Secure Deduplication with Efficient and Reliable Convergent Key Management

Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick P.C. Lee, and Wenjing Lou

Abstract—Data deduplication is a technique for eliminating duplicate copies of data, and has been widely used in cloud storage to reduce storage space and upload bandwidth. Promising as it is, an arising challenge is to perform secure deduplication in cloud storage. Although convergent encryption has been extensively adopted for secure deduplication, a critical issue of making convergent encryption practical is to efficiently and reliably manage a huge number of convergent keys. This paper makes the first attempt to formally address the problem of achieving efficient and reliable key management in secure deduplication. We first introduce a baseline approach in which each user holds an independent master key for encrypting the convergent keys and outsourcing them to the cloud. However, such a baseline key management scheme generates an enormous number of keys with the increasing number of users and requires users to dedicatedly protect the master keys. To this end, we propose *Dekey*, a new construction in which users do not need to manage any keys on their own but instead securely distribute the convergent key shares across multiple servers. Security analysis demonstrates that Dekey is secure in terms of the definitions specified in the proposed security model. As a proof of concept, we implement Dekey using the Ramp secret sharing scheme and demonstrate that Dekey incurs limited overhead in realistic environments.

Index Terms—Deduplication, proof of ownership, convergent encryption, key management

1 Introduction

The advent of cloud storage motivates enterprises and organizations to outsource data storage to third-party cloud providers, as evidenced by many real-life case studies [3]. One critical challenge of today's cloud storage services is the management of the ever-increasing volume of data. According to the analysis report of IDC, the volume of data in the wild is expected to reach 40 trillion gigabytes in 2020 [9]. To make data management scalable, deduplication has been a well-known technique to reduce storage space and upload bandwidth in cloud storage. Instead of keeping multiple data copies with the same content, deduplication eliminates redundant data by keeping only one physical copy and referring other redundant data to that copy. Each such copy can be defined based on different granularities: it may refer to either a whole file (i.e., file-level deduplication), or a more fine-grained fixed-size or variable-size data block (i.e., block-level deduplication). Today's commercial cloud storage services, such as Dropbox, Mozy, and Memopal, have been applying deduplication to user data to save maintenance cost [12].

From a user's perspective, data outsourcing raises security and privacy concerns. We must trust third-party cloud providers to properly enforce confidentiality, integrity checking, and access control mechanisms against any insider and outsider attacks. However, deduplication, while improving storage and bandwidth efficiency, is incompatible with traditional encryption. Specifically, traditional encryption requires different users to encrypt their data with their own keys. Thus, identical data copies of different users will lead to different ciphertexts, making deduplication impossible.

Convergent encryption [8] provides a viable option to enforce data confidentiality while realizing deduplication. It encrypts/decrypts a data copy with a *convergent key*, which is derived by computing the cryptographic hash value of the content of the data copy itself [8]. After key generation and data encryption, users retain the keys and send the ciphertext to the cloud. Since encryption is deterministic, identical data copies will generate the same convergent key and the same ciphertext. This allows the cloud to perform deduplication on the ciphertexts. The ciphertexts can only be decrypted by the corresponding data owners with their convergent keys.

To understand how convergent encryption can be realized, we consider a baseline approach that implements convergent encryption based on a layered approach. That is, the original data copy is first encrypted with a convergent key derived by the data copy itself, and the convergent key is then encrypted by a master key that will be kept locally and securely by each user. The encrypted convergent keys are then stored, along with

- *J. Li is with the School of Computer Science, Guangzhou University, China, and also with the Department of Computer Science, Virginia Polytechnic Institute and State University, USA. E-mail: lijin@gzhu.edu.cn.*
- *X. Chen is with the State Key Laboratory of Integrated Service Networks (ISN), Xidian University, Xi'an, China, and also with the Department of Computer Science, Virginia Polytechnic Institute and State University, USA. E-mail: xfchen@xidian.edu.cn.*
- *M. Li and P.P.C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. E-mail: {mqli, pclee}@cse.cuhk.edu.hk.*
- *J. Li is with the College of Information Technical Science, Nankai University, China. E-mail: lijw@mail.nankai.edu.cn.*
- *W. Lou is with the Department of Computer Science, Virginia Polytechnic Institute and State University, USA. E-mail: wjlou@vt.edu.*

Manuscript received 7 July 2013; revised 21 Oct. 2013; accepted 30 Oct. 2013. Date of publication 7 Nov. 2013; date of current version 16 May 2014. Recommended for acceptance by D. Xuan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.284

the corresponding encrypted data copies, in cloud storage. The master key can be used to recover the encrypted keys and hence the encrypted files. In this way, each user only needs to keep the master key and the metadata about the outsourced data.

However, the baseline approach suffers two critical deployment issues. First, it is inefficient, as it will generate an enormous number of keys with the increasing number of users. Specifically, each user must associate an encrypted convergent key with each block of its outsourced encrypted data copies, so as to later restore the data copies. Although different users may share the same data copies, they must have their own set of convergent keys so that no other users can access their files. As a result, the number of convergent keys being introduced linearly scales with the number of blocks being stored and the number of users. This key management overhead becomes more prominent if we exploit fine-grained block-level deduplication. For example, suppose that a user stores 1 TB of data with all unique blocks of size 4 KB each, and that each convergent key is the hash value of SHA-256, which is used by Dropbox for deduplication [17]. Then the total size of keys will be 8 GB. The number of keys is further multiplied by the number of users. The resulting intensive key management overhead leads to the huge storage cost, as users must be billed for storing the large number of keys in the cloud under the pay-as-you-go model.

Second, the baseline approach is unreliable, as it requires each user to dedicatedly protect his own master key. If the master key is accidentally lost, then the user data cannot be recovered; if it is compromised by attackers, then the user data will be leaked.

This motivates us to explore how to efficiently and reliably manage enormous convergent keys, while still achieving secure deduplication. To this end, we propose a new construction called *Dekey*, which provides efficiency and reliability guarantees for convergent key management on both user and cloud storage sides. Our idea is to apply deduplication to the convergent keys and leverage secret sharing techniques. Specifically, we construct secret shares for the convergent keys and distribute them across multiple independent key servers. Only the first user who uploads the data is required to compute and distribute such secret shares, while all following users who own the same data copy need not compute and store these shares again. To recover data copies, a user must access a minimum number of key servers through authentication and obtain the secret shares to reconstruct the convergent keys. In other words, the secret shares of a convergent key will only be accessible by the authorized users who own the corresponding data copy. This significantly reduces the storage overhead of the convergent keys and makes the key management reliable against failures and attacks. To our knowledge, none of existing studies formally address the problem of convergent key management.

This paper makes the following contributions.

- A new construction *Dekey* is proposed to provide efficient and reliable convergent key management through convergent key deduplication and secret sharing. Dekey supports both file-level and block-level deduplications.
- Security analysis demonstrates that Dekey is secure in terms of the definitions specified in the proposed security model. In particular, Dekey remains secure even the adversary controls a limited number of key servers.
- We implement Dekey using the Ramp secret sharing scheme that enables the key management to adapt to different reliability and confidentiality levels. Our evaluation demonstrates that Dekey incurs limited overhead in normal upload/download operations in realistic cloud environments.

This paper is organized as follows. In Section 2, we describe some preliminaries. In Section 3, we present the system model and security requirements of deduplication. Our construction and its security and efficiency analysis are presented in Section 4. The implementation and evaluation have been given in Sections 5 and 6, respectively. Finally, we draw conclusions in Section 7.

2 Preliminaries

In this section, we formally define the cryptographic primitives used in our secure deduplication.

2.1 Symmetric Encryption

Symmetric encryption uses a common secret key κ to encrypt and decrypt information. A symmetric encryption scheme consists of three primitive functions:

- $\mathsf{KeyGen}_{\mathsf{SE}}(1^\lambda) \rightarrow \kappa$ is the key generation algorithm that generates κ using security parameter 1^λ;
- $\mathsf{Encrypt}_{\mathsf{SE}}(\kappa, M) \rightarrow C$ is the symmetric encryption algorithm that takes the secret κ and message M and then outputs the ciphertext C;
- $\mathsf{Decrypt}_{\mathsf{SE}}(\kappa, C) \rightarrow M$ is the symmetric decryption algorithm that takes the secret κ and ciphertext C and then outputs the original message M.

2.2 Convergent Encryption

Convergent encryption [5], [8] provides data confidentiality in deduplication. A user (or data owner) derives a convergent key from each original data copy and encrypts the data copy with the convergent key. In addition, the user derives a *tag* for the data copy, such that the tag will be used to detect duplicates. Here, we assume that the tag correctness property [5] holds, i.e., if two data copies are the same, then their tags are the same. To detect duplicates, the user first sends the tag to the server side to check if the identical copy has been already stored. Note that both the convergent key and the tag are independently derived, and the tag cannot be used to deduce the convergent key and compromise data confidentiality. Both the encrypted data copy and its corresponding tag will be stored on the server side. Formally, a convergent encryption scheme can be defined with four primitive functions:

- $\mathsf{KeyGen}_{\mathsf{CE}}(M) \rightarrow K$ is the key generation algorithm that maps a data copy M to a convergent key K;

- $\mathsf{Encrypt}_{\mathsf{CE}}(K, M) \rightarrow C$ is the symmetric encryption algorithm that takes both the convergent key K and the data copy M as inputs and then outputs a ciphertext C;
- $\mathsf{Decrypt}_{\mathsf{CE}}(K, C) \rightarrow M$ is the decryption algorithm that takes both the ciphertext C and the convergent key K as inputs and then outputs the original data copy M; and
- $\mathsf{TagGen}_{\mathsf{CE}}(M) \rightarrow T(M)$ is the tag generation algorithm that maps the original data copy M and outputs a tag $T(M)$. We allow $\mathsf{TagGen}_{\mathsf{CE}}$ to generate a tag from the corresponding ciphertext as in [5], by using $T(M) = \mathsf{TagGen}_{\mathsf{CE}}(C)$, where $C = \mathsf{Encrypt}_{\mathsf{CE}}(K, M)$.

2.3 Proof of Ownership

The notion of proof of ownership (PoW) is to solve the problem of using a small hash value as a proxy for the entire file in client-side deduplication [11], where the adversary could use the storage service as a content distribution network. This proof mechanism in PoW provides a solution to protect the security in client-side deduplication. In this way, a client can prove to the server that it indeed has the file. Dekey supports client-side deduplication with PoW to enable users to prove their ownership of data copies to the storage server. Specifically, PoW is implemented as an interactive algorithm (denoted by `PoW`) run by a prover (i.e., user) and a verifier (i.e., storage server). The verifier derives a short value $\phi(M)$ from a data copy M. To prove the ownership of the data copy M, the prover needs to send ϕ' and run a proof algorithm with the verifier. It is passed if and only if $\phi' = \phi(M)$ and the proof is correct. In our paper, we use the notations of $\mathsf{PoW}_{\mathsf{F}}$ and $\mathsf{PoW}_{\mathsf{B}}$ to denote PoW for a file F and block B, respectively. Specifically, the notation of $\mathsf{PoW}_{\mathsf{F},\mathsf{j}}$ will be used to denote a PoW protocol with respect to $T_j(F) = \mathsf{TagGen}_{\mathsf{CE}}(F, j)$. These notations will be further explained in Section 4.

2.4 Ramp Secret Sharing

Dekey uses the Ramp secret sharing scheme (RSSS) [6], [25] to store convergent keys. Specifically, the (n, k, r)-RSSS (where $n > k > r \geq 0$) generates n shares from a secret such that 1) the secret can be recovered from any k shares but cannot be recovered from fewer than k shares, and 2) no information about the secret can be deduced from any r shares. It is known that when $r = 0$, the $(n, k, 0)$-RSSS becomes the (n, k) Rabin's Information Dispersal Algorithm (IDA) [23]; when $r = k - 1$, the $(n, k, k-1)$-RSSS becomes the (n, k) Shamir's Secret Sharing Scheme (SSSS) [26]. The (n, k, r)-RSSS builds on two primitive functions:

- `Share` divides a secret S into $(k - r)$ pieces of equal size, generates r random pieces of the same size, and encodes the k pieces using a non-systematic k-of-n erasure code[1] into n shares of the same size;
- `Recover` takes any k out of n shares as inputs and then outputs the original secret S.

To make the generated shares appropriate for deduplication, we replace the above random pieces with pseudorandom pieces in the implementation of Dekey. The details of how to generate such pseudorandom pieces are elaborated in Section 5.

Dekey uses RSSS to provide a *tunable* key management mechanism to balance among confidentiality, reliability, storage overhead, and performance. We study the effects of different parameters in Section 6.

3 Problem Formulation

3.1 System Model

We first formulate a data outsourcing model used by Dekey. There are three entities, namely: the user, the storage cloud service provider (S-CSP), and the key-management cloud service provider (KM-CSP), as elaborated below.

- *User*. A user is an entity that wants to outsource data storage to the S-CSP and access the data later. To save the upload bandwidth, the user only uploads unique data but does not upload any duplicate data, which may be owned by the same user or different users.
- *S-CSP*. The S-CSP provides the data outsourcing service and stores data on behalf of the users. To reduce the storage cost, the S-CSP eliminates the storage of redundant data via deduplication and keeps only unique data.
- *KM-CSP*. A KM-CSP maintains convergent keys for users, and provides users with minimal storage and computation services to facilitate key management. For fault tolerance of key management, we consider a quorum of KM-CSPs, each being an independent entity. Each convergent key is distributed across multiple KM-CSPs using RSSS (see Section 2).

In this work, we refer a data copy to be either a whole file or a smaller-size block, and this leads to two types of deduplication: 1) *file-level deduplication*, which eliminates the storage of any redundant files, and 2) *block-level deduplication*, which divides a file into smaller fixed-size or variable-size blocks and eliminates the storage of any redundant blocks. Using fixed-size blocks simplifies the computations of block boundaries, while using variable-size blocks (e.g., based on Rabin fingerprinting [22]) provides better deduplication efficiency. We deploy our deduplication mechanism in both file and block levels. Specifically, to upload a file, a user first performs the file-level duplicate check. If the file is a duplicate, then all its blocks must be duplicates as well; otherwise, the user further performs the block-level duplicate check and identifies the unique blocks to be uploaded. Each data copy (i.e., a file or a block) is associated with a *tag* for the duplicate check (see Section 2). All data copies and tags will be stored in the S-CSP.

1. As discussed in [14], not all non-systematic k-of-n erasure codes can be used here. To provide the confidentiality that an (n, k, r)-RSSS promises, we choose the erasure code whose generator matrix is a Cauchy matrix.

3.2 Threat Model and Security Goals

Our threat model considers two types of attackers: 1) An outside attacker may obtain some knowledge (e.g., a hash value) of the data copy of interest via public channels. It plays a role of a user that interacts with the S-CSP. This kind of attacker includes the adversary who uses the S-CSP as an content distribution network; 2) An inside attacker is honest-but-curious, and it could refer to the S-CSP or any of the KM-CSPs. Its goal is to extract useful information of user data or convergent keys. We require the inside attacker to follow the protocol correctly.

Here, we allow the collusion between the S-CSP and KM-CSPs. However, we require that the number of colluded KM-CSPs is not more than a predefined threshold r if the (n, k, r)-RSSS is used (see Section 2), such that a convergent key cannot be guessed for an unpredictable message by a brute-force attack from the colluded KM-CSPs.

We aim to achieve the following security goals:

- *Semantic security of convergent keys*. We require that the convergent keys distributed stored among the KM-CSPs remain semantically secure, even if the adversary controls a predefined number of KM-CSPs. Furthermore, these KM-CSPs are also allowed to collude with S-CSP and users. The goal of the adversary is to retrieve and recover the convergent keys for files that do not belong to them.
- *Data confidentiality*. We require that the encrypted data copies be semantically secure when they are unpredictable (i.e., have high min-entropy). Actually, this requirement has recently been formalized in [5] and called the privacy against chosen distribution attack. This also implies that the data is secure against the adversary who does not own the data. That is, the user cannot get the ownership of the data from the S-CSP and KM-CSPs by running the PoW protocol if the user does not have the file.

4 Constructions

In this section, we present a baseline approach that realizes convergent encryption in deduplication, and discuss the limitations of the baseline approach in key management. To this end, we present our construction *Dekey*, which aims to mitigate the key management overhead and provide fault tolerance guarantees for key management, while preserving the required security properties of secure deduplication.

4.1 Baseline Approach

The baseline approach involves only the user and the S-CSP (i.e., no KM-CSPs are required). Its idea is that each user has all his data copies encrypted by the corresponding convergent keys, which are then further encrypted by an independent master key. The encrypted convergent keys are outsourced to the S-CSP, while the master key is securely maintained by the user. The details of the baseline approach are elaborated as follows.

4.1.1 System Setup

The system setup phase initializes the necessary parameters in the following two steps:

S1: The following entities are initialized: 1) a symmetric encryption scheme with the primitive functions ($\mathsf{KeyGen}_{\mathsf{SE}}$, $\mathsf{Encrypt}_{\mathsf{SE}}, \mathsf{Decrypt}_{\mathsf{SE}}$) and the user's master key $\kappa = \mathsf{KeyGen}_{\mathsf{SE}}(1^{\lambda})$ for some security parameter 1^{λ}; 2) a convergent encryption scheme with the primitive functions ($\mathsf{KeyGen}_{\mathsf{CE}}, \mathsf{Encrypt}_{\mathsf{CE}}, \mathsf{Decrypt}_{\mathsf{CE}}, \mathsf{TagGen}_{\mathsf{CE}}$); and 3) a PoW algorithm $\mathsf{PoW}_{\mathsf{F}}$ for the file and a PoW algorithm for the block, which is denoted by $\mathsf{PoW}_{\mathsf{B}}$.

S2: The S-CSP initializes two types of storage systems: a rapid storage system for storing the tags for efficient duplicate checks, and a file storage system for storing both encrypted data copies and encrypted convergent keys. Both storage systems are initialized to be $\perp$.

4.1.2 File Upload

Suppose that a user uploads a file F. First, it performs file-level deduplication as follows.

S1: On input file F, the user computes and sends the file tag $T(F) = \mathsf{TagGen}_{\mathsf{CE}}(F)$ to the S-CSP.

S2: Upon receiving $T(F)$, the S-CSP checks whether there exists the same tag on the S-CSP. If so, the S-CSP replies the user with a response ''file duplicate,'' or ''no file duplicate'' otherwise.

S3: If the user receives the response ''no file duplicate'', then it jumps to S5 to proceed with block-level deduplication. If the response is ''file duplicate,'' then the user runs $\mathsf{PoW}_{\mathsf{F}}$ on F with the S-CSP to prove that it actually owns the same file F that is stored on the S-CSP.

S4: If $\mathsf{PoW}_{\mathsf{F}}$ is passed, the S-CSP simply returns a file pointer of F to the user, and no further information will be uploaded. If $\mathsf{PoW}_{\mathsf{F}}$ fails, the S-CSP aborts the upload operation.

The user then performs block-level deduplication to further eliminate any redundant blocks, as described below.

S5: On input file F and the master key κ, the user performs the following computations: 1) Divide F into a set of blocks $\{B_i\}$ (where $i = 1, 2, \ldots$); 2) for each block B_i, compute block tag $T(B_i) = \mathsf{TagGen}_{\mathsf{CE}}(B_i)$; and 3) Send the set of block tags $\{T(B_i)\}$ to the S-CSP for duplicate checks.

S6: Upon receiving block tags $\{T(B_i)\}$, the S-CSP computes a block signal vector σ_B in the following manner: for each i, if there exists some stored block tag that matches $T(B_i)$, the S-CSP sets $\sigma_B[i] = 1$ to indicate ''block duplicate''; otherwise it sets $\sigma_B[i] = 0$ to indicate ''no block duplicate'' and also stores $T(B_i)$ in its rapid storage. Then, the S-CSP returns σ_B to the user.

S7: Upon receiving the signal σ_B, the user performs the following operations: for each i, if $\sigma_B[i] = 1$, the user runs $\mathsf{PoW}_{\mathsf{B}}$ on B_i with the S-CSP to prove that it owns the block B_i. If it is passed, the S-CSP simply returns a block poiner of B_i to the user. Then, the user keeps the block pointer of B_i and does not need to upload B_i; otherwise it computes the

encrypted block $C_i = \mathsf{Encrypt}_{\mathsf{CE}}(K_i, B_i)$ with the convergent key $K_i = \mathsf{KeyGen}_{\mathsf{CE}}(B_i)$.

S8: For all blocks $\{B_i\}$, the user also computes the encrypted convergent keys $\{CK_i\}$, where $CK_i = \mathsf{Encrypt}_{\mathsf{SE}}(\kappa, K_i)$ with the master key κ and convergent key K_i.

S9: The user uploads the unique blocks B_i's with $\sigma_B[i] = 0$, all encrypted convergent keys $\{CK_i\}$ and $T(F)$ to the S-CSP, which then stores them in the file storage system.

4.1.3 File Download

Suppose a user wants to download a file F. It first sends a request and the file name to the S-CSP and performs the following steps.

S1: Upon receiving the request and file name, the S-CSP will check whether the user is eligible to download F. If failed, the S-CSP sends back an `abort` signal to the user to indicate the download failure. Otherwise, the S-CSP returns the corresponding ciphertexts $\{C_i\}$ and the encrypted convergent keys $\{CK_i\}$ to the user.

S2: Upon receiving the encrypted data from the S-CSP, the user first uses his master key to recover each convergent key $K_i = \mathsf{Decrypt}_{\mathsf{SE}}(\kappa, CK_i)$. Then it uses K_i to recover the original block $B_i = \mathsf{Decrypt}_{\mathsf{CE}}(K_i, C_i)$. Finally, the user can obtain the original file $F = \{B_i\}$.

4.1.4 Limitations

The baseline approach suffers two major problems. The first problem is the enormous storage overhead in key management. In particular, each user must associate a convergent key with each data copy that he owns and encrypts all convergent keys with his own master key. The encrypted convergent keys (i.e., CK_i's) are different across users due to the different master keys. Thus, the number of convergent keys increases linearly with the number of unique data copies being stored and the number of users, thereby imposing heavy storage overhead. Another problem is that the master key presents the single-point-of-failure and needs to be securely and reliably maintained by the user.

4.2 Dekey

Dekey is designed to efficiently and reliably maintain convergent keys. Its idea is to enable deduplication in convergent keys and distribute the convergent keys across multiple KM-CSPs. Instead of encrypting the convergent keys on a per-user basis, Dekey constructs secret shares on the original convergent keys (that are in plain) and distributes the shares across multiple KM-CSPs. If multiple users share the same block, they can access the same corresponding convergent key. This significantly reduces the storage overhead for convergent keys. In addition, this approach provides fault tolerance and allows the convergent keys to remain accessible even if *any* subset of KM-CSPs fails. We now elaborate the details of Dekey as follows.

4.2.1 System Setup

The system setup phase in Dekey is similar to that in the baseline approach, but involves an additional step for initializing the key storage in KM-CSPs. In Dekey, we assume that the number of KM-CSPs is n.

S1: On input security parameter 1^λ, the user initializes a convergent encryption scheme, and two PoW protocols $\mathsf{POW_F}$ and $\mathsf{POW_B}$ for the file ownership proof and block ownership proof, respectively.

S2: The S-CSP initializes both the rapid storage system and the file storage system and set them to be $\perp$.

S3: Each KM-CSP initializes a rapid storage system for block tags and a lightweight storage system for holding convergent key shares, and sets them to be $\perp$.

4.2.2 File Upload

To upload file F, the user and the S-CSP perform both file-level and block-level deduplications. The file-level deduplication operation is identical to that in the baseline approach. More precisely, the user sends the file tag $T(F)$ to the S-CSP for the file duplicate check. If a file duplicate is found, the user will run the PoW protocol $\mathsf{POW_F}$ with the S-CSP to prove the file ownership. It skips the block-level duplicate check and jumps to the key distribution stage. If no duplicate exists, then block-level deduplication will be performed as the same as S5-S7 of the baseline scheme. Finally, the S-CSP stores the ciphertext C_i with $\sigma_B[i] = 0$ and returns the corresponding pointers back to user for local storage.

After both file-level and block-level duplicate checks, an additional stage called *key distribution* is performed. As opposed to the baseline approach, this stage enables Dekey to not rely on keeping a master secret key for each user, but instead share each convergent key among multiple KM-CSPs. If a file duplicate is found on S-CSP, the user will run the PoW protocol $\mathsf{POW_{F,j}}$ for the tag $T_j(F) = \mathsf{TagGen}_{\mathsf{CE}}(F, j)$ with the j-th KM-CSP to prove the file ownership. All the pointers for the key shares of F stored on the j-the KM-CSP will be returned to the user if the proof is passed. If no file duplicate is found, the following steps will be taken.

S1: On input file $F = \{B_i\}$, for each block B_i, the user computes and sends the block tag $T(B_i) = \mathsf{TagGen}_{\mathsf{CE}}(B_i)$ to each KM-CSP. Furthermore, a file tag $T_j(F) = \mathsf{TagGen}_{\mathsf{CE}}(F, j)$ will be computed and sent to the j-th KM-CSP, $1 \le j \le n$.

S2: For each received $T(B_i)$, the j-th KM-CSP checks whether another same tag has been stored: if so, a PoW for block $\mathsf{POW_{B,j}}$ will be performed between the user and j-th KM-CSP with respect to $T_j(B_i) = \mathsf{TagGen}_{\mathsf{CE}}(B_i, j)$. If it is passed, the j-th KM-CSP will return a pointer for the secret share stored for the convergent key K_i to the user; otherwise it keeps $T(B_i)$ and sends back a signal to ask for the secret share on the convergent key.

S3: Upon receiving results for a block B_i returned from KM-CSPs, if it is a valid pointer, the user stores it locally; otherwise the user computes the secret shares $K_{i1}, K_{i2}, \ldots, K_{ik}$ by running $\mathsf{Share}(K_i)$ using the (n, k, r)-RSSS. It then sends the share K_{ij} and $T_j(B_i) = \mathsf{TagGen}_{\mathsf{CE}}(B_i, j)$ to the j-th KM-CSP for $j = 1, 2, \ldots, n$ via a secure channel.

S4: Upon receiving K_{ij} and $T_j(B_i)$, the j-th KM-CSP stores them and sends back the pointer for K_{ij} to the user for future access.

4.2.3 File Download

To download file F, the user first downloads the encrypted blocks $\{C_i\}$ from the S-CSP as described in the baseline scheme. It needs to decrypt these encrypted blocks by recovering the convergent keys. Specifically, the user sends all the pointers for F to k out of n KM-CSPs and fetches the corresponding shares K_{ij} for each block B_i. After gathering all the shares, the user continues to reconstruct the convergent key $K_i = \mathsf{Recover}(\{K_{ij}\})$ for B_i. Finally, the encrypted blocks $\{C_i\}$ can be decrypted with $\{K_i\}$ to obtain the original file F.

Remarks. Note that the reason of introducing an index j in Step S2 in the file upload phase is to prevent one server from getting the key shares of the other KM-CSPs for the same block. For example, we can implement $T_j(B_i) = H_j(B_i)$ with a cryptographic hash function and use it as a proof of $\mathsf{POW}_{\mathsf{B,j}}$. In this way, a server with $T_j(B_i)$ could not compute and send a valid proof $T_{j'}(B_i)$ to the j'-th KM-CSP. To further save the communication cost, the user could perform the duplicate check with only one of KM-CSPs at first. Then he processes the proof with the other servers depends on the result returned from this KM-CSP, which could save his communication cost.

4.3 Security Analysis

Dekey is designed to solve the key management problem in secure deduplication where the files have been encrypted by utilizing convergent encryption. Some basic tools have been used to construct the secure deduplication and key management protocols. Thus, we assume that the underlying building blocks are secure, which include the convergent encryption scheme, symmetric encryption scheme, and the PoW scheme. Based on this assumption, we show that Dekey is secure with respect to the security of keys and data, as detailed below.

4.3.1 Confidentiality of Outsourced Data at S-CSP

The files have been encoded by the convergent encryption scheme before being outsourced to the S-CSP. Thus, the confidentiality of data can be achieved if the adversary cannot get the secret keys in convergent encryption or break the security of convergent encryption. Several security notions, for example, privacy against chosen distribution attack, have been defined for the confidentiality. Thus, our construction can also achieve the security for data based on a secure convergent encryption scheme if the encryption key is securely kept by the user.

4.3.2 Security of Convergent Encryption Key

In our construction, the convergent encryption keys are securely stored at the KM-CSPs in a distributed manner. Thus, the semantic security of convergent keys could be guaranteed even if any r KM-CSPs collude. This could be easily achieved because RSSS is a semantically secure secret sharing scheme even if any r of n shares have been leaked. Recall that it requires the user to perform a PoW protocol for the corresponding shares stored at different KM-CSPs. We require that the values used in PoW with different KM-CSPs are independent and the adversary cannot compute the other short value even if he has the knowledge of r values $T_j(B_i)$. Actually, in our implementation, $T_j(B_i)$ is implemented with a cryptographic hash function $H_j(\cdot)$ and the above assumption will be held obviously. In this way, if the adversary wants to get the j-th key share that it does not have, then he has to convince the j-th KM-CSP by running a PoW protocol. However, the values used to perform PoW with different KM-CSPs are different and the adversary cannot derive the other key shares even if he could get r shares from dishonest KM-CSPs controlled by him.

5 Implementation

In this section, we discuss the implementation details of Dekey. Dekey builds on the Ramp secret sharing scheme (RSSS) [6], [25] to distribute the shares of convergent keys across multiple key servers (see Section 2).

5.1 RSSS with Pseudorandomness

In Dekey, the RSSS secret is the hash key H_0 of a data block B, where $H_0 = \mathrm{hash}(B)$. Recall from Section 2 that the `Share` function of the (n, k, r)-RSSS embeds r random pieces to achieve a confidentiality level of r. One challenge is that randomization conflicts with deduplication, since the random pieces cannot be deduplicated with each other. Instead of directly adopting RSSS, we here replace these random pieces with *pseudorandom* pieces in our Dekey implementation.

We generate the r pseudorandom pieces as follows. Let $m = \lceil \frac{r}{(k-r)} \rceil$. We first generate m additional hash values as $H_1 = \mathrm{hash}(B+1), H_2 = \mathrm{hash}(B+2), \ldots, H_m = \mathrm{hash}(B+m)$. We then fill in the r pieces with the generated m additional hash values $H_1, H_2, \ldots, H_m$. These r pieces are pseudorandom because

1. $H_1, H_2, \ldots, H_m$ cannot be guessed by attackers as long as the corresponding data block B is unknown; and
2. $H_1, H_2, \ldots, H_m$ together with H_0 cannot be deduced from each other as long as the corresponding data block B is unknown.

The parameters n, k, and r determine the following four factors, whose effects are evaluated in Section 6:

- *Confidentiality level*: It is decided by the parameter r.
- *Reliability level*: It depends on the parameters n and k, and can be defined by $n - k$.
- *Storage blowup*: It determines the key management overhead and depends on the parameters n, k, and r. It can be theoretically calculated by $\frac{n}{k-r}$.
- *Performance*: It refers to the *encoding performance* and *decoding performance* when using the k-of-n erasure code in the `Share` and `Recover` functions, respectively.

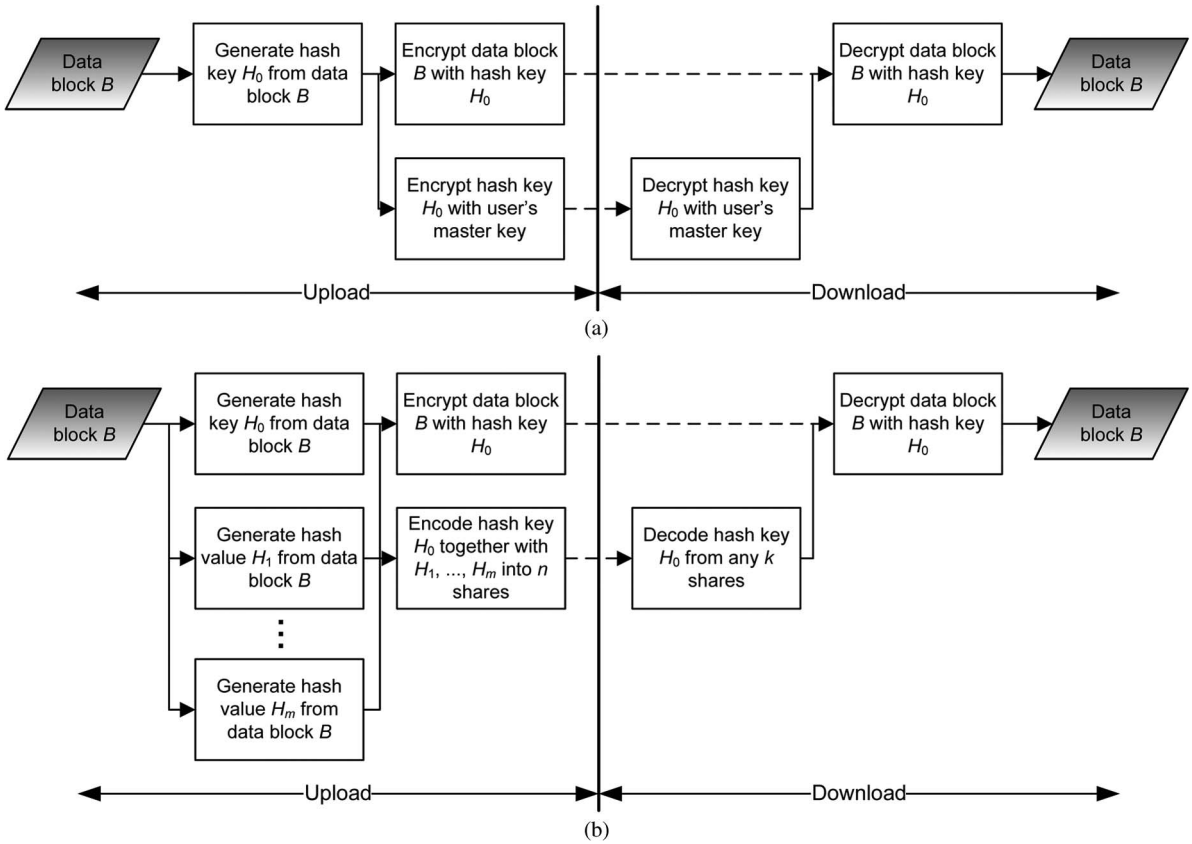


Fig. 1. Flow block diagrams of core modules in two different approaches. (a) Baseline approach (keeping the hash key with an encryption scheme). (b) Dekey (keeping the hash key with (n, k, r)-RSSS).

5.2 Implementation Details

Fig. 1 presents the flow block diagrams of core modules in the baseline approach and Dekey that we implement. In this figure, we omit the ordinary file transfer and deduplication modules for simplification. To make full use of the multi-core feature of contemporary processors, we assume that these modules running in parallel on different cores in a pipeline style. In the baseline approach, we simply encrypt each hash key H_0 with the user's master key, while in Dekey, we generate n shares of H_0.

We choose 4 KB as the default data block size. A larger data block size (e.g., 8 KB instead of 4 KB) results in better encoding/decoding performance due to fewer chunks being managed, but has less storage reduction offered by deduplication [7], [15], [16], [29]. For each data block, a hash key of size 32 bytes is generated using the hash function SHA-256, which belongs to the family of SHA-2 that is now recommended by the US National Institute of Standards and Technology (NIST) [2]. In addition, we adopt the symmetric-key encryption algorithm AES-256 in Cipher-Block Chaining (CBC) mode as the default encryption algorithm. Both SHA-256 and AES-256 are implemented using the EVP library of OpenSSL Version 1.0.1e [1].

We implement the RSSS based on Jerasure Version 1.2 [20]. Regarding to the encoding and decoding modules in Fig. 1b, the choice of code symbol size w (in bits) deserves our discussion here. For an erasure code, a code symbol of size w bits refers to a basic unit of encoding and decoding operations, both of which are performed in a finite field $GF(2^w)$. In the (n, k, r)-RSSS, we choose the erasure code whose generator matrix is a Cauchy matrix, and thus, w should meet the condition $2^w \geq n + k$ [21]. However, when each hash key is divided into $(k - r)$ pieces with a size of multiple w, its size (i.e., 32 bytes) is often not a multiple of $w \times (k - r)$. We thus often need to pad additional zeros to fill in the $(k - r)$ pieces, resulting in different storage blowup ratios. Fig. 2a shows the storage blowup ratios versus different values of w for (6, 4, 2)-RSSS. We see that for some w, the storage blowup ratio can be much higher than the theoretical value calculated by $\frac{n}{(k-r)}$. However, we find that if the minimum w is chosen, the practical storage blowup can often be closely matched to the theoretical value. In addition, we evaluate the corresponding encoding and decoding times on an Intel Xeon E5530 (2.40 GHz) server with Linux 3.2.0-23-generic OS, and the results are shown in Fig. 2b. We find that the encoding and decoding times increase with w. Therefore, our Dekey implementation always chooses the minimum w that meets $2^w \geq n + k$.

6 Evaluation

In this section, we evaluate the encoding and decoding performance of Dekey on generating and recovering key shares, respectively. All our experiments were performed on an Intel Xeon E5530 (2.40 GHz) server with Linux 3.2.0-23-generic OS.

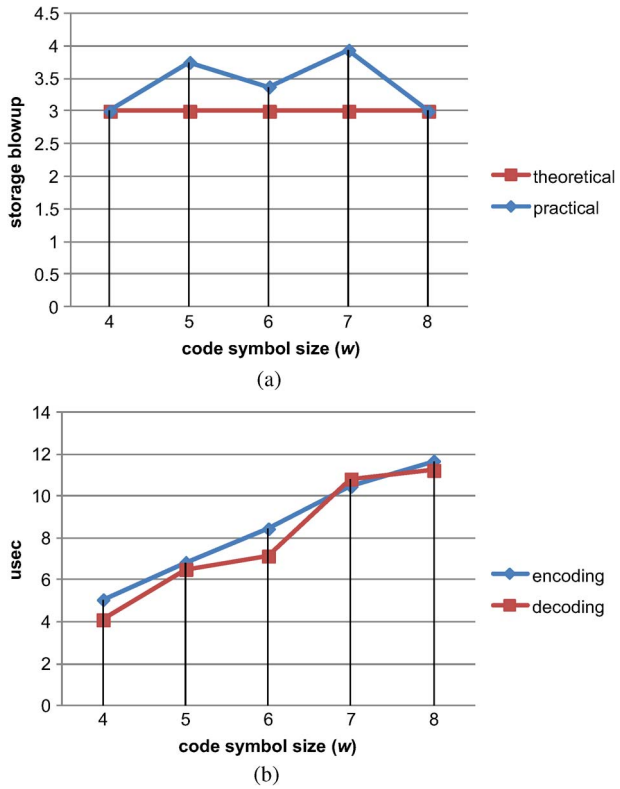


Fig. 2. Impact of code symbol size w on storage blowup and performance in the case of (6, 4, 2)-RSSS. (a) Storage blowup. (b) Encoding/decoding time.

6.1 Overall Results

With (n, k, r)-RSSS being used, we test all the following cases: $3 \leq n \leq 8$, $2 \leq k \leq n-1$, and $1 \leq r \leq k-1$, as shown in Fig. 3. We can see that the encoding and decoding times of Dekey for each hash key (per 4 KB data block) are always on the order of microseconds, and hence are negligible compared to the data transfer performance in the Internet setting. Note that the encoding time in general is higher than the decoding time, mainly because the encoding operation involves all n shares, while the decoding operation only involves a subset of $k < n$ shares.

We first evaluate several basic modules that appear in both the baseline approach and Dekey:

- Average time for generating a 32-byte hash from a 4 KB data block: 25.196 usec;
- Average time for encrypting a 4 KB data block with its 32-byte hash: 23.518 usec;
- Average time for decrypting a 4 KB data block with its 32-byte hash: 22.683 usec.

Comparing the above results and those in Fig. 3, we can see that the encoding/decoding overhead of Dekey can be masked by that of the basic modules via parallelization (see Section 5.2). Specifically, during the file upload, the encoding time of Dekey is less than 20 usec in most cases, and is less than that of encrypting a data block. If we parallelize both the encoding and encryption modules, then the bottleneck lies in the encryption part. During the file download, the decoding time is less than the time of decrypting a data block. We can pipeline both the decoding and decryption modules, making the decryption part become the bottleneck. We have also tested the cases with other data block sizes (like 2 KB and 8 KB) and made similar observations.

In the following, we highlight some evaluation results with respect to some specific factors, including the number of KM-CSPs n, the confidentiality level r, and the reliability level $n-k$.

6.2 Number of KM-CSPs n

Fig. 4 shows the encoding/decoding times versus the number of KM-CSPs n, where we fix the confidentiality level $r = 2$ and the reliability level $n - k = 2$. As expected, the encoding/decoding times increase with n since more shares are involved.

6.3 Confidentiality Level r

Fig. 5 shows the encoding/decoding times versus the confidentiality level r, where we fix the number of KM-CSPs $n = 6$ and the reliability level $n - k = 2$. The encoding/decoding times increase with r. Recall that the `Share` function of RSSS divides a secret into $k - r$ equal-size pieces (see Section 2). With a larger r, the size of each piece increases, and this increases the encoding/decoding overhead as well.

6.4 Reliability Level $n - k$

Fig. 6 shows the encoding/decoding times versus the reliability level $n - k$, where we fix $n = 6$ and the

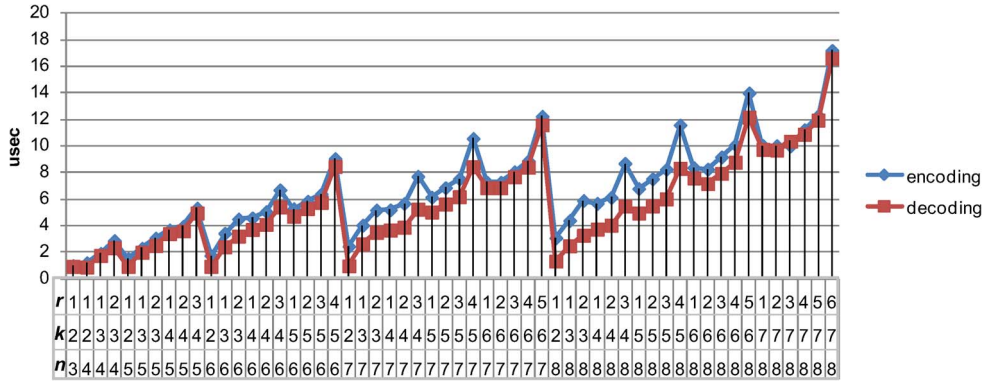


Fig. 3. Encoding and decoding times for Dekey.

confidentiality level $r = 2$. With the increase of $n - k$, the encoding/decoding times decrease since fewer pieces (i.e., k) are being erasure-coded.

7 Related Work

7.1 Traditional Encryption

To protect the confidentiality of outsourced data, various cryptographic solutions have been proposed in the literature (e.g., [13], [30], [31], [33]). Their idea builds on traditional (symmetric) encryption, in which each user encrypts data with an independent secret key. Some studies [10], [28] propose to use threshold secret sharing [26] to maintain the robustness of key management. However, the above studies do not consider deduplication. Using traditional encryption, different users will simply encrypt identical data copies with their own keys, but this will lead to different ciphertexts and hence make deduplication impossible.

7.2 Convergent Encryption

Convergent encryption [8] ensures data privacy in deduplication. Bellare *et al.* [5] formalize this primitive as message-locked encryption, and explore its application in space-efficient secure outsourced storage. There are also several implementations of convergent implementations of different convergent encryption variants for secure deduplication (e.g., [4], [24], [27], [32]). It is known that some commercial cloud storage providers, such as Bitcasa, also deploy convergent encryption [5]. However, as stated before, convergent encryption leads to a significant number of convergent keys.

7.3 Proof of Ownership

Halevi *et al.* [11] propose ''proofs of ownership'' (PoW) for deduplication systems, such that a client can efficiently prove to the cloud storage server that he/she owns a file without uploading the file itself. Several PoW constructions based on the Merkle Hash Tree are proposed [11] to enable client-side deduplication, which include the bounded leakage setting. Pietro and Sorniotti [19] propose another efficient PoW scheme by choosing the projection of a file onto some randomly selected bit-positions as the file proof. Note that all the above schemes do not consider data privacy. Recently, Ng *et al.* [18] extend PoW for encrypted files, but they do not address how to minimize the key management overhead.

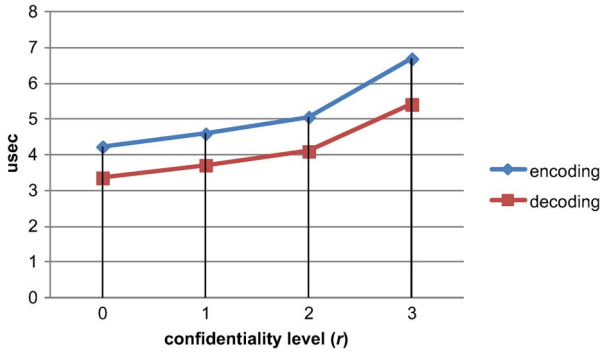


Fig. 5. Impact of confidentiality level r on the encoding/decoding times where $n = 6$ and $n - k = 2$.

8 Conclusion

We propose Dekey, an efficient and reliable convergent key management scheme for secure deduplication. Dekey applies deduplication among convergent keys and distributes convergent key shares across multiple key servers, while preserving semantic security of convergent keys and confidentiality of outsourced data. We implement Dekey using the Ramp secret sharing scheme and demonstrate that it incurs small encoding/decoding overhead compared to the network transmission overhead in the regular upload/download operations.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (61100224 and 61272455), the Fundamental Research Funds for the Central Universities (K50511010001 and JY10000901034), China 111 Project (B08038), GRF CUHK 413813 from the Research Grant Council of Hong Kong, and seed grants from the CUHK MoE-Microsoft Key Laboratory of Human-centric Computing and Interface Technologies. Besides, Lou's work is supported by U.S. National Science Foundation under Grant (CNS-1217889).

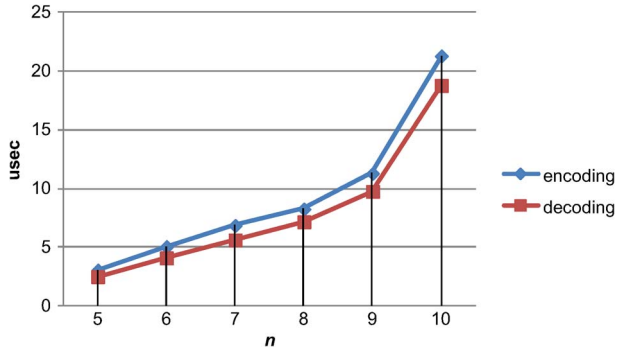


Fig. 4. Impact of number of KM-CSPs n on encoding/decoding times, where $r = 2$ and $n - k = 2$.

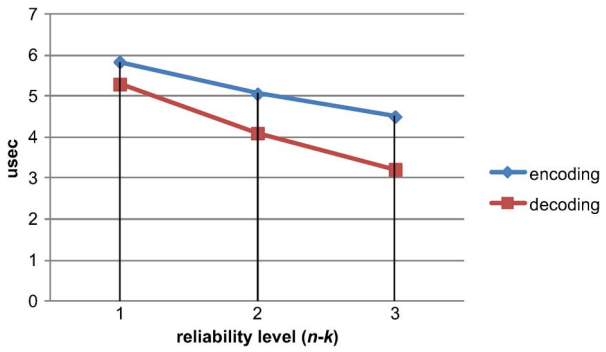


Fig. 6. Impact of reliability level $n - k$ on encoding/decoding times, where $n = 6$ and $r = 2$.

REFERENCES

[1] OpenSSL Project. [Online]. Available: http://www.openssl.org/

[2] NIST's Policy on Hash Functions, Sept. 2012. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/policy.html

[3] AmazonCase Studies. [Online]. Available: https://aws.amazon.com/solutions/case-studies/#backup

[4] P. Anderson and L. Zhang, ''Fast and Secure Laptop Backups with Encrypted De-Duplication,'' in *Proc. USENIX LISA*, 2010, pp. 1-8.

[5] M. Bellare, S. Keelveedhi, and T. Ristenpart, ''Message-Locked Encryption and Secure Deduplication,'' in *Proc. IACR Cryptology ePrint Archive*, 2012, pp. 296-3122012:631.

[6] G.R. Blakley and C. Meadows, ''Security of Ramp Schemes,'' in *Proc. Adv. CRYPTO*, vol. 196, *Lecture Notes in Computer Science*, G.R. Blakley and D. Chaum, Eds., 1985, pp. 242-268.

[7] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li, ''Decentralized Deduplication in San Cluster File Systems,'' in *Proc. USENIX ATC*, 2009, p. 8.

[8] 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. ICDCS*, 2002, pp. 617-624.

[9] J. Gantz and D. Reinsel, *The Digital Universe in 2020: Big Data, Bigger Digital Shadows, Biggest Growth in the Far East*, Dec. 2012. [Online]. Available: http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

[10] R. Geambasu, T. Kohno, A. Levy, and H.M. Levy, ''Vanish: Increasing Data Privacy with Self-Destructing Data,'' in *Proc. USENIX Security Symp.*, Aug. 2009, pp. 316-299.

[11] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, ''Proofs of Ownership in Remote Storage Systems,'' in *Proc. ACM Conf. Comput. Commun. Security*, Y. Chen, G. Danezis, and V. Shmatikov, Eds., 2011, pp. 491-500.

[12] 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, Nov./Dec. 2010.

[13] S. Kamara and K. Lauter, ''Cryptographic Cloud Storage,'' in *Proc. Financial Cryptography: Workshop Real-Life Cryptograph. Protocols Standardization*, 2010, pp. 136-149.

[14] M. Li, ''On the Confidentiality of Information Dispersal Algorithms and their Erasure Codes,'' in *Proc. CoRR*, 2012, pp. 1-4abs/1206.4123.

[15] D. Meister and A. Brinkmann, ''Multi-Level Comparison of Data Deduplication in a Backup Scenario,'' in *Proc. SYSTOR*, 2009, pp. 1-12.

[16] D.T. Meyer and W.J. Bolosky, ''A Study of Practical Deduplication,'' in *Proc. 9th USENIX Conf. FAST*, 2011, pp. 1-13.

[17] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl, ''Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space,'' in *Proc. USENIX Security*, 2011, p. 5.

[18] W.K. Ng, Y. Wen, and H. Zhu, ''Private Data Deduplication Protocols in Cloud Storage,'' in *Proc. 27th Annu. ACM Symp. Appl. Comput.*, S. Ossowski and P. Lecca, Eds., 2012, pp. 441-446.

[19] R.D. Pietro and A. Sorniotti, ''Boosting Efficiency and Security in Proof of Ownership for Deduplication,'' in *Proc. ACM Symp. Inf., Comput. Commun. Security*, H.Y. Youm and Y. Won, Eds., 2012, pp. 81-82.

[20] J.S. Plank, S. Simmerman, and C.D. Schuman, ''Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications—Version 1.2,'' University of Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08-627, Aug. 2008.

[21] J.S. Plank and L. Xu, ''Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications,'' in *Proc. 5th IEEE Int'l Symp. NCA*, Cambridge, MA, USA, July 2006, pp. 173-180.

[22] M.O. Rabin, ''Fingerprinting by Random Polynomials,'' Center for Research in Computing Technology, Harvard University, Cambridge, MA, USA, Tech. Rep. TR-CSE-03-01, 1981.

[23] M.O. Rabin, ''Efficient Dispersal of Information for Security, Load Balancing, Fault Tolerance,'' *J. ACM*, vol. 36, no. 2, pp. 335-348, Apr. 1989.

[24] A. Rahumed, H.C.H. Chen, Y. Tang, P.P.C. Lee, and J.C.S. Lui, ''A secure Cloud Backup System with Assured Deletion and Version Control,'' in *Proc. 3rd Int'l Workshop Security Cloud Comput.*, 2011, pp. 160-167.

[25] A.D. Santis and B. Masucci, ''Multiple Ramp Schemes,'' *IEEE Trans. Inf. Theory*, vol. 45, no. 5, pp. 1720-1728, July 1999.

[26] A. Shamir, ''How to Share a Secret,'' *Commun. ACM*, vol. 22, no. 11, pp. 612-613, 1979.

[27] M.W. Storer, K. Greenan, D.D.E. Long, and E.L. Miller, ''Secure Data Deduplication,'' in *Proc. StorageSS*, 2008, pp. 1-10.

[28] Y. Tang, P.P. Lee, J.C. Lui, and R. Perlman, ''Secure Overlay Cloud Storage with Access Control and Assured Deletion,'' *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 6, pp. 903-916, Nov./Dec. 2012.

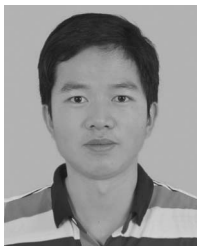
[29] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, ''Characteristics of Backup Workloads in Production Systems,'' in *Proc. 10th USENIX Conf. FAST*, 2012, pp. 1-16.

[30] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, ''Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 5, pp. 847-859, May 2011.

[31] W. Wang, Z. Li, R. Owens, and B. Bhargava, ''Secure and Efficient Access to Outsourced Data,'' in *Proc. ACM CCSW*, Nov. 2009, pp. 55-66.

[32] Z. Wilcox-O'Hearn and B. Warner, ''Tahoe: The Least-Authority Filesystem,'' in *Proc. ACM StorageSS*, 2008, pp. 21-26.

[33] A. Yun, C. Shi, and Y. Kim, ''On Protecting Integrity and Confidentiality of Cryptographic File System for Outsourced Storage,'' in *Proc. ACM CCSW*, Nov. 2009, pp. 67-76.

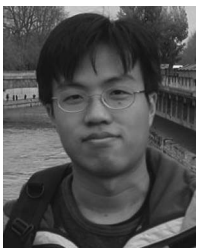


Jin Li received his BS degree in mathematics from Southwest University, in 2002. He got his PhD degree in information security from Sun Yat-sen University, in 2007. Currently, he works at Guangzhou University as a professor. He has been selected as one of science and technology new star in Guangdong province. His research interests include Applied Cryptography and Security in Cloud Computing. He has published over 50 research papers in refereed international conferences and journals and has served as the program chair or program committee member in many international conferences.

Xiaofeng Chen received his BS and MS degrees in mathematics from Northwest University, China. He got his PhD degree in cryptography from Xidian University, in 2003. Currently, he works at Xidian University as a professor. His research interests include applied cryptography and cloud computing security. He has published over 80 research papers in refereed international conferences and journals. His work has been cited more than 1,000 times at Google Scholar. He has served as the program/general chair or program committee member in over 20 international conferences.

Mingqiang Li received the PhD degree (with honors) in computer science from Tsinghua University, in June 2011. He also received the BS degree in mathematics from the University of Electronic Science and Technology of China, in July 2006. He worked as a Staff Researcher at the IBM China Research Laboratory from July 2011 to February 2013. He is now a Postdoctoral Fellow at the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His current research interests include storage systems, data reliability, data security, data compression, virtual machines, distributed systems, and wireless networking.

Jingwei Li received his BS degree in mathematics, in 2005 from the Hebei University of Technology, China. He is currently a visiting student (sponsored by The State Scholarship Fund of China) in Penn State University, USA. Besides, he is a PhD candidate in computer technology in Nankai University. His research interests include applied cryptography, cloud security.



Patrick P.C. Lee received the BEng degree (first-class honors) in information engineering from the Chinese University of Hong Kong, in 2001, the MPhil degree in computer science and engineering from the Chinese University of Hong Kong, in 2003, and the PhD degree in computer science from Columbia University, USA, in 2008. He is now an Assistant Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in cloud storage, distributed systems and networks, and security/resilience.

Wenjing Lou received the BS and MS degrees in computer science and engineering from the Xián Jiaotong University, China, the MASc degree in computer communications from the Nanyang Technological University, Singapore, and the PhD degree in electrical and computer engineering from the University of Florida, USA. She is now an Associate Professor in the Computer Science Department at Virginia Polytechnic Institute and State University, USA.