixed-size and variable-size chunking schemes. We implement variable-size chunking using Rabin fingerprinting [44], which takes the minimum, maximum, and average chunk sizes as inputs. We fix the minimum and maximum chunk sizes at 2KB and 16KB, respectively, and vary the average chunk size in our evaluation. In file downloads, the client reassembles collected chunks into the original file.

The client implements both basic and enhanced encryption schemes (and the corresponding decryption schemes). It also implements the RSA-based key regression scheme [30] for generating new key states during rekeying, and protects each key state using CP-ABE (via the CP-ABE toolkit [1]). In chunk encryption, the client transforms a chunk into a trimmed package and a stub, in which we configure the stub size as 64 bytes for each chunk to resist brute-force attacks on the stub yet preserving storage efficiency. We write the stubs of all the chunks of the same file into a separate *stub file*, which we encrypt with the corresponding file key. For both encryption schemes, we set the fixed-size canary c to be 32 bytes of zeroes for integrity checking.

Key manager: For key management, a client communicates with the key manager via an SSL/TLS-based authentication channel. The key generation follows the OPRF protocol of DupLESS [17] to "blind" MLE key generation: (i) The key manager is configured with a system-wide public/private key pair, which we now generate based on 1024-bit RSA; (ii) The client sends a blinded fingerprint of each chunk to the key manager; (iii) The key manager computes an RSA signature on the blinded fingerprint; and (iv) The client computes an RSA signature verification and unblinds the result, which is hashed to form the MLE key. Other approaches, such as blinded BLS signatures [23], can be used to implement blinded MLE key generation.

Server: A server can receive file data from multiple clients. It performs deduplication on the trimmed packages received from a client and checks if identical trimmed packages have already been stored (by the same client or a different client), and only stores unique trimmed packages in the storage backend. It also keeps track of the deduplication metadata, including the fingerprints of all trimmed packages for deduplication, as well as the references to all trimmed packages and file recipes in the storage backend for file retrieval.

Storage backend: We separate the storage of key information and file data for better management. Specifically, we create two stores at the storage backend: (i) the *data store*, which stores the file data such as file recipes, trimmed packages, stub files, and all related file metadata, and (ii) the *key store*, which stores the key information such as encrypted key states.

B. Optimization

Our REED implementation leverages standard optimization techniques for better performance.

Batching: We batch small requests to mitigate network and I/O overheads. We note that if a client sends individual per-chunk MLE key generation requests to the key manager, there will be significant round-trip overheads, especially if we handle many small-size chunks. We batch multiple per-chunk key generation requests to reduce round-trip overheads. In addition, during file uploads, a client batches multiple trimmed packages in an in-memory buffer (currently we set its size as 4MB), and sends the batched packages to the server when the buffer is full. Furthermore, the server batches the unique trimmed packages after deduplication into 4MB units before storing them in the storage backend, so as to mitigate I/O overheads.

Caching: We note that MLE key generation leads to substantial computational overhead, mainly because the key manager uses the public-key-based OPRF protocol for key generation [17]. The computational overhead is more prominent when we generate MLE keys for small-size chunks. To reduce the computational overhead of the key manager, our observation is that the adjacent uploads of a client often share high content similarity. For example, when the client uploads weekly backup snapshots of the same file system, the backup snapshots may be very similar in content if there are only small modifications to the file system. In this case, the client may reuse the MLE keys for a large proportion of identical chunks for the previous upload. Specifically, the client maintains a least-recently-used cache (512MB by default) in memory for holding the most recently generated MLE keys. It looks up for the cached MLE keys before sending requests to the key manager. Note that the caching approach introduces security risks against the MLE keys, as the adversary can access the cached MLE keys after compromising a client. Fortunately, our enhanced encryption scheme resists the risk as it protects against the leakage of MLE keys (see Section IV-B).

Parallelization: We exploit parallelization to improve performance. First, each client parallelizes encryption and decryption via multi-threading: it dispatches chunks to multiple threads, each of which performs encryption and decryption on a subset of chunks. In addition, we run separate servers to manage both data store and key store in the storage backend, and use multiple servers to manage the data store. A client divides each

trimmed package and stub file into multiple pieces and sends each piece to a different server, which now processes only smaller quantities of data. Furthermore, each client creates multiple threads to connect to multiple servers, while each server creates multiple threads to accept connections from multiple clients.

VI. Evaluation

We evaluate REED on a LAN testbed composed of multiple machines, each of which is equipped with a quad-core 3.4GHz Intel Core i5-3570, 7200RPM SATA hard disk, and 8GB RAM, and installed with 64-bit Ubuntu 12.04.2 LTS. All machines are connected via a 1Gb/s switch.

Our default setting of REED is as follows. We run one REED client, one key manager, and five REED servers in different machines. We use multiple REED servers for improved scalability (see Section V-B). In particular, four of the five servers manage the data store, and the remaining one server manages the key store. In practice, both the data store and the key store should be deployed in a shared storage backend (e.g., cloud storage); however, to remove the I/O overhead of accessing the shared storage backend in our evaluation, we simply have each server store information in its local hard disk. In addition to the default setting, we describe additional specific settings in each experiment, and also consider the case where multiple clients are involved. We compile our programs with g++ 4.8.1 with the -O3 option. For performance tests, we present the average results over 10 runs. We do not include the variance results in our plots, as they are generally very small in our evaluation. In the following, we consider a synthetic dataset and a real-world dataset to drive our evaluation.

A. Synthetic Data

We evaluate different REED operations through synthetic data. Specifically, we generate a 2GB file of synthetic data with globally unique chunks (i.e., the chunks have no duplicate content). Before each experiment, we load the synthetic data into memory to avoid generating any disk I/O overhead.

Experiment A.1 (MLE key generation performance): We first measure the performance in MLE key generation between the client and the key manager based on our default setting. The client first creates chunks of the input 2GB file using variable-size chunking based on Rabin fingerprinting with a specified average chunk size. It then requests for MLE keys for these chunks from the key manager. We measure the *MLE key generation speed*, defined as the ratio of the file size (i.e., 2GB) to the total time starting from when the client sends the blinded fingerprints to the key manager for MLE keys until the client obtains the MLE keys of all chunks from the key manager. We focus on evaluating the impact of two parameters: (i) the average chunk size and (ii) the batch size of key generation requests (i.e., the number of per-chunk key generation requests in a batch) (see Section V-B).

Figure 5(a) first shows the MLE key generation speed versus the average chunk size, in which we fix the batch size as 256 per-chunk key generation requests. We observe that the speed increases with the average chunk size, mainly because fewer chunks need to be processed. For example, when the average chunk size is 16KB, the key generation speed reaches 17.64MB/s.

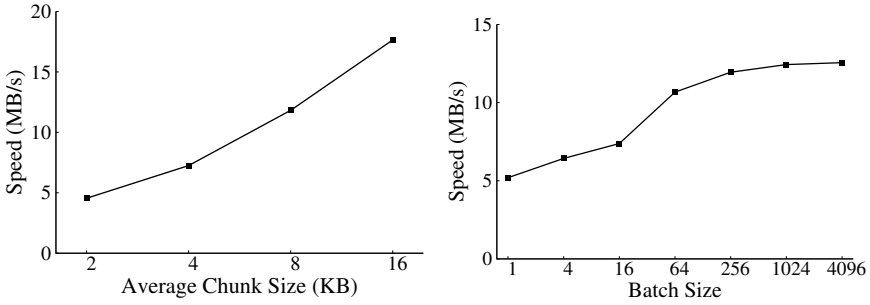
Figure 5(b) shows the MLE key generation speed versus the batch size, in which we fix the average chunk size as 8KB. A larger batch size implies less round-trip overhead. We observe that when the batch size goes beyond 256, the key generation speed becomes steady (at about 12.5MB/s), since the key manager is now saturated by key generation requests and the speed is bounded by the computation of the key manager.

Experiment A.2 (Encryption performance): We measure the performance of both basic and enhanced encryption schemes. Suppose that the client has created chunks with variable-size chunking and obtained MLE keys from the key manager. We also exploit multi-threading (see Section V-B) by configuring the encryption module with two threads. We do not consider more threads, mainly because our machines only have four CPU cores and more threads will lead to contention. In addition, our results indicate that two threads are sufficient for achieving the required performance (see below). Here, we measure the *encryption speed*, defined as the ratio of the file size (i.e., 2GB) to the total time of encrypting all chunks into trimmed packages and stubs.

Figure 6 shows the speeds of both basic and enhanced encryption schemes versus the average chunk size. The throughput of both encryption schemes increases with the average chunk size, mainly because fewer chunks need to be processed. The basic scheme is faster than the enhanced scheme, as the enhanced scheme introduces an additional encryption (see Section IV-B). For example, for the average chunk size 8KB, the basic scheme has 203MB/s, 24% faster than 155MB/s in the enhanced scheme. We observe that the encryption speeds of both schemes are higher than the network speed (which is now 1Gb/s), and hence the encryption speed is not the performance bottleneck in REED. We further justify this claim in Experiment A.3.

Experiment A.3 (Upload and download performance): We now measure the upload and download performance of REED. We first consider the case of a single client. We enable all optimizations in the client (see Section V-B), including: (i) setting the batch size as 256 per-chunk key generation requests, (ii) turning on the key cache with size 512MB, (iii) using two threads for encryption and decryption. The client first uploads a 2GB file of unique data, followed by the same 2GB file with identical content, and finally downloads the 2GB file. We measure the *upload speed* as the ratio of the file size to the total time of sending all file data to the servers (including the chunking, key generation, encryption, and data transfer), and the *download speed* as the ratio of the file size to the total time starting from when the client issues a download request until all original data is recovered.

Figure 7(a) shows the upload speeds under both encryption schemes versus the average chunk size. For the first upload, the upload speeds of both basic and enhanced encryption schemes are relatively low, ranging from around 4MB/s (for the 2KB chunk size) to 17MB/s (for the 16KB chunk size). The upload speed is mainly bounded by the MLE key generation speed (see Figure 5(a)). In the second upload, all the MLE keys have been cached, and hence the upload speeds of



(a) Varying chunk size (b) Varying batch size.

Fig. 5. Experiment A.1 (MLE key generation performance).

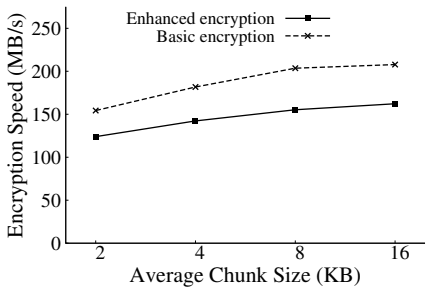
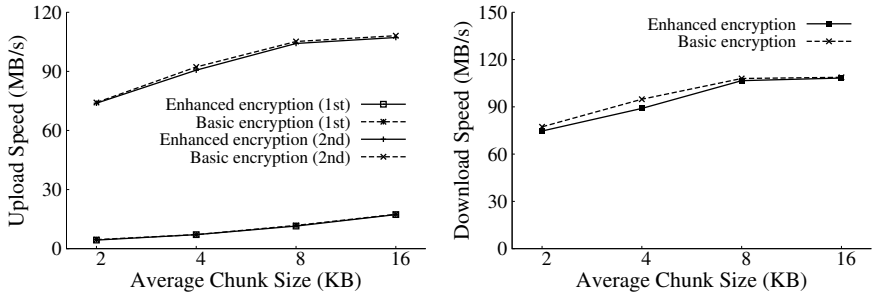


Fig. 6. Experiment A.2 (Encryption performance).



(a) Upload speed (b) Download speed (c) Aggregate upload speed

Fig. 7. Experiment A.3 (Upload and download performance).

both encryption schemes increase significantly, for example, to 108.1MB/s and 107.2MB/s for the basic and enhanced schemes, respectively, for the 16KB chunk size. We find that the effective network speed in our LAN testbed is around 116MB/s, which is approximately reached by both encryption schemes. Note that both encryption schemes have only minor performance differences.

Figure 7(b) shows the download speeds under both encryption schemes versus the average chunk size. When the average chunk size goes beyond 8KB, the download speeds of both encryption schemes (e.g., 108.0MB/s for basic encryption and 106.6MB/s for enhanced encryption) approximate the effective network speed.

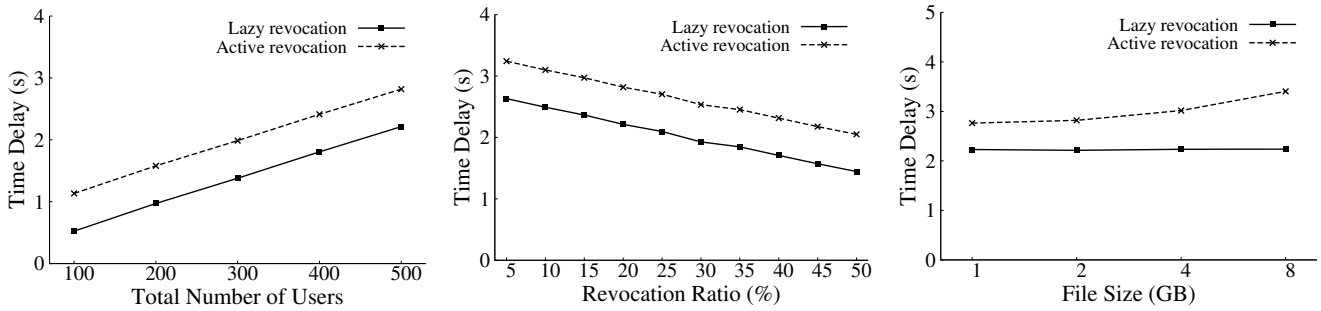
We consider the case where multiple REED clients are used. We vary the number of clients from one to eight, and each client runs on a different machine. Here, we focus on the aggregate upload performance under the enhanced encryption scheme. Each client first uploads a 2GB file of unique data, followed by uploading the same 2GB file with identical content. Each client enables the same optimization setting as the above single-client case. All clients perform uploads simultaneously. We measure the *aggregate upload speed*, defined as the ratio of the total amount of file data (i.e., 2GB times the number of clients) to the total time when all uploads are finished.

Figure 7(c) shows the aggregate upload speed versus the number of clients. We see that the speed increases with the number of clients. The performance of the first aggregate upload is bounded by the MLE key generation, while that of the second aggregate upload takes advantage of the cached MLE keys and now becomes bounded by the network bandwidth. When there are eight clients, the second aggregate upload speed reaches 374.9MB/s.

Experiment A.4 (Rekeying performance): We measure the rekeying performance in both lazy and active revocation schemes. Recall that the rekeying operation of REED requires a CP-ABE decryption with the original policy and another CP-ABE encryption with a new policy. REED treats each policy as an access tree with an OR gate connecting all the authorized user identifiers (see Section IV-C). This implies that the CP-ABE decryption time is constant [19], while its encryption time grows with the number of authorized users in the new policy. Thus, we focus on evaluating the impact of three parameters in the rekeying operation: (i) *total number of users*, i.e., the number of authorized users in the original policy; (ii) *revocation ratio*, the percentage of the number of users to be revoked and removed from the access tree; and (iii) *file size*, the size of the rekeyed file. We measure the *rekeying delay*, defined as the total time of performing all rekeying steps including: downloading and decrypting a key state, deriving a new key state, encrypting and uploading the new key state, and re-encrypting the stub file (for active revocation only).

Figure 8(a) shows the rekeying delay versus the total number of users, while we fix the rekeyed file size at 2GB and the revocation ratio at 20%. The rekeying delays of both revocation schemes increase with the total number of users, mainly because the CP-ABE encryption overhead increases with a larger access tree. Nevertheless, the rekeying delays are within three seconds in both revocation schemes. In particular, lazy revocation is faster than active revocation by about 0.6s, as it defers re-encryption process to the next file update.

Figure 8(b) shows the rekeying delay versus the revocation



(a) Varying the total number of users (b) Varying the revocation ratio (c) Varying the file size

Fig. 8. Experiment A.4 (Rekeying performance).

ratio, while we fix the rekeyed file size at 2GB and the total number of users at 500. With a larger revocation ratio, the new policy has fewer authorized users, thereby reducing the revocation time. When the revocation ratio is 50%, the rekeying delays of the lazy and active revocation schemes are 1.44s and 2s, respectively.

Figure 8(c) shows the rekeying delay versus the size of the rekeyed file, while we fix the total number of users at 500 and the revocation ratio at 20%. The rekeyed file size has no impact on lazy revocation, in which the rekeying delay is kept at 2.25s. For active revocation, as the file size increases, it spends more time for transferring and re-encrypting the stub file. Thus, the rekeying delay increases, for example, to 3.4s for an 8GB file. Nevertheless, if we compare the rekeying delay of active revocation with the time of transferring a whole file in the network (e.g., at least 64s in a 1Gb/s network), the rekeying delay is insignificant. Thus, the rekeying operation in REED is lightweight in general.

B. Real-world Data

We now consider a real-world public dataset collected by the File systems and Storage Lab (FSL) at Stony Brook University [6]. The original FSL dataset contains daily backups of the home directories of nine users in a shared file system, and it lasts from 2011 to 2014. We focus on the `Fslhomes` dataset in 2013, which comprises 147 daily snapshots from January 22 to June 17, 2013. Each snapshot represents a daily backup, represented by a collection of 48-bit fingerprints of variable-size chunks with an average 8KB chunk size. The dataset we consider accounts for a total of 56.20TB of pre-deduplicated data.

Experiment B.1 (Storage overhead): We first measure the storage overhead due to REED. Our goal is to show that REED still maintains storage efficiency via deduplication, even though it can only deduplicate part of a chunk (i.e., trimmed package). We define three types of data: (i) *logical data*, the original data before any encryption or deduplication; (ii) *stub data*, the encrypted stub files being stored; (iii) *physical data*, the trimmed packages being stored after deduplication. We aggregate the data from all users and measure the total size of each data type. Here, we do not consider the metadata overhead, which is much less than that of the file data.

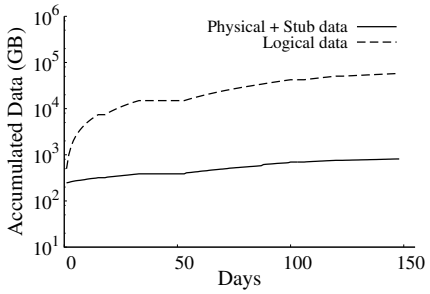
Figure 9(a) first shows the cumulative data sizes over the number of days of storing daily backups of all users. Each daily backup contains 290-680GB of logical data for all users, yet the physical and stub data that REED actually stores after deduplication accounts for only 5.52GB per day on average. After 147 days, there is a total of 57,548GB of logical data, and REED generates only 812GB of physical and stub data after deduplication. It achieves a total saving of 98.6%. This shows that we still maintain high storage efficiency through deduplication.

Figure 9(b) compares the cumulative sizes of physical and stub data after deduplication. The cumulative size of stub data increases over days. After 147 days, there is 431.89GB of physical data due to the unique trimmed packages, and there is 380.14GB of stub data. Note that the stub data cannot be deduplicated as it is encrypted by a renewable file key. Nevertheless, deduplication effectively reduces the overall storage space according to Figure 9(a).

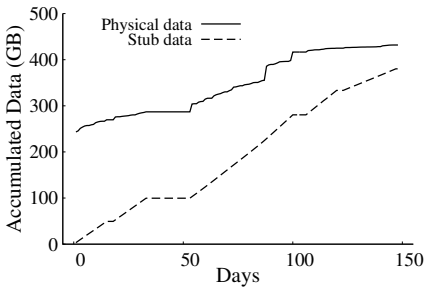
Experiment B.2 (Trace-driven upload and download performance): We evaluate upload and download speeds of a single REED client using the real-world dataset, as opposed to synthetic dataset in Experiment A.3. Since the dataset just includes chunk fingerprints and chunk sizes, we reconstruct a chunk by repeatedly writing its fingerprint to a spare chunk until reaching the specified chunk size; this ensures that the same (distinct) fingerprint returns the same (distinct) chunk. The reconstructed chunk is treated as the output of chunking module of the REED client. Thus, we do not include the chunking time in this experiment.

The client uploads all daily backups (on behalf of all users), followed by downloading them. Due to the large dataset, we only run part of the dataset to reduce the evaluation time. Specifically, we choose seven consecutive daily backups from March 19 to March 25, 2013 for the nine users, covering a total of 3.64TB of data before deduplication. We use the same optimization setting as in Experiment A.3; in particular, we enable the key cache in the client since the dataset has high content similarity. We use the following order of the upload sequence: we upload the backups of the first user day by day through the client, followed by the second user, and so on. Before uploading the backups of each user, we clear the key cache. This ensures that different users will not share the same key cache of the client.

Figure 10 shows the upload and download speeds over days. The upload speed for the first day is around 13.1MB/s, since all users need to contact the key manager and generate MLE keys for most of the chunks. For the subsequent backups, the upload speed increases significantly, as most of the MLE



(a) Before and after deduplication



(b) Physical and stub data

Fig. 9. Experiment B.1 (Storage overhead).

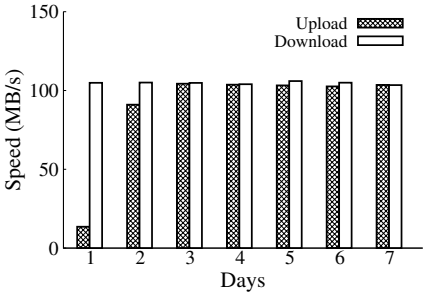


Fig. 10. Experiment B.2 (Trace-driven upload and download performance).

keys have been cached in the client. Note that since the dataset has a high deduplication rate, the upload speeds for the subsequent backups are similar to that for the synthetic data upload with identical content (see Figure 7(a)), and reach around 105MB/s. However, the trace-driven download speed is slightly lower than the synthetic one (see Figure 7(b)). The reason is deduplication introduces *chunk fragmentation* for subsequent backups [39], and the download speed will gradually degrade. We do not address the chunk fragmentation problem, which is beyond the scope of the work.

VII. Related Work

Encrypted deduplication storage: Section II reviews MLE [18] and DupLESS [17], which address the theoretical and applied aspects of encrypted deduplication storage, respectively. Bellare *et al.* [18] propose a theoretical framework of MLE, and provide formal definitions of privacy and tag consistency. The follow-up studies [7], [16] further examine message correlation and parameter dependency of MLE.

On the applied side, convergent encryption (CE) [28] has been implemented and experimented in various storage systems (e.g., [9], [10], [24], [48], [52], [57]). DupLESS [17] implements server-aided MLE. Duan [29] improves the robustness of key management in DupLESS via threshold signature [49]. Zheng *et al.* [58] propose a layer-level strategy specifically for video deduplication. Liu *et al.* [40] propose a password-authenticated key exchange protocol for MLE key generation. ClearBox [11] enables clients to verify the effective storage space that their data occupies after deduplication. CDStore [38] realizes CE in existing secret sharing algorithms by replacing the embedded random seed with a message-derived hash to construct shares. REED focuses on the applied aspect, and complements the above designs by enabling rekeying in encrypted deduplication storage.

Rekeying: Abdalla *et al.* [8] rigorously analyze key-derivation methods, in which a sequence of subkeys is derived from a shared master key so as to extend the lifetime of the master key for secure communication. Follow-up studies examine key derivation (in either key rotation or key regression) in content distribution networks [14], [30], [36] and cloud storage [43]. A recent work [55] examines ciphertext re-encryption using an approach similar to REED, in that it performs AONT on files and updates a small piece from the AONT package, yet it does not consider deduplication and has no prototype that demonstrates the applicability. REED differs from the above approaches by addressing the rekeying problem in encrypted deduplication storage. REED also uses the key regression scheme [30] in key derivation to enable lazy revocation.

REED is related to secure deletion (see detailed surveys [27], [45]), which ensures that securely deleted data is permanently inaccessible by anyone. Secure deletion can be achieved through cryptographic deletion (e.g., [22], [42]), which securely erases keys in order to make encrypted data unrecoverable. REED builds on the AONT-based cryptographic deletion [42] and preserves content similarity for deduplication. It further allows lightweight dynamic access control.

Access control: Cryptographic primitives have been proposed for enabling access control on encrypted storage, such as broadcast encryption [21], proxy re-encryption [13], and ABE [32]. REED builds on CP-ABE [19] to implement fine-grained access control for encrypted deduplication storage.

VIII. Conclusion

We present REED, an encrypted deduplication storage system that aims for secure and lightweight rekeying. The core rekeying design of REED is to renew a key of a deterministic all-or-nothing-transform (AONT) package. We propose two encryption schemes for REED: the basic scheme has higher encryption performance, while the enhanced scheme is resilient against key leakage. We further extend REED with dynamic access control by integrating both CP-ABE and key regression primitives. We show the confidentiality and integrity properties of REED under our security definitions. We implement a REED prototype, and conduct trace-driven evaluation in a LAN testbed to demonstrate its performance and storage effectiveness. The source code of our REED prototype is available for download at **http://ansrlab.cse.cuhk.edu.hk/software/reed**.

Acknowledgments

This work was supported in part by GRF CUHK413813 from HKRGC, Cisco University Research Program Fund (CG#593822) from Silicon Valley Community Foundation, Natural Science Foundation of Guangdong Province for Distinguished Young Scholars (2014A030306020), National Natural Science Foundation of China (61472091, 61572115), Major Program for the Fundamental Research of Sichuan (2016JY0007), and Distinguished Young Scholars Fund of Department of Education of Guangdong Province (Yq2013126).

REFERENCES

[1] "CP-ABE toolkit," http://acsc.cs.utexas.edu/cpabe/.

[2] "Google Genomics," https://cloud.google.com/genomics/.

[3] "OpenSSL," https://www.openssl.org.

[4] "Netapp deduplication helps duke institute for genome sciences and policy reduce storage requirements for genomic information by 83 percent," http://www.netapp.com/us/company/news/press-releases/news-rel-20081008.aspx, 2008.

[5] "Architecting for genomic data security and compliance in AWS," 2014.

[6] "FSL traces and snapshots public archive," http://tracer.filesystems.org/, 2014.

[7] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev, "Message-locked encryption for lock-dependent messages," in *Proc. of CRYPTO*, 2013.

[8] M. Abdalla and M. Bellare, "Increasing the lifetime of a key: A comparative analysis of the security of re-keying techniques," in *Proc. of ASIACRYPT*, 2000.

[9] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. of USENIX OSDI*, 2002.

[10] P. Anderson and L. Zhang, "Fast and secure laptop backups with encrypted de-duplication," in *Proc. of USENIX LISA*, 2010.

[11] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud," in *Proc. of ACM CCS*, 2015.

[12] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of ACM CCS*, 2007.

[13] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 1–30, Feb. 2006.

[14] M. Backes, C. Cachin, and A. Oprea, "Secure key-updating for lazy revocation," in *Proc. of ESORICS*, 2006.

[15] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "NIST Special Publication 800-57 recommendation for key management," National Institute of Standards & Technology, Tech. Rep., July 2012.

[16] M. Bellare and S. Keelveedhi, "Interactive message-locked encryption and secure deduplication," in *Proc. of PKC*, 2015.

[17] M. Bellare, S. Keelveedhi, and T. Ristenpart, "DupLESS: Server-aided encryption for deduplicated storage," in *Proc. of USENIX Security*, 2013.

[18] ——, "Message-locked encryption and secure deduplication," in *Proc. of EUROCRYPT*, 2013.

[19] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *IEEE S&P*, 2007.

[20] J. Black, "Compare-by-hash: a reasoned analysis," in *Proc. of USENIX ATC*, 2006.

[21] D. Boneh, C. Gentry, and B. Waters, "Collusion resistant broadcast encryption with short ciphertexts and private keys," in *Proc. of CRYPTO*, 2005.

[22] D. Boneh and R. Lipton, "A revocable backup system," in *Proc. of USENIX Security*, 1996.

[23] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Proc. of ASIACRYPT*, 2001.

[24] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: Making backup cheap and easy," in *Proc. of USENIX OSDI*, 2002.

[25] D. Csaplar, "Building business resillience through active archiving," 2011.

[26] Debian Security Advisory, "DSA-1571-1 openssl – predictable random number generator," https://www.debian.org/security/2008/dsa-1571, May 2008.

[27] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Comput. Surv.*, vol. 43, no. 1, pp. 2:1–2:37, Dec. 2010.

[28] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proc. of IEEE ICDCS*, 2002.

[29] Y. Duan, "Distributed key generation for encrypted deduplication: Achieving the strongest privacy," in *Proc. of ACM CCSW*, 2014.

[30] K. Fu, S. Kamara, and T. Kohno, "Key regression: Enabling efficient key distribution for secure distributed storage," in *Proc. of NDSS*, 2006.

[31] S. Goldwasser and M. Bellare, "Lecture notes on cryptography," https://cseweb.ucsd.edu/~mihir/papers/gb.html, July 2008.

[32] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proc. of ACM CCS*, 2006.

[33] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proc. of ACM CCS*, 2011.

[34] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[35] A. Juels and B. S. Kaliski, Jr., "PORs: Proofs of retrievability for large files," in *Proc. of ACM CCS*, 2007.

[36] M. Kallahall, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proc. of USENIX FAST*, 2002.

[37] D. Kaminsky, "These are not the certs you're looking for," http://dankaminsky.com/2011/08/31/notnotar/, Aug 2011.

[38] M. Li, C. Qin, and P. P. C. Lee, "CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal," in *Proc. of USENIX ATC*, 2015.

[39] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. of USENIX FAST*, 2013.

[40] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers," in *Proc. of ACM CCS*, 2015.

[41] National Institutes of Health, "NIH security best practices for controlled-access data subject to the NIH genomic data sharing policy," 2015.

[42] Z. N. J. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. D. Rubin, "Secure deletion for a versioning file system," in *Proc. of USENIX FAST*, 2005.

[43] K. P. Puttaswamy, C. Kruegel, and B. Y. Zhao, "Silverline: toward data confidentiality in storage-intensive cloud applications," in *Proc. of ACM SoCC*, 2011.

[44] M. O. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technology, Harvard University. Tech. Report TR-CSE-03-01, 1981.

[45] J. Reardon, D. Basin, and S. Capkun, "SoK: Secure data deletion," in *Proc. of IEEE S&P*, 2013.

[46] J. K. Resch and J. S. Plank, "AONT-RS: Blending security and performance in dispersed storage systems," in *Proc. of USENIX FAST*, 2011.

[47] R. L. Rivest, "All-or-nothing encryption and the package transform," in *Proc. of FSE*, 1997.

[48] P. Shah and W. So, "Lamassu: Storage-efficient host-side encryption," in *Proc. of USENIX ATC*, 2015.

[49] V. Shoup, "Practical threshold signatures," in *Proc. of EUROCRYPT*, 2000.

[50] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger, "Md5 considered harmful today," http://www.win.tue.nl/hashclash/rogue-ca/, Dec 2008.

[51] L. D. Stein, "The case for cloud computing in genome informatics," *Genome Biology*, 2010.

[52] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in *Proc. of ACM StorageSS*, 2008.

[53] U.S. Computer Emergency Readiness Team, "OpenSSL 'heartbleed' vulnerability (CVE-2014-0160)," https://www.us-cert.gov/ncas/alerts/TA14-098A, April 2014.

[54] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *Proc. of USENIX FAST*, 2012.

[55] D. Watanabe and M. Yoshino, "Key update mechanism for network storage of encrypted data," in *Proc. of IEEE CloudCom*, 2013.

[56] A. F. Webster and S. E. Tavares, "On the design of S-boxes," in *Proc. of CRYPTO*, 1985.

[57] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: The least-authority filesystem," in *Proc. of ACM StorageSS*, 2008.

[58] Y. Zheng, X. Yuan, X. Wang, J. Jiang, C. Wang, and X. Gui, "Enabling encrypted cloud media center with secure deduplication," in *Proc. of ACM ASIACCS*, 2015.