Balancing Storage Efficiency and Data Confidentiality with Tunable Encrypted Deduplication

Jingwei Li[1,2], Zuoru Yang[3], Yanjing Ren[1], Patrick P. C. Lee[3], and Xiaosong Zhang[1]
[1]Center for Cyber Security, University of Electronic Science and Technology of China
[2]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
[3]Department of Computer Science and Engineering, The Chinese University of Hong Kong

Abstract

Conventional encrypted deduplication approaches retain the deduplication capability on duplicate chunks after encryption by always deriving the key for encryption/decryption from the chunk content, but such a deterministic nature causes information leakage due to frequency analysis. We present TED, a tunable encrypted deduplication primitive that provides a tunable mechanism for balancing the trade-off between storage efficiency and data confidentiality. The core idea of TED is that its key derivation is based on not only the chunk content but also the number of duplicate chunk copies, such that duplicate chunks are encrypted by distinct keys in a controlled manner. In particular, TED allows users to configure a storage blowup factor, under which the information leakage quantified by an information-theoretic measure is minimized for any input workload. We implement an encrypted deduplication prototype TEDStore to realize TED in networked environments. Evaluation on real-world file system snapshots shows that TED effectively balances the trade-off between storage efficiency and data confidentiality, with small performance overhead.

*CCS Concepts• **Information systems → Cloud based storage**; **Deduplication**.

ACM Reference Format:
Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick P. C. Lee, and Xiaosong Zhang. 2020. Balancing Storage Efficiency and Data Confidentiality with Tunable Encrypted Deduplication. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387531

EuroSys'20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387531

1 Introduction

Outsourcing storage management to the cloud is appealing to enterprises and individuals to cope with the unprecedented growth of data in the wild [34]. Practical storage outsourcing solutions should fulfill two goals: (i) *storage efficiency*, which consumes the least possible storage footprints to save outsourcing costs, and (ii) *data confidentiality*, which protects outsourced storage from the unauthorized access by malicious users or even the cloud providers that host the outsourcing services.

To achieve both goals, we explore *encrypted deduplication* for outsourced storage. Deduplication is a popular data reduction technique to achieve storage efficiency. It removes duplicate data at the granularity of *chunks* and keeps only one physical copy of all duplicate chunks. Encrypted deduplication further augments deduplication with encryption, such that its goal is to transform the original pre-deduplicated chunks (called *plaintext chunks*) into the encrypted chunks (called *ciphertext chunks*) that will be kept in deduplicated storage. However, conventional *symmetric-key encryption (SKE)* is incompatible with deduplication, as it uses a distinct key (e.g., obtained via random key generation) for encryption/decryption. This causes duplicate plaintext chunks to be encrypted into distinct ciphertext chunks due to distinct keys, thereby prohibiting deduplication on the ciphertext chunks. Bellare *et al.* [16] propose a cryptographic primitive called *message-locked encryption (MLE)* to formalize the key derivation in encrypted deduplication, in which each plaintext chunk is encrypted by a key derived from the chunk content, so that duplicate plaintext chunks are encrypted into identical ciphertext chunks for deduplication. Examples of MLE constructions include convergent encryption [26] and server-aided MLE [15] (see details in §2.1).

However, existing MLE constructions remain vulnerable to information leakage, as they build on *deterministic encryption* to always map duplicate plaintext chunks into identical ciphertext chunks through content-based key derivation; this is in contrast to SKE, in which a plaintext chunk is mapped to a distinct ciphertext chunk subject to a distinct key. The deterministic nature of MLE inevitably leaks the *frequency* (i.e., number of duplicate chunk copies) distribution of the plaintext chunks, making encrypted deduplication vulnerable to the *frequency analysis* attack [44] that examines the ciphertext chunks and infers their original plaintext chunks;

hence, data confidentiality cannot be fully achieved.

Thus, encrypted deduplication poses a dilemma in choosing a proper cryptographic primitive: MLE achieves storage efficiency via deduplication but introduces frequency leakage due to its deterministic nature, while SKE is robust against frequency leakage but prohibits deduplication. Some existing approaches resolve the dilemma to some extent, but they rely on either expensive cryptographic primitives that are impractical, or simple heuristics with limited protection and configurability guarantees (§2.4). To our knowledge, there is no rigorous treatment in the literature of the trade-off between storage efficiency and data confidentiality in encrypted deduplication, and this motivates our work.

We present TED, a cryptographic primitive for enabling **t**unable **e**ncrypted **d**eduplication in outsourced storage. TED provides a tunable mechanism that allows users to balance the trade-off between storage efficiency and data confidentiality. Its core idea is to augment the key derivation in MLE, such that the key used for encrypting/decrypting a chunk is derived from not only the chunk content but also the chunk frequency, so as to allow duplicate plaintext chunks to be encrypted by distinct keys (i.e., relaxing the deterministic encryption nature). To achieve storage efficiency, TED derives a distinct key only when the chunk frequency increases (i.e., more duplicates accumulate) by some factor, so that a large portion of duplicate plaintext chunks are still encrypted into identical ciphertext chunks to maintain the deduplication effectiveness.

Clearly, TED introduces a storage blowup over *exact deduplication* (i.e., all duplicates are removed by deduplication). Nevertheless, by limiting a small storage blowup, TED still maintains high storage savings via "near-exact" deduplication. For example, practical backup workloads under deduplication can achieve a storage saving of 90% (or a 10× deduplication ratio) [66]. If we limit the storage blowup to 20% over exact deduplication, then the storage saving reduces to 88% (or an 8.3× deduplication ratio), which remains significant.

To realize the idea of tuning the storage-confidentiality trade-off in encrypted deduplication, TED builds on three key design techniques:

- *Sketch-based frequency counting*, which leverages a compact data summary structure called *sketch* [23] to estimate the frequencies of all chunks with bounded errors. Using a sketch not only reduces the memory usage for frequency counting, but also protects the chunk identities during frequency counting.
- *Probabilistic key generation*, which non-deterministically derives keys for duplicate plaintext chunks from a candidate set of keys to create distinct sequences of ciphertext chunks, while preserving the deduplication effectiveness. This avoids encrypting identical files into the same sequence of ciphertext chunks, and hence protects the information leakage of identical files.
- *Automated parameter configuration*, which formulates the parameter configuration problem as an optimization problem that minimizes the information leakage for an input workload subject to a configurable storage blowup factor over exact deduplication; here, the leakage is quantified by the information-theoretic measure *Kullback-Leibler distance (KLD)* (or relative entropy) [40] with respect to the uniform distribution of chunk frequencies. This allows users to readily configure a storage blowup factor based on their affordable storage overhead, instead of tuning any non-intuitive system-level parameter for balancing the storage-confidentiality trade-off for different workloads.

We implement a proof-of-concept encrypted deduplication prototype called TEDStore that realizes TED for outsourced storage applications. We conduct extensive trace-driven evaluation on both TED and TEDStore using two real-world datasets of file system snapshots [1, 52]. Compared to the two baseline primitives SKE and MLE, TED reduces the KLD of MLE by up to 84.7% with a configurable storage blowup factor of 1.2 (i.e., 20% more storage space over exact deduplication), while achieving high storage savings over SKE. We also show that the configurable storage blowup factor matches well the actual storage blowup. Finally, TEDStore achieves high upload/download performance in networked storage, while TED only incurs small overhead and is not the performance bottleneck in TEDStore.

We now release the source code of both TED and TEDStore at **http://adslab.cse.cuhk.edu.hk/software/ted**.

2 Problem and Motivation

We present the background on both deduplication and encrypted deduplication (§2.1). We show the encrypted deduplication architecture (§2.2) and describe the threat model (§2.3). Finally, we discuss the limitations of existing approaches in addressing the threat model (§2.4).

2.1 Basics

Deduplication. Deduplication is a coarse-grained compression technique that eliminates duplicate data copies in storage. Our work focuses on *chunk-based* deduplication, which divides file data into a sequence of variable-size chunks (e.g., 4-8 KB each) [28] and uniquely identifies each chunk by the cryptographic hash (called *fingerprint*) of the chunk content (the hash collision of two distinct chunks is highly unlikely in practice [19]). Only one physical copy of duplicate chunks is stored, while other duplicate chunks are stored as small-size pointers that refer to the physical chunk. Deduplication is shown to achieve huge storage savings in backups [47, 66, 72], virtual machine images [35], and file system snapshots [52, 65], and is adopted by commercial cloud services (e.g., Dropbox and Memopal) [32].

Encrypted deduplication. As described in §1, encrypted deduplication preserves the deduplication effectiveness on ci-

phertext chunks that are encrypted from duplicate plaintext chunks. Conventional encrypted deduplication approaches can be characterized via the symmetric-key encryption primitive called *message-locked encryption (MLE)* [16], which formalizes how the key of each chunk is derived from the chunk content for symmetric encryption/decryption. One well-known MLE construction is *convergent encryption (CE)* [26], in which the key of a chunk is set as the chunk fingerprint. CE has been realized and extensively evaluated in many systems (e.g., [5, 8, 24, 26, 37, 58, 64, 68]). However, CE is vulnerable to *offline brute-force attacks* [15], as an adversary can compute the fingerprints (via the cryptographic hash of the chunk content) for all candidate plaintext chunks in a brute-force manner and check if a chunk is encrypted into any existing ciphertext chunk. Thus, the security of MLE builds on the assumption that the chunks are derived from an *unpredictable* message space, so that offline brute-force attacks become infeasible [15].

To defend against offline brute-force attacks for the chunks derived from a *predictable* message space, DupLESS [15] realizes *server-aided MLE*, in which the key generation is controlled by a dedicated *key manager*. Specifically, a client requests the key of a chunk from the key manager by submitting the chunk fingerprint. The key manager then derives the key not only on the chunk fingerprint, but also on a *global secret* owned by the key manager. If the global secret is secure, an adversary cannot feasibly compute the keys of all candidate plaintext chunks; even if the global secret is compromised, the security of server-aided MLE reduces to that of the original MLE. To further secure the key generation process, DupLESS proposes two mechanisms. First, when a client requests the secret key of a chunk, it submits a "blinded" fingerprint via the *oblivious pseudo-random function (OPRF)* [54] to the key manager, such that the key manager can return the same key for identical fingerprints, yet it does not know the original fingerprint. Second, the key manager rate-limits the key generation requests to protect against online brute-force attacks by malicious clients that attempt to issue many key generation requests.

Frequency analysis. Both the original MLE [16] and server-aided MLE [15] build on *deterministic encryption*, meaning that each plaintext chunk is always mapped to a ciphertext chunk. It inevitably leaks the frequency (i.e., number of duplicate chunk copies) of each chunk, thereby making encrypted deduplication vulnerable to frequency analysis.

To launch frequency analysis against encrypted deduplication, an adversary first accesses an *auxiliary dataset* [55]; for example, the auxiliary dataset can refer to the plaintext chunks of some prior versions of backups [44]. Previous studies show that the auxiliary dataset can be obtained via public data releases [30, 55], security breaches [18], or storage device theft [33]. The adversary also accesses a set of ciphertext chunks as the attack object (e.g., the latest version of backups [44]). It then ranks the plaintext chunks and ciphertext chunks separately by their respective frequencies. Finally, it reverts each ciphertext chunk to the plaintext chunk in the same frequency rank.

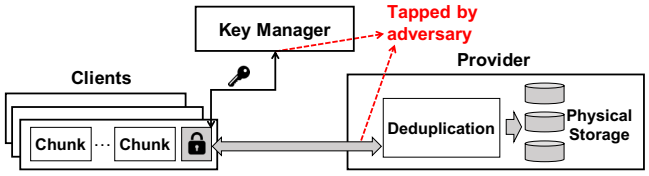


Figure 1. An encrypted deduplication architecture. An adversary may have access to the key manager and the provider to monitor the behaviors of their operations (§2.3).

Frequency analysis is a historically well-known cryptanalysis attack [6]. It is recently shown to cause substantial information leakage in encrypted databases [18, 30, 55] as well as encrypted deduplication [44]. Our goal is to achieve data confidentiality against frequency analysis.

2.2 Architecture

Our work builds on the server-aided MLE [15] architecture (Figure 1) with three entities: (i) multiple *clients*, which provide interfaces for applications to access file data under encrypted deduplication; (ii) the *key manager*, which performs key generation for each client; and (iii) the *storage provider* (or *provider* in short), which provides outsourced deduplicated storage. To prevent side-channel leakage (e.g., a malicious client can infer the existence of a chunk by checking if the chunk can be deduplicated) [32, 53], we perform deduplication in the provider, so that malicious clients cannot infer the deduplication patterns via client-side deduplication [32].

To upload a file, a client divides file data into chunks. It generates the key for each chunk via the interaction with the key manager, encrypts each chunk by the corresponding key, and uploads the chunk to the provider. In addition, for file reconstruction, the client generates a *file recipe* that lists the chunk fingerprints and the chunk sizes based on the chunk order in the file, as well as a *key recipe* that keeps the keys for all chunks. It encrypts both the file recipe and the key recipe with a per-client master key for protection, and uploads them with the ciphertext chunks to the provider. The provider performs deduplication on the ciphertext chunks. It maintains a *fingerprint index*, a key-value store that tracks the fingerprints of physical chunks for duplicate detection. Note that the provider does not apply deduplication to metadata; instead, it directly keeps both file recipes and key recipes (in encrypted forms) in physical storage.

To download a file, the client first retrieves both the file recipe and the key recipe from the provider, and decrypts them with its master key. It then retrieves the ciphertext chunks from the provider based on the file recipe, and decrypts them with the keys stored in the key recipe.

2.3 Threat Model

Adversarial capabilities. We consider an honest-but-curious (i.e., no modification to the system protocol) but *knowledgeable* adversary that has the access to some auxiliary datasets and knows the frequency distribution of plaintext chunks. The adversary aims to identify the original content of the ciphertext chunks in remote storage by tapping into the entry points of both the provider and the key manager (Figure 1):

- It has the access to the provider and eavesdrops the ciphertext chunks being written to the provider before deduplication, so as to learn the frequency of each ciphertext chunk and launch frequency analysis.
- It has the access to the key manager and eavesdrops both the key generation requests sent from the clients and the replies from the key manager. If it learns the global secret of the key manager, the security reduces to that of the original MLE (§2.1).

Adversarial assumptions. Our threat model makes the following assumptions: (i) the communication channels among the clients, the key manager, and the provider are protected by SSL/TLS against tampering; (ii) the key manager rate-limits each client's key generation requests, so as to defend against online brute-force attacks [15] (§2.1); (iii) both the file recipe and key recipe of each file are secure as they are protected by each client's master key (§2.2); and (iv) in order to ensure data availability, we can deploy remote auditing [12, 36] for data integrity, as well as deduplication-aware secret sharing [45] across multiple storage sites for fault tolerance.

2.4 Limitations of Existing Approaches

Several encrypted deduplication designs can defend against frequency analysis. Here, we review four such designs.

- *Random MLE* [4]: It encrypts each plaintext chunk with a random key. To support deduplication, it attaches each ciphertext chunk with a (random) payload for detecting if the corresponding underlying plaintext chunk is identical via the *non-interactive zero knowledge (NIZK) proofs*.
- *Interactive MLE* [14]: It also encrypts each plaintext chunk with a random key as in random MLE. To support deduplication, it leverages *fully homomorphic encryption (FHE)* to implement an evaluation function for checking if the ciphertext chunks are derived from duplicate plaintext chunks without decrypting the ciphertext chunks.
- *Layered encryption* [63]: It first encrypts each plaintext chunk with MLE, and then encrypts the resulting ciphertext chunk with the *threshold public-key cryptosystem*, such that the decryption key for the ciphertext chunk is transformed into multiple random shares that are sent to the provider. When the provider receives a threshold number of shares, it can rebuild the decryption key, recover the ciphertext chunk, and perform deduplication as in MLE.
- *MinHash encryption* [44]: It groups multiple consecutive plaintext chunks into *segments*. For each segment, it derives a key as the minimum fingerprint of all chunks in the segment, and encrypts all the chunks with the same key. For backup workloads, the segments are often similar with a large fraction of duplicate plaintext chunks [17], so the keys (i.e., the minimum fingerprints) for similar segments are likely the same (due to Broder's theorem [21]). Thus, most duplicate chunks are encrypted by the same key, making deduplication viable after encryption.

Random MLE, interactive MLE, and layered encryption provide *semantic security* [38] guarantees for plaintext chunks, as the encryption is no longer deterministic (i.e., the same plaintext chunk is encrypted to some "random" outputs). For MinHash encryption, although similar segments likely have the same key, a small fraction of duplicate plaintext chunks in different segments may still be encrypted by different keys. This alters the frequency ranking of ciphertext chunks, and is empirically shown to mitigate the severity of frequency analysis [44]. However, the above designs still face several practical limitations.

- *Limitation 1: Expensive cryptographic primitives.* Random MLE and interactive MLE build on expensive primitives (i.e., NIZK proofs and FHE, respectively) that are theoretically proven but are not readily implemented in practice. Layered encryption builds on the threshold public-key cryptosystem, which is less efficient than the symmetric key cryptosystem when encrypting large data.
- *Limitation 2: Limited protection.* MinHash encryption builds on the file similarity assumption [17] for its deduplication effectiveness, so its storage efficiency may not hold for general workloads. More importantly, the minimum chunk fingerprints in segments have limited randomness (otherwise, the deduplication effectiveness will be lost), so MinHash encryption only slightly breaks the deterministic nature of MLE and provides no security guarantees against frequency analysis.
- *Limitation 3: Limited configurability.* All the designs do not provide a configurable mechanism to quantify the trade-off between storage efficiency and the resilience against frequency analysis. For example, MinHash encryption disturbs the frequency ranking of ciphertext chunks by sacrificing storage efficiency (e.g., the duplicate plaintext chunks in different segments are encrypted by different keys and cannot be deduplicated after encryption). However, it derives the keys purely from the chunk content (i.e., the minimum fingerprints of segments), and cannot control how much storage efficiency is lost.

Some defense approaches (e.g., [39, 41, 42]) protect encrypted databases against frequency analysis. However, they are not applicable to encrypted deduplication (§6).

3 TED Design

3.1 Design Goals

TED is an encrypted deduplication primitive that aims for the following design goals.

- *Storage efficiency.* TED applies deduplication to remove duplicate chunks from storage.
- *Data confidentiality.* TED protects deduplicated storage from unauthorized access through encryption. It also mitigates the information leakage due to frequency analysis.
- *Configurability.* TED allows a tunable trade-off between storage efficiency and data confidentiality, such that the information leakage is minimized subject to a configurable storage blowup factor.
- *Low performance overhead.* TED incurs low performance overhead in encrypted deduplication deployment.

Application scenario. TED mainly targets backup workloads, which have high degrees of content duplicates that can be effectively removed via deduplication [7, 66]. It is applicable for an organization that plans to securely outsource the storage management for its clients to a third-party cloud storage provider. The organization maintains a key manager for the key management of its clients and configures the storage-confidentiality trade-off subject to an affordable storage blowup factor. It also deploys a virtual machine or container instance in the cloud to perform deduplication on the data uploaded by clients for storage savings. Such a deployment allows the organization to achieve secure and space-efficient outsourced storage.

3.2 Design Overview

TED's principle is to derive the key of each plaintext chunk (denoted by M) based on two additional inputs in addition to its content: (i) its *current frequency* f, which specifies the number of duplicate copies of M that have been uploaded by all clients, and (ii) the *balance parameter* t, which controls the trade-off between frequency protection and deduplication effectiveness. The key, denoted by K, of M is generated by the key manager (§2.2) as follows:

$$K = \mathbf{H}(\kappa \parallel P \parallel \lfloor f/t \rfloor), \tag{1}$$

where $\mathbf{H}(\cdot)$ is a cryptographic hash function, κ is the global secret owned by the key manager, P is the fingerprint of M (derived from the chunk content of M), $\parallel$ is the concatenation operator, and $\lfloor f/t \rfloor$ is the maximum integer smaller than f/t.

Note that f is cumulative and increases as more duplicates are detected. The key K will be updated as the integer $\lfloor f/t \rfloor$ increases. Thus, the duplicates of M will be encrypted by different keys in general depending on the value of t. If $t = 1$, each duplicate of M has a distinct K and TED reduces to SKE; if $t \to \infty$, all duplicates of M have the identical K and TED reduces to MLE. Intuitively, t can be viewed as the *maximum number of duplicate copies for a ciphertext chunk*.

However, realizing TED's idea is not trivial. We pose three challenges, which we address in the following subsections.

- *Q1: How does the key manager obtain the frequencies of all chunks?* The key manager needs to know the current frequency of each chunk for key generation. A challenge is that given the huge number of chunks being processed, the frequency counting in the key manager must incur low overhead. Another challenge is that if the key manager uses blinded key generation as in DupLESS [15], in which the blinded fingerprints look like "random" values to the key manager (§2.1), it cannot infer the original fingerprints to count the chunk frequencies by fingerprints.
- *Q2: How should the key manager generate the key of each chunk?* The key generation in Equation (1), unfortunately, raises a security issue: for identical files with the same sequence of chunks, Equation (1) will return the same keys that lead to also the same sequence of ciphertext chunks, thereby allowing an adversary to infer if two encrypted files are originally identical. Thus, the key generation must produce distinct sequences of ciphertext chunks for identical files, while preserving the deduplication effectiveness.
- *Q3: How should the balance parameter be configured?* The balance parameter t determines the storage blowup over exact deduplication. However, the same value of t may lead to significantly different storage blowups across workloads. Thus, it is critical to automatically configure t to make the actual storage blowup controllable for different workloads.

3.3 Sketch-based Frequency Counting

To address Q1, TED implements the *Count-Min Sketch (CM-Sketch)* [23] in the key manager for the frequency estimation of each chunk with fixed-size memory usage. A CM-Sketch is composed of a two-dimensional array with r rows of w counters each. We configure r pairwise independent hash functions $\{h_i(\cdot)\}_{i=1}^{r}$, such that each hash function h_i maps a chunk to a counter (indexed from 1 to w) in row i ($1 \le i \le r$). For each input chunk, we increment each of its hashed counters by one. To recover the frequency of a chunk, we use the minimum value of the r hashed counters as an estimate. Given r and w, the estimated frequency provably incurs a bounded error with a high probability [23]. For example, our trade-off analysis (§5.2) by default sets $r = 4$ with $w = 2^{25}$ 4-byte counters each, so the memory usage is 512 MB. Also, the estimation error is bounded within $n \times e/2^{25}$ with a probability of at least $1 - 1/4^e$, where e is Euler's number and n is the total number of chunks being counted [23].

In TED, for each plaintext chunk M, a client sends the hashes $\{h_1(M), h_2(M), \ldots, h_r(M)\}$ to the key manager, which updates the CM-Sketch accordingly. The key manager estimates the current frequency of M and uses the estimated frequency to derive the key.

The advantages of using the CM-Sketch are two-fold. First, it limits the memory usage for tracking the frequencies of

all chunks, while the errors are provably bounded. Second, the approximate counting protects the chunk information from the key manager, which is a security requirement in DupLESS [15] (§2.1). Each $h_i(\cdot)$ is a *short* hash function that returns a counter index ranging from 1 to w. Since w is generally small compared to the range of fingerprint values, each $h_i(\cdot)$ leads to many hash collisions (i.e., multiple chunks are mapped to the same short hashes). Thus, the key manager cannot readily guess a chunk from the short hashes.

3.4 Probabilistic Key Generation

To address Q2, TED realizes a probabilistic key generation approach that can encrypt identical files (with the same sequence of plaintext chunks) non-deterministically into distinct sequences of ciphertext chunks, while preserving the deduplication effectiveness.

Our insight is to randomly select the key for a chunk from a set of candidates, instead of returning the same key as in Equation (1). Specifically, for each plaintext chunk M, let $x = \lfloor f/t \rfloor$, where f is the current frequency of M and t is the balance parameter. Upon receiving the hashes of M, the key manager computes a *key seed candidate* k_x as:

$$k_x = \mathbf{H}(\kappa \parallel h_1(M) \parallel h_2(M) \ldots \parallel h_r(M) \parallel x). \tag{2}$$

It then uniformly selects a key seed from the candidate set $\{k_0, k_1, \ldots, k_x\}$:

$$k \xleftarrow{\$} \{k_0, k_1, \ldots, k_{x-1}, k_x\}. \tag{3}$$

The client finally derives the key of M as

$$K = \mathbf{H}(k \parallel P), \tag{4}$$

where P is the fingerprint of M. Note that TED does not use k as the key of M in order to prevent the key manager, as well as an adversary that can eavesdrop the replies of the key manager (§2.3), from directly accessing the keys.

Intuitively, as we observe more duplicates of M (i.e., f increases), the recent duplicates of M are encrypted based on some of the old key seeds from $\{k_0, k_1, \ldots, k_x\}$ that have been used before. Thus, TED maintains the deduplication effectiveness by allowing some duplicates to be protected by the same key seed. Meanwhile, the generation of ciphertext chunks is non-deterministic, as they are derived from randomly selected key seeds (as opposed to the deterministic key generation in Equation (1)). In general, the plaintext chunks with higher frequencies will be encrypted to a more diverse set of ciphertext chunks, as more candidate key seeds can be chosen as f increases.

3.5 Automated Parameter Configuration

To address Q3, instead of letting users directly configure the balance parameter t, which is a system-level parameter that is less intuitive for general users to choose for different workloads, TED automatically configures t by solving an optimization problem for an input workload subject to the affordable storage overhead specified by users. Specifically, users can indicate the storage overhead over exact deduplication via a *storage blowup factor* b, defined as the ratio between the physical storage size due to TED and that due to exact deduplication. Typically, $b \geq 1$; if $b = 1$, then TED reduces to MLE (which supports exact deduplication). The optimization goal of TED is to minimize the information leakage for an input workload subject to the configurable parameter b, and identify the corresponding t.

Optimization problem. Here, we use the number of chunks as an approximation of the physical storage size. Specifically, let n be the total number of unique plaintext chunks $\{M_i\}_{i=1}^{n}$, such that each (unique) plaintext chunk M_i has a frequency f_i (i.e., the number of duplicates of M_i). Without loss of generality, let $f_1 \leq \ldots \leq f_n$. Let n^* be the total number of unique ciphertext chunks $\{C_i\}_{i=1}^{n^*}$, where $n^* = n \times b$ (assuming that both ciphertext and plaintext chunks have the same average chunk size), such that each (unique) ciphertext chunk C_i has a frequency f_i^*. Each duplicate copy of plaintext chunk M_i is encrypted into the ciphertext chunk C_i (for $1 \leq i \leq n$) or another ciphertext chunk C_j for some $n+1 \leq j \leq n^*$. In other words, a plaintext chunk is not always mapped to the same ciphertext chunk as in MLE, as it may also be mapped to some other ciphertext chunks to make the encryption non-deterministic.

We leverage the information-theoretic measure *Kullback-Leibler distance (KLD)* [40] to characterize how the frequency distribution of the ciphertext chunks differs from the uniform distribution (i.e., how well TED is secure against frequency leakage); note that KLD is also used to measure the frequency leakage in encrypted databases [41]. Let $p_i^* = f_i^* / \sum_{i=1}^{n^*} f_i^*$ be the probability density function corresponding to f_i^*. Then the KLD of the frequency distribution of ciphertext chunks (with respect to the uniform distribution) is:

$$KLD = \sum_{i=1}^{n^*} p_i^* \log \frac{p_i^*}{1/n^*} = \log n^* + \sum_{i=1}^{n^*} p_i^* \log p_i^*. \tag{5}$$

A smaller KLD (whose minimum is zero) implies that the frequency distribution of the ciphertext chunks is closer to the uniform distribution (i.e., less frequency leakage).

Our goal is to find $\{f_i^*\}_{i=1}^{n^*}$ by solving the following optimization problem:

$$\begin{aligned} &\text{minimize } KLD \\ &\text{subject to } \sum_{i=1}^{n^*} f_i^* = \sum_{i=1}^{n} f_i, \\ &\qquad 0 \leq f_i^* \leq f_i \ \forall i \in [1, n], \\ &\qquad f_i, f_i^* \text{ are integers } \forall i \in [1, n]. \end{aligned} \tag{6}$$

The constraints ensure that the total frequency of all ciphertext chunks is preserved as that of all plaintext chunks, the frequency of each plaintext chunk M_i is no less than that of the corresponding ciphertext chunk C_i (as M_i may be mapped to multiple distinct ciphertext chunks), and the frequencies are integers.

Optimization solution. Since $\{f_i^*\}_{i=1}^{n^*}$ are integers, the optimization problem is an *mixed integer non-linear optimization* problem, which is known to be NP-hard [56]. Thus, we relax the integer constraints by allowing $\{f_i^*\}_{i=1}^{n^*}$ to be floating-point numbers, and round the results into integers. With the relaxations, the problem becomes a *convex optimization* problem. We can show that the optimal solution (which can be found based on the simplex algorithm [20]) is:

$$\begin{cases} f_i^* = f_i, & 1 \le i \le m, \\ f_i^* = \frac{\sum_{j=m+1}^{n} f_j}{n^*-m}, & m+1 \le i \le n^*, \end{cases} \tag{7}$$

where m is the maximum integer such that $f_m \le \frac{\sum_{j=m+1}^{n} f_j}{n^*-m}$. Since $f_1 \le \ldots \le f_n$, we also ensure that $f_1^* \le f_2^* \le \ldots f_{n^*}^*$. Intuitively, the optimal solution caps the frequencies of the top-frequent ciphertext chunks, so the frequency distribution of the ciphertext chunks is close to the uniform distribution.

Since t controls the maximum number of duplicate copies among all ciphertext chunks (§3.2), we configure t as the maximum frequency in $\{f_i^*\}_{i=1}^{n^*}$ to approximate the frequency distribution of the ciphertext chunks as shown in Equation (7):

$$t = \left\lceil \frac{\sum_{j=m+1}^{n} f_j}{n^* - m} \right\rceil. \tag{8}$$

Configuring t in practice. In TED, the key manager solves the optimization problem and obtains t for key seed generation (§3.4). Note that the optimization solution depends on the frequency distribution of plaintext chunks. While including *all* plaintext chunks in frequency counting returns an accurate frequency distribution, it inevitably incurs a long processing delay, since a client cannot start the chunk encryption until the key manager finishes frequency counting and returns the key seeds.

Thus, we periodically update t based on the frequency distribution of a *batch* of plaintext chunks. Specifically, we initialize $t = 1$. A client issues the key generation requests for the plaintext chunks on a per-batch basis, and the key manager solves the optimization problem and updates t for each batch. Once the client receives the key seeds for a batch of chunks from the key manager, it derives the keys for the chunks and performs chunk encryption, and in the meantime, it issues the key generation requests for the next batch of chunks. Thus, a client can perform both key generation and chunk encryption in parallel. The batch size is configurable; choosing a larger batch size returns a more accurate frequency distribution, but delays chunk processing. By default, we set the batch size as 48,000, which implies that each update of t is based on around 0.37% of input data for a 100 GB file (assuming that the average chunk size is 8 KB).

3.6 Security Discussion

Finally, we discuss the security implications of TED via a quantitative analysis. Specifically, we quantify the frequency leakage of a set of ciphertext chunks by analyzing the likelihood that an adversary distinguishes the frequency distribution of the set of ciphertext chunks from a uniform distribution; if the likelihood is low, we argue that the frequency leakage is limited. We consider an adversary that accesses a number of sampled ciphertext chunks that are chosen from either a frequency distribution derived from an encryption scheme (e.g., SKE, MLE, MinHash encryption [44], or TED) or a uniform distribution; however, the adversary does not know exactly which distribution is used. The adversarial goal is to guess the distribution from which the sampled ciphertext chunks are chosen. The more sampled ciphertext chunks that the adversary can access, the more likely it is to successfully guess the correct distribution. The success probability of the guess can be approximated as [13]:

$$P \approx 1 - \Phi(-\frac{\sqrt{2S \times KLD}}{2}), \tag{9}$$

where S is the number of sampled ciphertext chunks, and $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution. If the KLD is zero (e.g., in SKE), then the success probability is approximately 0.5, implying that the adversary has no advantage over a random guess.

In general, no encrypted deduplication scheme (including MLE, MinHash encryption, and TED) can suppress the success probability P as low as in SKE without sacrificing the deduplication effectiveness. Nevertheless, TED effectively reduces the KLD (with respect to the uniform distribution) and hence increases the difficulty for the adversary to correctly guess the frequency distribution of ciphertext chunks. For example, referring to Experiment A.1 in §5.2, if we set the storage blowup factor $b = 1.2$, then MLE, MinHash encryption, and TED have a KLD of 1.72, 1.35, and 0.26, respectively, implying that the adversary needs to access $1.72/0.26 \approx 6.6\times$ and $1.35/0.26 \approx 5.2\times$ sampled ciphertext chunks to achieve the same success probability of the guess against TED when compared to MLE and MinHash encryption, respectively.

Our quantitative analysis provides one possible explanation of how TED is less vulnerable to frequency analysis than existing encrypted deduplication schemes (i.e., an adversary needs to access more sampled ciphertext chunks in TED to achieve the same attack effectiveness against MLE and MinHash encryption). However, it remains an open question of how to quantify an *acceptable* trade-off between storage efficiency and data confidentiality in real deployment; in other words, how users should configure a proper storage blowup factor in practical encrypted deduplication deployment to achieve a reasonable level of data confidentiality. We pose a more in-depth analysis of the security implications of TED as our future work.

4 Implementation

To show the applicability of TED, we built an encrypted deduplication prototype called TEDStore (based on Figure 1

in §2.2), which realizes TED for chunk encryption. TEDStore is written in C++ on Linux. It uses OpenSSL 1.1.1c [3] for cryptographic operations and MurmurHash3 [9] for the hash operations in key generation (§3.4). Our prototype contains around 4.5K lines of code.

Deduplication. Each client now implements content-defined chunking based on *Rabin fingerprinting* [60], which takes the minimum, average, and maximum chunk sizes as input (by default, we set them as 4 KB, 8 KB, and 16 KB, respectively) and computes a rolling hash over a window of chunks to identify chunk boundaries. TEDStore performs deduplication in the provider (§2.2). The provider implements the fingerprint index as a key-value store based on LevelDB [2] to map the fingerprint of each ciphertext chunk to the physical address where the ciphertext chunk is stored. To mitigate the disk access overhead, the provider packs the unique chunks (on the order of KB each) in fixed-size *containers* (on the order of MB each), such that the I/O operations are in units of containers [46]. We now fix the container size as 8 MB.

Key generation. Recall that a client derives the key for a chunk by sending r short hashes (now $r = 4$) to the key manager (§3.4). To efficiently generate the short hashes, the client computes a 128-bit hash (via Murmurhash3) and divides the hash result into four short hashes (i.e., only one hash computation). By default, we choose SHA-256 for $\mathbf{H}(.)$ and AES-256 for chunk encryption, i.e., the key manager generates the key seed via SHA-256 (Equation (2)), while the client derives the secret key of each chunk via SHA-256 (Equation (4)) and encrypts the chunk via AES-256.

Performance optimization. Our TEDStore prototype exploits simple performance optimization techniques. For example, it uses multi-threading, in which the client parallelizes the processing of chunking, encryption, and uploads via multiple threads, while the key manager and the provider serve the requests from multiple clients with different threads. It also combines the transmissions of multiple small-size data units (e.g., hashes in key generation and chunks in uploads/downloads) into a single transmission to mitigate network overhead.

Prototype limitations. Our TEDStore prototype currently focuses on only the deduplication of data chunks, but not metadata (e.g., file recipes [51]). Also, it does not address the fault tolerance of the key manager and the provider, yet we can implement a quorum-based design for key generation [27] and storage [45]. Finally, it focuses on confidentiality, and does not support fine-grained access control (e.g., attribute-based encryption [29]).

5 Evaluation

We conduct trace-driven evaluation to study the storage-confidentiality trade-off of TED (§5.2) and the performance of TEDStore in networked settings (§5.3). Our evaluation shows the following key findings:

- TED balances the trade-off between storage efficiency and data confidentiality, which are not achievable by SKE and MLE (§1). Compared to MinHash encryption (§2.4), it achieves *both* lower KLD and less storage blowup.
- TED maps the plaintext chunks with high frequencies into distinct ciphertext chunks in different encryption runs.
- TED automatically controls the actual storage blowup by the configurable storage blowup factor b.
- TED achieves approximately 30× key generation speedup over existing approaches. It also incurs limited performance overhead (e.g., only 7.2% of the overall upload time) when being deployed in TEDStore.

5.1 Datasets

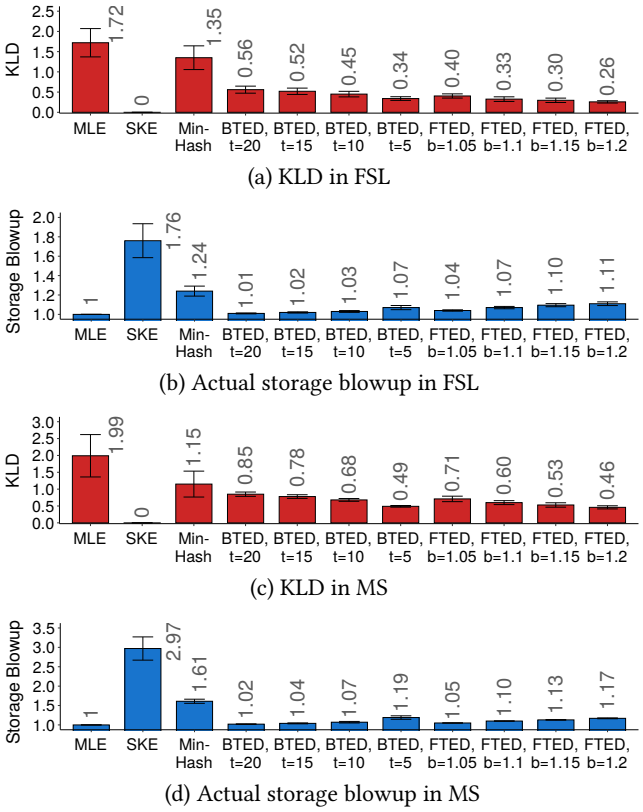
We consider two real-world file system snapshots.

- *FSL.* This dataset is collected by the File systems and Storage Lab (FSL) at Stony Brook University [1]. We choose the `fslhomes` snapshots taken from the home directories of different users on a shared file system. Each snapshot is represented as an ordered list of 48-bit chunk fingerprints obtained from content-defined chunking. We focus on the snapshots whose average chunk sizes are 8 KB. We sample a total of 42 snapshots from nine users over a span of January 22 to June 17 in 2013 (more precisely, on January 22, February 22, March 22, April 22, May 17, and June 17). The snapshot sizes vary significantly from 2.73 GB to 251.01 GB. Our dataset covers a total of 3.08 TB of pre-deduplicated data. The data size reduces to 1.54 TB if we perform deduplication on each snapshot.
- *MS.* This dataset contains the Windows file system snapshots collected by Microsoft [52]. Each snapshot is represented as an ordered list of 40-bit chunk fingerprints obtained from content-defined chunking. We focus on the snapshots whose average chunk sizes are 8 KB. We sample 30 snapshots, each of which is of size around 100 GB. Our dataset covers a total of 3.91 TB of pre-deduplicated data. The data size reduces to 1.34 TB if we perform deduplication on each snapshot.

5.2 Trade-off Analysis on TED

Setup. We consider two variants of TED: (i) *Basic TED (BTED)*, which chooses a fixed balance parameter t; and (ii) *Full TED (FTED)*, which automatically configures t for a given storage blowup factor b. Both variants employ sketch-based frequency counting and probabilistic key generation. By default, we fix $r = 4$ rows and $w = 2^{25}$ counters per row in the CM-Sketch for key generation (§3.4). For FTED, we disable batching in key generation (§3.5), such that t is derived from the frequencies of *all* plaintext chunks per snapshot (we evaluate the impact of batching in Experiment A.5).

Experiment A.1 (Overall analysis). We first analyze the overall trade-off of different encryption approaches, in terms of the KLD and the actual storage blowup over exact dedupli-



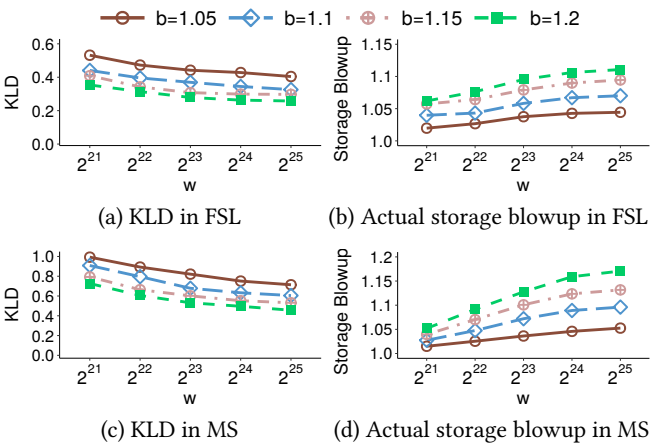
(a) KLD in FSL

(b) Actual storage blowup in FSL

(c) KLD in MS

(d) Actual storage blowup in MS

Figure 2. Experiment A.1: the storage-confidentiality trade-off for different encryption approaches in both FSL and MS datasets. Each error bar represents the variance (in terms of the 95% conference interval) across different snapshots in each encryption approach.



(a) KLD in FSL

(b) Actual storage blowup in FSL

(c) KLD in MS

(d) Actual storage blowup in MS

Figure 3. Experiment A.2: Analysis of sketch-based frequency counting. A larger w implies more accurate counting, at the expense of more memory usage.

cation for *each* snapshot in both FSL and MS datasets (i.e., we only apply deduplication to each snapshot, but not across all snapshots in a dataset). We compare five approaches: MLE, SKE, MinHash encryption, BTED, and FTED. For MinHash encryption, we fix its minimum, average, and maximum segment sizes as 512 KB, 1 MB, and 2 MB [44]; for BTED, we vary t; for FTED, we vary b.

Since our datasets represent the chunks by fingerprints and do not contain the actual content (§5.1), we simulate each encryption approach by treating each fingerprint as a plaintext chunk and deriving the key for the chunk according to the specific key derivation scheme. For MLE, the key is the SHA-256 hash of the fingerprint; for SKE, the key is a fresh random 32-byte string; for MinHash encryption, the key is the SHA-256 hash of the minimum fingerprint of the associated segment; for BTED and FTED, the key is computed from the frequency of the fingerprint. Given the derived key, we encrypt the fingerprint via AES-256 to obtain the ciphertext chunk.

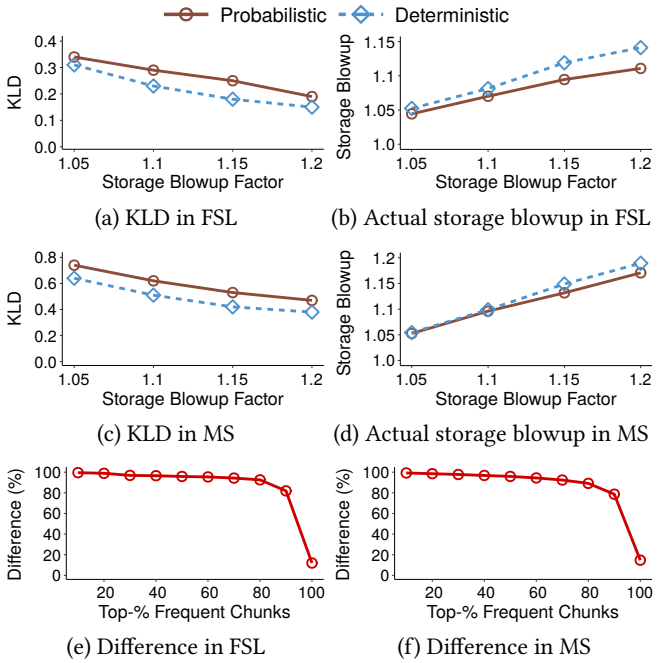
Figure 2 shows the average results over all snapshots in both FSL and MS datasets, with the 95% confidence intervals. MLE achieves exact deduplication (i.e., its actual storage blowup is always one), but incurs the highest KLD due to deterministic encryption. SKE has the minimum KLD (zero), but incurs a high actual storage blowup. MinHash encryption, BTED, and FTED realize the trade-off of KLD and storage blowup. For example, in the FSL (MS) dataset, FTED with $b = 1.2$ reduces the KLD of MLE by 84.7% (76.8%), and reduces the actual storage blowup of SKE by 37.0% (60.6%).

Both BTED and FTED achieve simultaneously less KLD and less storage blowup than MinHash encryption. In the FSL (MS) dataset, MinHash encryption has a KLD of 1.35 (1.15) with an actual storage blowup of 1.24 (1.61), while all BTED and FTED variants have a KLD below 0.56 (0.85) and an actual storage blowup at most 1.11 (1.17).

Comparing BTED and FTED, while BTED has a larger KLD and a smaller actual storage blowup for a larger t, and vice versa, its actual storage blowup cannot be readily configured with t. In contrast, FTED provides an effective way to control the actual storage blowup. As b increases from 1.05 to 1.2, the actual storage blowup increases from 1.04 to 1.11 in FSL and from 1.05 to 1.17 in MS. Note that the actual storage blowup in the FSL dataset is smaller than the given b when b is large (e.g., the actual storage blowup is 1.11 for $b = 1.2$), since some snapshots have very few duplicate chunks and their maximum achievable storage blowups over exact deduplication can be smaller than the given b.

Experiment A.2 (Analysis of sketch-based frequency counting). We evaluate how various CM-Sketch sizes affect the storage-confidentiality trade-off. We focus on FTED with varying b (from 1.05 to 1.2). For the CM-Sketch, we fix $r = 4$, and vary w from 2^{21} to 2^{25} (i.e., if the counter size is 4 bytes, the memory size varies from 32 MB to 512 MB).

Figure 3 shows the results. For all FTED variants, a smaller w implies a larger KLD and a smaller actual storage blowup. The reason is that a smaller w leads to larger over-estimates of chunk frequencies (due to more hash collisions in a counter), so FTED configures a larger t that leads to more identical

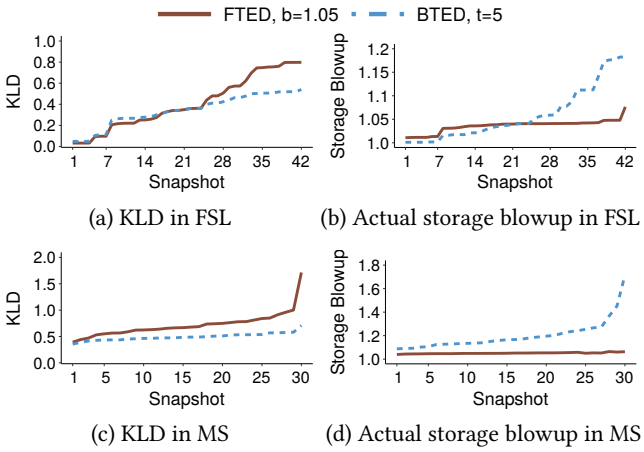


(a) KLD in FSL (b) Actual storage blowup in FSL

(c) KLD in MS (d) Actual storage blowup in MS

(e) Difference in FSL (f) Difference in MS

Figure 4. Experiment A.3: Analysis of probabilistic key generation, which we compare with deterministic key generation.



(a) KLD in FSL (b) Actual storage blowup in FSL

(c) KLD in MS (d) Actual storage blowup in MS

Figure 5. Experiment A.4: Comparison between BTED ($t = 5$) and FTED ($b = 1.05$) in the controllability of the actual storage blowup. Here, the x-axis refers to the snapshots sorted in the ascending order of their y-axis values.

ciphertext chunks for deduplication. For example, in the MS dataset, as w decreases from 2^{25} (our default) to 2^{21}, the actual storage blowup of FTED with $b = 1.2$ drops from 1.17 to 1.04, while the KLD increases from 0.46 to 0.73.

Experiment A.3 (Analysis of probabilistic key generation). We study the effect of probabilistic key generation. We compare it with a *deterministic* key generation approach, in which the client derives the key K by directly setting $k = k_x$ (see Equations (2) and (4) in §3.4). We focus on FTED with varying b (from 1.05 to 1.2).

Figures 4(a)-4(d) show the KLD and actual storage blowup results, averaged over all snapshots in both FSL and MS datasets. Probabilistic key generation has a slightly smaller actual storage blowup than deterministic key generation (by 0.9-2.8% in FSL and 0.7-1.6% in MS), mainly because it may choose some previously used key seeds for key generation and generate more duplicate ciphertext chunks for deduplication. The trade-off is that it has a higher KLD (by 9.6-26.7% in FSL and 15.6-26.2% in MS).

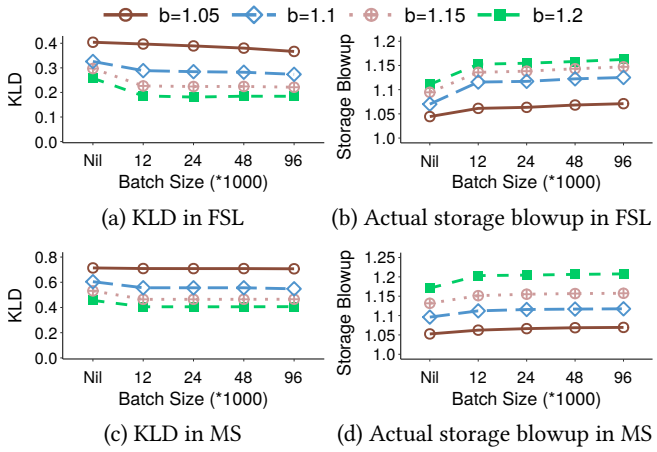
Nevertheless, probabilistic key generation adds randomness to the resulting ciphertext chunks. To show this effect, we encrypt each snapshot twice. We then measure the *difference rate* for each plaintext chunk, defined as the ratio between the number of distinct ciphertext chunks in two encryption runs and the number of duplicate copies for the plaintext chunk. For example, suppose that a plaintext chunk has four duplicate copies (M_1, M_2, M_3, M_4), which are encrypted into the ciphertext chunks (C_1, C_2, C_3, C_4) and (C_1, C_2', C_3', C_4') respectively in the two encryption runs (i.e., the last three ciphertext chunks are different). The difference rate is 75%. Note that for deterministic key generation, every plaintext chunk has a zero difference rate, as the keys across all encryption runs are identical. Also, if a plaintext chunk has only one unique copy, its difference rate is zero, as there is only one key seed to select. We focus on FTED with $b = 1.05$.

Figures 4(e) and 4(f) show the average difference rates for different top-percentiles of plaintext chunks, ranked by their frequencies, in both FSL and MS datasets. A plaintext chunk with a high frequency is more likely encrypted to a distinct ciphertext chunk (e.g., the top-80% of plaintext chunks have a difference rate of 89.2% in MS), since it has more key seed candidates and different key seeds are more likely to be chosen across encryption runs.

Experiment A.4 (Controllability of storage blowup). We study how TED controls the actual storage blowup via automated parameter configuration. We compare BTED and FTED, where we set $t = 5$ for BTED and $b = 1.05$ for FTED. The results are similar for other BTED and FTED variants.

Figure 5 shows the results. BTED incurs a larger variance of the actual storage blowup across the snapshots (from 1.00 to 1.18 in FSL and from 1.08 to 1.71 in MS). The reason is that the frequency characteristics of plaintext chunks are different across snapshots, and the same value of t cannot control the actual storage blowup to the same level for all snapshots. In contrast, FTED controls the actual storage blowup to around the pre-defined storage blowup factor $b = 1.05$ (from 1.02 to 1.07 in FSL and from 1.04 to 1.06 in MS), by automatically tuning t for each snapshot based on its frequency distribution of plaintext chunks.

Experiment A.5 (Impact of batch size). To efficiently configure t in practice, a client issues key generation requests



(a) KLD in FSL (b) Actual storage blowup in FSL

(c) KLD in MS (d) Actual storage blowup in MS

Figure 6. Experiment A.5: Impact of the batch size of key generation on KLD and the actual storage blowup; "Nil" means t is derived from the frequencies of all plaintext chunks per snapshot.

for a batch of plaintext chunks (§3.5). We study how batching affects the KLD and the actual storage blowup. Recall that with batching, TED initializes $t = 1$, and adjusts t on a per-batch basis. We focus on FTED with varying b.

Figure 6 shows the results versus the batch size (varied from 12,000 to 96,000) in both the FSL and MS datasets; for comparisons, we also consider the default case where the client issues the key generation requests for all plaintext chunks (labelled as "Nil"). Compared to the default case, batching has a slightly higher actual storage blowup; for example, for $b = 1.05$ and the batch size 12,000, the actual storage blowup is 1.06 in both FSL and MS datasets. Also, the actual storage blowup increases with the batch size; for example, for $b = 1.05$, it increases from 1.061 to 1.071 in FSL and from 1.062 to 1.069 in MS. The main reason is that TED initializes $t = 1$, so all duplicate plaintext chunks are encrypted to distinct ciphertext chunks, so the actual storage blowup is higher. Also, a larger batch size takes TED a longer time to increase t (a larger t implies more duplicate ciphertext chunks). Overall, the impact of the batch size remains limited compared to the default case, yet it allows t to be efficiently configured in practice (§3.5).

5.3 Performance Evaluation on TEDStore

We evaluate TEDStore in networked environments using both synthetic and real-world workloads. We analyze different performance aspects of TEDStore. By default, we choose the following parameters when realizing TED in TEDStore: (i) $b = 1.05$, (ii) $r = 4$ and $w = 2^{21}$ (i.e., 32 MB memory) for sketch-based frequency counting (§3.3), and (iii) a batch size of 48,000 chunks for key generation (§3.5).

5.3.1 Synthetic Workloads

We first evaluate TEDStore using synthetic workloads with only unique data in a testbed configured with one or multiple clients. We also remove the disk I/O overhead from our evaluation. Our goal is to understand the maximum achievable performance of TEDStore without the impact of deduplication and disk I/O, and show that TED accounts for limited overhead in TEDStore.

Methodology. We deploy TEDStore in a LAN testbed with multiple machines, each of which has a quad-core 3.4 GHz Intel Core i5-7500 CPU and 32 GB RAM. All machines are connected with 10 GbE, and run Linux Ubuntu 16.04.

We generate a synthetic dataset with a set of 2 GB files, each of which comprises globally unique chunks (i.e., no duplicates). We let one or multiple clients issue a 2 GB file of unique data to the provider as fast as possible. To avoid disk I/O, we load all data into each client's memory before each test, and let the provider store all received data in memory.

Experiment B.1 (Microbenchmarks). We start with microbenchmark evaluation by deploying a client, a key manager, and a provider all in a single machine, in which all entities are connected via the local loopback interface. We measure the computational time of different steps when the client uploads a 2 GB file of unique data to the provider. The steps include: (i) *chunking*, in which the client divides the file data into chunks; (ii) *fingerprinting*, in which the client computes the fingerprint of each chunk; (iii) *hashing*, in which the client computes the short hashes of each chunk; (iv) *key seeding*, in which the key manager performs frequency counting, solves the optimization problem, and returns the key seed for each chunk; (v) *key derivation*, in which the client derives the key of each chunk; and (vi) *encryption*, in which the client encrypts each chunk.

Since TEDStore's performance depends on the choices of its underlying cipher and hash functions, we consider two versions of cipher/hash algorithms: (i) *Fast*, which uses MD5 for fingerprinting and the hash function $\mathbf{H}(\cdot)$ (§3.4) as well as AES-128 for chunk encryption; and (ii) *Secure* (our default implementation), which uses SHA-256 and AES-256 instead.

Table 1 presents the breakdown of the computational time (per 1 MB of uploads) in both the fast and secure TEDStore versions. Fingerprinting and encryption are the most time-consuming steps, since they perform cryptographic operations on all file data. In contrast, the key generation under TED, including hashing, key seeding, and key derivation, takes only 7.2% and 6.1% of the overall computational time in the fast and secure versions, respectively. Thus, TED is not a performance bottleneck.

Experiment B.2 (Key generation performance). We evaluate the overall key generation performance of TEDStore (including hashing, key seeding, and key derivation, as described in Experiment B.1) in a networked setting. We compare it with two state-of-the-art blinded key generation protocols (§2.1): (i) *blind RSA* [15] and (ii) *blind BLS* [10]. We focus on a single-client case, and deploy the client and the key manager in two different machines connected with 10 GbE.

Steps	Fast	Secure
Chunking	0.8ms	
Fingerprinting	1.7ms	2.6ms
Hashing	0.4ms	
Key seeding	0.01ms	0.04ms
Key derivation	0.07ms	0.1ms
Encryption	3.7ms	4.9ms

Table 1. Experiment B.1: Breakdown of computational time per 1 MB of uploads in synthetic workloads.

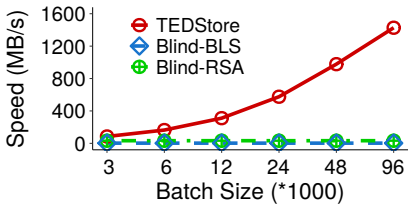


Figure 7. Experiment B.2: Key generation performance.

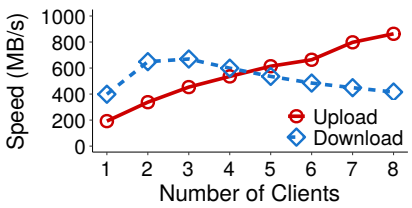


Figure 8. Experiment B.3: Multi-client performance.

For TEDStore, we vary the batch size in key generation (§3.5). We measure the *key generation speed* as the ratio of the file data size (i.e., 2 GB) to the total running time from when the client computes the hashes (in TEDStore) or blinded fingerprints (in blinded RSA and blinded BLS) for all chunks until it obtains all keys from the key manager.

Figure 7 shows the results versus the batch size (from 3,000 to 96,000); note that blind RSA and blind BLS do not consider parameter updates and their performance remains the same independent of the batch size. TEDStore achieves a much higher key generation speed than blind RSA and blind BLS, since it uses lightweight hash computations instead of expensive blinded key generation operations. For example, when the batch size is 48,000, TEDStore achieves a key generation speed of 997.4 MB/s, while those of blind RSA and blind BLS are only 32.5 MB/s and 2.3 MB/s, respectively (i.e., at least a 30× speedup in TEDStore). Also, the key generation speed of TEDStore increases with the batch size, as it solves the optimization problem fewer times.

Experiment B.3 (Multi-client performance). We evaluate the performance of TEDStore by having multiple clients upload/download data concurrently. Each client uploads a 2 GB file of unique data to the provider, and then downloads the 2 GB file from the provider. We issue the concurrent uploads/downloads of multiple clients at the same time. We measure the *aggregate upload (download) speed* as the ratio of the total uploaded (downloaded) data size to the total time all clients finish the uploads (downloads).

Figure 8 shows the results versus the number of clients (from one to eight). The aggregate upload speed increases with the number of clients and finally reaches 863.2 MB/s. On the other hand, the aggregate download speed first increases to 669.3 MB/s for three clients, followed by dropping to 415.5 MB/s for eight clients due to the read contention across multiple clients. We can improve the download performance by pre-fetching appropriate chunks [22, 46].

5.3.2 Real-World Workloads

We evaluate the performance of TEDStore when deduplication and disk I/O are in effect, using real-world workloads based on the large-scale datasets in §5.1.

Methodology. We deploy TEDStore in a LAN testbed with three machines. Specifically, we deploy a client on a machine with a 10-core 2.4 GHz Intel Xeon E5-2640v4 CPU and 64 GB RAM, a key manager on a machine with two six-core 2.4 GHz Intel Xeon E5-2620v3 CPUs and 32 GB RAM, and a provider on a machine with a 16-core 2.1 GHz Intel Xeon E5-2683v4 CPU and 64 GB RAM. The provider machine is attached with a RAID-5 array of four Western Digital Blue 4 TB 5400 rpm SATA harddisks.

Recall that our datasets only contain the fingerprints and sizes of chunks (§5.1). We reconstruct each chunk by repeatedly writing its fingerprints to a chunk of the specified size, so the same (distinct) fingerprint returns the same (distinct) chunk.

Experiment B.4 (Microbenchmarks). We conduct microbenchmarks on real-world workloads in a networked setting (as opposed to the single-machine setting in Experiment B.1). For each of the FSL and MS datasets, we choose the snapshot that has the medium size among all snapshots in the dataset. For FSL, the selected snapshot has 116.9 GB of pre-deduplicated data (or 13.4 M chunks), while for MS, the selected snapshot has 97.8 GB (or 16.2 M chunks) of pre-deduplicated data. We measure the upload time breakdown as in Experiment B.1 with two exceptions: (i) we do not include chunking due to our trace replay, and (ii) we add a *write* step, in which the client writes the pre-deduplicated ciphertext chunks to the provider, which performs deduplication and stores the unique ciphertext chunks on disk.

Table 2 presents the breakdown of the computational time (per 1 MB of uploads) in both the FSL and MS snapshots. In general, uploading the FSL snapshot is faster than uploading the MS snapshot in individual steps. Our investigation reveals that the FSL snapshot has a larger chunk size on average and hence fewer chunks to process per 1 MB of data. Overall, as in Experiment B.1, the key generation of TED remains efficient and does not slow down the overall upload operation.

Experiment B.5 (Upload/download speeds). We conduct trace-driven evaluation on the upload and download performance of TEDStore. We pick ten snapshots from each of the FSL and MS datasets, such that the aggregate pre-

Steps	FSL	MS
Fingerprinting	2.7ms	2.7ms
Hashing	0.4ms	0.4ms
Key seeding	1.3ms	2.0ms
Key derivation	0.07ms	0.1ms
Encryption	5.4ms	5.6ms
Write	5.4ms	5.8ms

Table 2. Experiment B.4: Breakdown of computational time per 1 MB of uploads in real-world workloads.

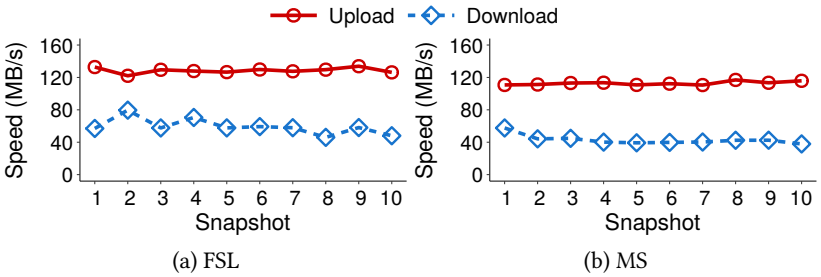


Figure 9. Experiment B.5: Upload/download speeds in real-world workloads. The x-axis represents the snapshots based on their upload/download orders.

deduplicated sizes of the FSL and MS snapshots are 2.0 TB and 1.1 TB, respectively. We upload the selected snapshots of each dataset in the order of their creation times, followed by downloading them.

Figure 9 shows the upload and download speeds for each snapshot. The upload speed remains stable, at 122.0-133.8 MB/s in FSL and 110.6-116.9 MB/s in MS. Note that the upload speed is lower than that in our evaluation on synthetic workloads (Experiment B.3), mainly due to the fingerprint index access overhead. Specifically, we now implement the fingerprint index based on LevelDB [2], which incurs high I/O compaction overhead when the number of entries increases [50]. For example, the number of fingerprint index entries in the MS dataset is about 1.78× that in the FSL dataset, so the upload speed of the MS dataset is 12.2% less than that of the FSL dataset.

The download speed generally decreases in both datasets, from 67.0 MB/s to 48.0 MB/s in FSL and from 57.6 MB/s to 37.9 MB/s in MS. In addition to the lookup overhead in the fingerprint index for identifying chunk locations, *chunk fragmentation* [46] also degrades the restore performance of the later added snapshots, whose chunks are scattered in storage after deduplication. Thus, more disk seeks are incurred to retrieve the later added snapshots.

To improve the upload/download performance, we can optimize the indexing techniques [69, 72], as well as leverage rewriting and caching to mitigate chunk fragmentation [46]. We pose these issues as future work.

6 Related Work

Encrypted deduplication. MLE [16] formalizes the theoretical framework of encrypted deduplication. Follow-up studies address different aspects of MLE from a theoretical perspective, including parameter dependency [4], data correlation [14], and updates [70]. Liu *et al.* [49] present a generalized security model for encrypted deduplication.

On the applied side, various encrypted deduplication systems (e.g., [5, 8, 24, 26, 37, 58, 64, 68]) realize the MLE construction via convergent encryption (CE) [26]. Some approaches augment CE with secret sharing [45] and transparent metadata management [62]. However, CE is vulnerable to offline brute-force attacks (§2.1).

DupLESS [15] implements server-aided MLE by performing key management in a dedicated key manager, so as to defend against offline brute-force attacks (§2.1). Several studies improve DupLESS in different aspects, such as quorum-based key management [27], efficient key generation via cross-user file-level deduplication [71], and decentralized key agreement among users without a dedicated key manager [48]. Some studies augment encrypted deduplication with new functionalities, such as periodic verification of storage space [10], dynamic access control [59], bandwidth-efficient uploads [25], or space-efficient metadata management [43].

However, all the above implementations of encrypted deduplication build on deterministic encryption and inevitably leak the frequency distribution of original data.

Attacks against encrypted deduplication. Some studies identify potential attacks against encrypted deduplication. Offline brute-force attacks [15] can infer the original plaintext chunk of a ciphertext chunk by testing all candidate plaintext chunks, or learn the remaining content of a file [67]. Side-channel attacks [11, 25, 31, 32, 57, 73] can infer the content of an already stored file by examining if a chunk can be deduplicated via client-side deduplication. Ritzdorf *et al.* [61] exploit chunk sizes to infer the existence of a file. TED performs server-aided MLE and provider-side deduplication to defend against offline brute-force attacks and side-channel attacks, respectively (§2.2). It can also be combined with the countermeasures against chunk size leakage [61].

Our work focuses on defending against frequency analysis in encrypted deduplication. Li *et al.* [44] show how to increase the inference rate of frequency analysis (from an adversarial perspective) by exploiting chunk locality [47, 72]. TED defends against frequency analysis by relaxing the deterministic nature of MLE via a tunable mechanism.

Defenses against frequency analysis. In §2.4, we have reviewed the limitations of existing defense approaches against frequency analysis in encrypted deduplication. Some studies propose frequency analysis defenses for encrypted databases. Kerschbaum [39] as well as Lewi and Wu [42] propose to hide attribute frequencies by randomizing ciphertexts. Frequency-

smoothing encryption [41] formalizes a cryptographic framework to prevent frequency analysis in databases. Such approaches, however, cannot be adapted to encrypted deduplication, since they either prohibit deduplication for generating random ciphertexts [39, 42], or incur high performance overhead by using computational expensive cryptographic primitives (e.g., homomorphic encoding) [41].

7 Conclusion

This paper addresses the dilemma of achieving both storage efficiency and data confidentiality in encrypted deduplication for outsourced storage. TED is a new cryptographic primitive that supports *tunable encrypted deduplication*, in which users can balance the trade-off between storage efficiency and data confidentiality through a configurable storage blowup factor, so as to relax the deterministic nature of the well-known MLE primitive and defend against frequency analysis. We realize TED in an encrypted deduplication storage prototype TEDStore, and demonstrate via extensive trace-driven evaluation that TED enables a tunable storage-confidentiality trade-off and incurs low performance overhead.

Acknowledgement

We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their valuable comments. We thank Haibin Zhang for his early suggestions. This work was supported in part by grants by National Key R&D Program of China (Grant number 2017YFB0802300), National Natural Science Foundation of China (Grant numbers 61972073 and 61602092), Open Research Project of the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences (Grant number 2019-MS-05), and the Research Grants Council of Hong Kong (CRF C7036-15G).

References

[1] FSL traces and snapshots public archive. http://tracer.filesystems.org/.
[2] LevelDB. https://github.com/google/leveldb.
[3] OpenSSL: Cryptography and SSL/TLS toolkit. https://www.openssl.org/.
[4] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In *Proc. of CRYPTO*, 2013.
[5] 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.
[6] I. A. Al-Kadit. Origins of Cryptology: The Arab Contributions. *Cryptologia*, 16(2):97–126, 1992.
[7] G. Amvrosiadis and M. Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proc. of USENIX ATC*, 2015.
[8] P. Anderson and L. Zhang. Fast and secure laptop backups with encrypted de-duplication. In *Proc. of USENIX LISA*, 2010.
[9] A. Appleby. SMHasher. https://github.com/aappleby/smhasher.
[10] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proc. of ACM CCS*, 2015.
[11] F. Armknecht, C. Boyd, G. T. Davies, K. Gjøsteen, and M. Toorani. Side channels in deduplication: Trade-offs between leakage and efficiency. In *Proc. of ACM ASIACCS*, 2017.
[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] T. Baignères, P. Junod, and S. Vaudenay. How far can we go beyond linear cryptanalysis? In *Proc. of ASIACRYPT*, 2004.
[14] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Proc. of PKC*, 2015.
[15] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.
[16] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EUROCRYPT*, 2013.
[17] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. of IEEE MASCOTS*, 2009.
[18] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. In *Proc. of VLDB*, 2018.
[19] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX ATC*, 2006.
[20] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
[21] A. Z. Broder. On the resemblance and containment of documents. In *Proc. of SEQUENCES*, pages 21–29, 1997.
[22] Z. Cao, H. Wen, F. Wu, and D. H. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proc. of USENIX FAST*, 2018.
[23] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[24] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of USENIX OSDI*, 2002.
[25] H. Cui, C. Wang, Y. Hua, Y. Du, and X. Yuan. A bandwidth-efficient middleware for encrypted deduplication. In *Proc. of IEEE DSC*, 2018.
[26] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.
[27] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *Proc. of ACM CCSW*, 2014.
[28] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. Technical Report HPL-2005-30(R.1), Hewlett-Packard Laboratories, 2005.
[29] 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.
[30] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proc. of IEEE S & P*, 2017.
[31] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proc. of ACM CCS*, 2011.
[32] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.
[33] HIPAA Journal. Hard drive theft sees data of 1 million individuals exposed. https://www.hipaajournal.com/hard-drive-theft-sees-data-1-million-individuals-exposed-8859/, 2017.
[34] IDC. Data age 2025. https://www.seagate.com/our-story/data-age-2025/.
[35] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. of ACM SYSTOR*, 2009.
[36] A. Juels and B. S. Kaliski, Jr. Pors: Proofs of retrievability for large files. In *Proc. of ACM CCS*, 2007.
[37] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus:

Scalable secure file sharing on untrusted storage. In *Proc. of USENIX FAST*, 2003.

[38] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014.

[39] F. Kerschbaum. Frequency-hiding order-preserving encryption. In *Proc. of ACM CCS*, 2015.

[40] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[41] M.-S. Lacharité and K. G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Trans. on Symmetric Cryptology*, pages 277–313, 2018.

[42] K. Lewi and D. J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proc. of ACM CCS*, 2016.

[43] J. Li, P. P. C. Lee, Y. Ren, and X. Zhang. Metadedup: Deduplicating metadata in encrypted deduplication via indirection. In *Proc. of IEEE MSST*, 2019.

[44] J. Li, C. Qin, P. P. C. Lee, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis. In *Proc. of IEEE/IFIP DSN*, 2017.

[45] 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.

[46] 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.

[47] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proc. of USENIX FAST*, 2009.

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

[49] J. Liu, L. Duan, Y. Li, and N. Asokan. Secure deduplication of encrypted data: Refined model and new constructions. In *Proc. of CT-RSA*, 2018.

[50] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proc. of USENIX FAST*, 2016.

[51] D. Meister, A. Brinkmann, and T. Süß. File recipe compression in data deduplication systems. In *Proc. of USENIX FAST*, 2013.

[52] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of USENIX FAST*, 2011.

[53] 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. of USENIX Security*, 2011.

[54] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, 2004.

[55] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. of ACM CCS*, 2015.

[56] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981.

[57] Z. Pooranian, K.-C. Chen, C.-M. Yu, and M. Conti. RARE: Defeating side channels based on data-deduplication in cloud storage. In *Proc. of IEEE INFOCOM*, 2018.

[58] P. Puzio, R. Molva, M. Önen, and S. Loureiro. ClouDedup: Secure deduplication with encrypted data for cloud storage. In *Proc. of IEEE CloudCom*, 2013.

[59] C. Qin, J. Li, and P. P. C. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Trans. on Storage*, 13(1):9, 2017.

[60] M. C. Rabin. Fingerprint by random polynomials, 1981.

[61] H. Ritzdorf, G. O. Karame, C. Soriente, and S. Čapkun. On information leakage in deduplicated storage systems. In *Proc. of ACM CCSW*, 2016.

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

[63] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In *Proc. of FC*, 2014.

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

[65] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok. A long-term user-centric analysis of deduplication patterns. In *Proc. of IEEE MSST*, 2016.

[66] 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.

[67] Z. Wilcox-O'Hearn. Drew perttula and attacks on convergent encryption. https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html.

[68] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proc. of ACM StorageSS*, 2008.

[69] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proc. of USENIX ATC*, 2011.

[70] Y. Zhao and S. S. Chow. Updatable block-level message-locked encryption. In *Proc. of ACM ASIACCS*, 2017.

[71] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, and C. Li. Secdep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In *Proc. of IEEE MSST*, 2015.

[72] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.

[73] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun. Mitigating traffic-based side channel attacks in bandwidth-efficient cloud storage. In *Proc. of IPDPS*, 2018.