Tunable Encrypted Deduplication with Attack-Resilient Key Management *

Zuoru Yang[†], Jingwei Li[‡], Yanjing Ren[‡], Patrick P. C. Lee[†]
[†]The Chinese University of Hong Kong
[‡]University of Electronic Science and Technology of China

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. In addition, we extend TED with a distributed key management architecture, and propose two attack-resilient key generation schemes that trade between performance and fault tolerance. 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.

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 [36]. 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

*An earlier version of this article appeared in [51]. In this extended version, we propose two attack-resilient key generation schemes (§4) and study their effectiveness in our experiments (§6.3). We enhance our prototype and revise our experiments based on the latest prototype. We also include new experiments to analyze the impact of the skewness of the chunk frequency distribution and the prototype performance in a geo-distributed environment. Furthermore, we present the detailed algorithm and correctness proof for automated parameter configuration (Appendix).

This work was supported in part by grants by the National Natural Science Foundation of China (61972073), the Key Research Funds of Sichuan Province (2021YFG0167), the Sichuan Science and Technology Program (2020JDTD0007), the Fundamental Research Funds for Chinese Central Universities (ZYGX2020ZB027, ZYGX2021J018), and CUHK Direct Grant 2020/21 (4055148). Corresponding author: Jingwei Li.

prohibiting deduplication on the ciphertext chunks. Bellare *et al.* [13] 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 [25] and server-aided MLE [12] (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 [48] 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 on 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) [78]. 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 the following key design techniques:

- *Sketch-based frequency counting*, which leverages a compact data summary structure called *sketch* [22] 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) [44] 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.

- *Attack-resilient key management*, which extends TED with distributed key management to defend against the compromise attacks against key management. We propose two attack-resilient key generation schemes that trade between performance and fault tolerance; both schemes preserve the configurability nature of TED.

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 [28, 59]. 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. Furthermore, our attack-resilient key generation schemes achieve at least a 14× key generation speedup over existing approaches [7, 12], while incurring limited performance overhead (e.g., up to 19.8%) in TEDStore. 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**.

The rest of the paper proceeds as follows. In §2, we formulate and motivate the problem of tunable encrypted deduplication, and present our threat model. In §3, we present the design of TED. In §4, we present the attack-resistant key generation schemes for TED. In §5, we present the implementation details of TEDStore based on TED. In §6, we present our evaluation results of both TED and TEDStore. In §7, we review related work. In §8, we conclude the paper. In Appendix, we present the detailed algorithm and correctness proof for automated parameter configuration (§3.5).

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 KiB each) [27] 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 [18]). 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 [54, 78, 88], virtual machine images [38], and file system snapshots [59, 77], and is adopted by commercial cloud services (e.g., Dropbox and Memopal) [34].

Encrypted deduplication. As described in §1, encrypted deduplication preserves the deduplication effectiveness on ciphertext 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)* [13], 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)* [25], 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., [2, 5, 23, 25, 40, 67, 75, 80]). However, CE is vulnerable to *offline brute-force attacks* [12], 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 assumes that the plaintext chunks are derived from an *unpredictable* message space, so that offline brute-force attacks become infeasible [12].

To defend against offline brute-force attacks for the plaintext chunks derived from a *predictable* message space, DupLESS [12] 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 key of a chunk, it submits a "blinded" fingerprint via the *oblivious pseudo-random function (OPRF)* [61] 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 [13] and server-aided MLE [12] 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* [62]; for example, the auxiliary dataset can refer to the plaintext chunks of some prior versions of backups [48]. Previous studies show that the auxiliary dataset can be obtained via public data releases [32,62], security breaches [17], or storage device theft [35]. The adversary also accesses a set of ciphertext chunks as the attack object (e.g., the latest version of backups [48]). 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.

Frequency analysis is a historically well-known cryptanalysis attack [3]. It is recently shown to cause substantial information leakage in encrypted databases [17, 32, 62] as well as encrypted deduplication [48]. Our goal is to achieve data confidentiality against frequency analysis.

2.2 Architecture

Our work builds on the server-aided MLE [12] 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) [34, 60], we perform deduplication in the provider, so that malicious clients cannot infer the deduplication patterns via client-side deduplication [34]. While the architecture in Figure 1 only shows a single key manager, we will extend our design to support multiple key managers to protect against the single-point-of-failure of a single key manager (§4).

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.

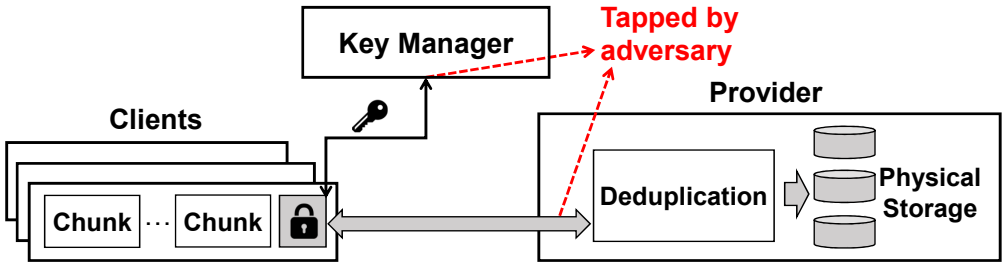


Figure 1: An encrypted deduplication architecture. An adversary may have access to the key manager and the provider to monitor the behaviors of their operations (§2.3).

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 (§2.1) 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 on 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 on 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 [12] (§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 [9, 39] for data integrity, as well as deduplication-aware secret sharing [52] 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* [1]: 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* [11]: 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* [74]: 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* [48]: 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 [16], so the keys (i.e., the minimum fingerprints) for similar segments are likely the same (due to Broder's theorem [20]). 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* [41] 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 [48]. 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 [16] 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 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., [42, 45, 46]) protect encrypted databases against frequency analysis. However, they are not applicable to encrypted deduplication (§7).

3 TED Design

We now present the design of TED. Our discussion in this section focuses on a single key manager, and we extend TED with multiple key managers in §4.

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 [4, 78]. 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 [12], 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)* [22] 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 ($1 \leq i \leq r$). For each input chunk, we increment each of its hashed counters by one. To query 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 [22]. For example, our trade-off analysis (§6.2) by default sets $r = 4$ with $w = 2^{25}$ 4-byte counters each, so the memory usage is 512 MiB. 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 [22].

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 [12] (§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.

Remarks. The security of sketch-based frequency counting builds on the assumption that a short hash function maps multiple chunks to the same counter index, such that an adversary cannot correctly deduce the input chunk from the counter index. While auxiliary knowledge (§2.3) may provide hints for an adversary to infer plaintext chunks from the short hashes, we can configure a small CM-Sketch (e.g., with small r and w) to deliberately increase the likelihood of hash collisions, so that the adversary cannot readily correlate the auxiliary knowledge with its observed frequency distribution of short hashes. The trade-off is that the error bound of sketch-based frequency counting increases, thereby affecting the frequency distribution of resulting ciphertext chunks (§3.4) and the parameter configuration for confining storage blowup (§3.5). In Experiment A.2 (§6.2), we study the impact of a relatively small w. We pose the security analysis of sketch-based frequency counting based on short hashes as future work.

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 short 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 k 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 on 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)* [44] 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 [45]. 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 a *mixed integer non-linear optimization* problem, which is known to be NP-hard [65]. 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 show that the optimal solution can be found based on the simplex algorithm [19] (see Appendix for details) and is given by:

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

where m is the maximum integer such that $f_m \leq \frac{\sum_{j=m+1}^{n} f_j}{n^* - m}$. Since $f_1 \leq \ldots \leq f_n$, we also ensure that $f_1^* \leq f_2^* \leq \ldots f_{n^*}^*$. 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):

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

Example. We show via a toy example how t is configured. Suppose that the chunk space is $\{M_i\}_{i=1}^{6}$ and the corresponding frequency distribution is $\{f_i\}_{i=1}^{6} = \{1, 1, 1, 2, 4, 6\}$ (i.e., M_i appears f_i times). Suppose that the storage blowup factor b is configured at $b = 1.5$. That is, we have the number of resulting unique ciphertext chunks $n^* = 6 \times 1.5 = 9$. We formulate the optimization problem according to Equation (6) as follows:

$$\begin{aligned} &\text{minimize } KLD \\ &\text{subject to } \sum_{i=1}^{n^*} f_i^* = 1 + 1 + 1 + 2 + 4 + 6 = 15, \\ &\qquad 0 \leq f_1^* \leq 1, 0 \leq f_2^* \leq 1, \\ &\qquad 0 \leq f_3^* \leq 1, 0 \leq f_4^* \leq 2, \\ &\qquad 0 \leq f_5^* \leq 4, 0 \leq f_6^* \leq 6, \\ &\qquad f_i^* \text{ is an integer } \forall i \in [1, n^*], \end{aligned} \tag{9}$$

where f_i^* is the frequency of each unique ciphertext chunk.

We solve the above optimization problem using Algorithm 1 (Appendix) and obtain the solution $\{f_i^*\}_{i=1}^{9} = \{1, 1, 1, 2, 2, 2, 2, 2, 2\}$ (Equation (7)). We configure the balance parameter $t = 2$ as the maximum value in $\{f_i^*\}_{i=1}^{9}$. This implies that the unique plaintext chunk (e.g., M_1, M_2, M_3, and M_4) whose frequency is less than or equal to two only forms a single unique ciphertext chunk, while the unique plaintext chunk M_5 (resp. M_6) whose frequency is greater than two is encrypted into two (resp. three) unique ciphertext chunks, since $\lfloor f/t \rfloor$ changes with the accumulation of the corresponding frequency f (Equation (1)). Note that we have at most $n^* = 9$ unique ciphertext chunks, and we bound the actual storage blowup by up to 1.5.

In addition, we show that TED reduces the KLD of MLE. MLE keeps the same frequency distribution (i.e., $\{1, 1, 1, 2, 4, 6\}$) of the plaintext chunks after encryption due to its deterministic nature. The KLD of the ciphertext chunks is 0.38. On the other hand, TED encrypts some plaintext chunks into multiple distinct ciphertext chunks, and reduces the KLD of the resulting ciphertext chunks to 0.06 (i.e., 84.2% less than that of MLE). This shows that TED is more secure against frequency leakage than MLE (see §3.6 for detailed analysis).

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 propose to solve the optimization problem repeatedly as the upload operation proceeds and 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 GiB file (assuming that the average chunk size is 8 KiB).

3.6 Security Discussion

Finally, we discuss the security implications of TED via 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 [48], 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 success probability of the guess can be approximated as [10]:

$$\text{Success Probability} \approx 1 - \Phi(-\frac{\sqrt{2S \times KLD}}{2}), \tag{10}$$

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 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 §6.2, MLE and MinHash encryption have a KLD of 1.72 and 1.35, respectively. If we set the storage blowup factor $b = 1.2$, TED achieves a KLD of 0.26. This implies that the adversary against TED needs to access $1.72/0.26 \approx 6.6\times$ sampled ciphertext chunks to achieve the same success probability of the guess when compared to that against MLE. Also, it needs to access $1.35/0.26 \approx 5.2\times$ sampled ciphertext chunks to achieve the same success probability of the guess when compared to that against MinHash encryption.

SKE does not support deduplication, since it uses random keys for encryption/decryption. Thus, the frequency distribution of ciphertext chunks is uniform (i.e., KLD is zero), and robust against information leakage. TED can match the same security level as SKE by configuring the balance parameter $t = 1$ (or equivalently the storage blowup factor $b \to \infty$), in which each chunk is encrypted with a distinct key.

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 Attack-resilient Key Management

One limitation of TED is that the single key manager suffers from both security and availability weaknesses. From the security perspective, if the key manager is compromised, an adversary can obtain the global secret and launch offline brute-force attacks (§2.1). From the availability perspective, the key manager is the

single-point-of-failure, in that if the key manager fails, the key management functionality and hence the whole TED system will become unavailable.

To address the security and availability weaknesses of TED, we extend TED with *multiple* key managers. We propose two attack-resistant key generation schemes, namely the *unanimity-based scheme* and the *quorum-based scheme*. Both schemes preserve the configurability nature of TED and aim to be secure against the compromise of a key manager. The major difference is that the unanimity-based scheme incurs limited performance overhead compared to the single-key-manager baseline, while the quorum-based scheme further provides fault tolerance with additional performance overhead.

4.1 Unanimity-based Scheme

Main idea. The idea of the unanimity-based scheme is that each key manager maintains an individual secret (rather than a global secret as in the single-key-manager baseline) and a CM-Sketch (§3.3) for key generation, such that the key of a chunk is derived based on the secrets of all key managers. In this case, the adversary cannot learn the key through offline brute-force attacks even if it compromises some (but not all) of the key managers. Specifically, to generate the key of a chunk, a client first sends the short hashes of the chunk to all key managers. Each key manager counts the frequency (§3.3) and returns the key seed based on the short hashes and its secret (§3.4). Note that the key seeds are different across the key managers. Finally, the client combines all the key seeds to form the key.

However, it is non-trivial to extend the above scheme with the probabilistic key generation approach (§3.4). In the single-key-manager baseline, the key manager randomly picks a key seed from a candidate set. When multiple key managers are used, since each key manager performs key generation independently, the probabilistic key generation approach significantly amplifies the key space of each chunk and hurts the deduplication effectiveness. For example, suppose that we deploy u key managers (e.g., $u = 4$ by default in our evaluation) and each key manager creates a candidate set of key seeds $\{k_0, k_1, \ldots, k_x\}$ for a chunk M, where $x = \lfloor f/t \rfloor$, f is the current frequency of M and t is the balance parameter. As each key manager independently picks a random key seed from the candidate set, the size of the key space of M becomes as large as $(x+1)^u$ (as opposed to $x+1$ for probabilistic key generation in the single-key-manager baseline). This implies that the current copy of M only has a probability of at most $f/(x+1)^u < f/x^u \approx \frac{t^u}{f^{u-1}}$ to be deduplicated with some of the f already stored copies. As f increases, the probability becomes practically negligible.

To enable probabilistic key generation with multiple key managers while preserving the deduplication effectiveness, we propose to let the client synchronize a common random factor θ across all key managers, so that they pick their key seeds derived from the same θ, so as to reduce the key space of each chunk for deduplication (see details below). Our current design allows the client to directly specify and send θ to each key manager, so as to simplify the coordination among key managers. Note that this design does not introduce additional security risks, since the key manager cannot infer any original information from the random factor θ.

Detailed procedure. We elaborate the unanimity-based key generation scheme as follows. To generate the key of each plaintext chunk M, a client picks a freshly new random factor θ (note that θ does not need to be identical for the same M), and sends the short hashes $\{h_i(M)\}_{i=1}^{r}$ of M, as well as θ, to all u key managers. Each key manager computes the key seed candidate k_x based on its own secret via Equation (2). It then picks the key seed k_τ from the candidate set $\{k_0, k_1, \ldots, k_x\}$, where $\tau = \theta \mod (x+1)$ (i.e., mapping the random factor θ generated by the client into τ, which lies within the range $[0, x]$). Note that while each key manager chooses the same τ, the returned k_τ is distinct for different key managers, since each key manager independently maintains its own (distinct) secret. Finally, the client computes k by performing a bitwise-XOR operation on the key seeds $\{k_\tau\}$ received from all key managers, and derives the key of M based on Equation (4).

The unanimity-based scheme keeps the size of the key space of M as $x+1$ (i.e., independent of the

number of key managers) as in the single-key-manager baseline, and hence maintains the deduplication effectiveness. It also incurs limited performance overhead over the single-key-manager baseline, since it only adds the lightweight bitwise-XOR operation to combine all key seeds together. Furthermore, it preserves the security against the compromise of up to $u-1$ key managers, since the key is generated based on the secrets of all u key managers.

4.2 Quorum-based Scheme

One limitation of the unanimity-based scheme is that it lacks the fault tolerance capability. Since the key of a chunk is derived from the key seeds of all key managers, the key generation will fail if any one of the key managers is unavailable. To this end, we propose a quorum-based scheme, which extends the unanimity-based scheme with fault tolerance, with a trade-off of incurring additional performance overhead. Our quorum-based scheme builds on two existing techniques: *Shamir's secret sharing* [73] and *homomorphic hashing* [43].

Background. Before we introduce the quorum-based scheme, we first provide the background details for Shamir's secret sharing [73] and homomorphic hashing [43].

Shamir's secret sharing can be constructed with two configurable parameters u and v, where $v < u$. A (u, v)-Shamir's secret sharing scheme encodes a data unit (called a *secret*) into u pieces (called *shares*), such that the secret can be reconstructed from any v out of the u shares via the linear combination with the corresponding reconstruction parameters, and the secret cannot be inferred from any $v-1$ shares.

Homomorphic hashing allows the hash of the sum of two data blocks to be computed from the hashes of individual data blocks. We focus on the *discrete-log-based homomorphic hash function* [43], which can be defined by two big prime numbers $\mathcal{P}$ and $\mathcal{Q}$ (e.g., of lengths 1024 bits and 33 bits, respectively), where $\mathcal{Q}$ is a factor of $\mathcal{P}-1$. It initializes a set of r bases $\{g_i\}_{i=1}^{r}$, where $g_i = s^{(\mathcal{P}-1)/\mathcal{Q}} \mod \mathcal{P}$, and s is a fresh random value selected from $[0, \mathcal{P}]$ for each g_i. All arithmetic is performed in the finite field $GF(\mathcal{P})$. Let A and B be any two data blocks of the same size. We divide A and B into r equal-size sub-blocks $\{A_i\}_{i=1}^{r}$ and $\{B_i\}_{i=1}^{r}$, respectively. Then the addition of two blocks A and B is defined as:

$$A + B = \{A_i\}_{i=1}^{r} + \{B_i\}_{i=1}^{r} = \{A_i + B_i\}_{i=1}^{r}, \tag{11}$$

and the scalar multiplication of some constant c is defined as:

$$c \cdot A = c \cdot \{A_i\}_{i=1}^{r} = \{A_i \cdot c\}_{i=1}^{r}. \tag{12}$$

The homomorphic hashes $\mathbf{H}'(A)$ and $\mathbf{H}'(B)$ of A and B are respectively defined as:

$$\mathbf{H}'(A) = \mathbf{H}'(\{A_i\}_{i=1}^{r}) = \prod_{i=1}^{r} g_i^{A_i} \quad \text{and} \quad \mathbf{H}'(B) = \mathbf{H}'(\{B_i\}_{i=1}^{r}) = \prod_{i=1}^{r} g_i^{B_i}. \tag{13}$$

Under homomorphic hashing, the hash of the sum of two data blocks (i.e., $\mathbf{H}'(A+B)$) can be computed from the hashes of individual data blocks (i.e., $\mathbf{H}'(A)$ and $\mathbf{H}'(B)$), i.e.,

$$\mathbf{H}'(A + B) = \mathbf{H}'(\{A_i + B_i\}_{i=1}^{r}) = \prod_{i=1}^{r} g_i^{A_i+B_i} = \prod_{i=1}^{r} g_i^{A_i} \cdot \prod_{i=1}^{r} g_i^{B_i} = \mathbf{H}'(A) \cdot \mathbf{H}'(B). \tag{14}$$

This property can be extended to scalar multiplication, i.e.,

$$\mathbf{H}'(c \cdot A) = \mathbf{H}'(\{c \cdot A_i\}_{i=1}^{r}) = \mathbf{H}'(A)^c. \tag{15}$$

Main idea. In the quorum-based scheme, we split a global secret into u shares via (u, v)-Shamir's secret sharing and store the shares in u different key managers, such that the global secret can be reconstructed if

any v out of u key managers are available for key generation, while no partial knowledge of the global secret is revealed if no more than $v-1$ key managers are compromised. To augment the key generation process with secret sharing, we let each key manager generate the key seeds based on homomorphic hashing, such that each key seed can still be derived from the short hashes and the share of the corresponding key manager (i.e., the share is treated as the secret of the key manager) (§3.4), while a client can reconstruct the key of a chunk upon receiving a sufficient number of key seeds from any v key managers.

Detailed procedure. We elaborate the quorum-based key generation scheme as follows. We deploy u key managers and allow the key of a chunk to be reconstructed from any v out of u key managers (where $v < u$) via secret sharing. Suppose that the j-th key manager, where $1 \leq j \leq u$, initially generates its secret κ_j based on (u, v)-Shamir's secret sharing on the global secret κ. Then the global secret κ can be reconstructed from the secrets of any v key managers via some linear combination. Without loss of generality, consider the first v key managers, and we have $\kappa = \sum_{j=1}^{v} \lambda_j \cdot \kappa_j$, where λ_j is some reconstruction parameter for the j-th key manager, where $1 \leq j \leq v$.

To generate the key of a plaintext chunk M, a client sends the short hashes $\{h_i(M)\}_{i=1}^{r}$ of M and the freshly new random factor θ to all u key managers as in the unanimity-based scheme (§4.1). Each key manager treats $\{h_i(M)\}_{i=1}^{r}$ as a data block, in which each short hash is a sub-block. It computes the key seed candidate k_x via the discrete-log-based homomorphic hash function as:

$$k_x = \mathbf{H}'(\kappa_j \cdot x \cdot \{h_i(M)\}_{i=1}^{r}), \tag{16}$$

where κ_j is the secret maintained by the j-th key manager ($1 \leq j \leq u$), $x = \lfloor f/t \rfloor$, f is the current frequency of M, and t is the balance parameter. It then follows probabilistic key generation (§4.1) to pick its key seed k_τ from the candidate set $\{k_0, k_1, \ldots, k_x\}$, where $\tau = \theta \mod (x+1)$. Suppose that the client receives v key seeds, say from the first v key managers. Let $k_{\tau,j}$ be the key seed returned by the j-th key manager for $1 \leq j \leq v$. The client then computes $k = \prod_{j=1}^{v} (k_{\tau,j})^{\lambda_j}$ and derives the key of M based on k (Equation (4)).

We point out that the quorum-based scheme ensures that k can be *deterministically* reconstructed based on any v out of u key seeds. Specifically, based on Equations (14-16), we can expand k as:

$$\begin{aligned} k &= \prod_{j=1}^{v} (k_{\tau,j})^{\lambda_j} = \prod_{j=1}^{v} \mathbf{H}'(\kappa_j \cdot \tau \cdot \{h_i(M)\}_{i=1}^{r})^{\lambda_j} = \prod_{j=1}^{v} \mathbf{H}'(\lambda_j \cdot \kappa_j \cdot \tau \cdot \{h_i(M)\}_{i=1}^{r}) \\ &= \mathbf{H}'(\sum_{j=1}^{v} \lambda_j \cdot \kappa_j \cdot \tau \cdot \{h_i(M)\}_{i=1}^{r}) = \mathbf{H}'(\kappa \cdot \tau \cdot \{(h_i(M)\}_{i=1}^{r}), \end{aligned} \tag{17}$$

in which the last term is always expressed as a function of the global secret κ.

Remarks on computational overhead. We now discuss the performance overhead of the major computational operations (that are performed in the finite field $GF(\mathcal{P})$) of the quorum-based scheme. For each plaintext chunk, to generate a key seed k_x, each key manager incurs $2 \times r$ multiplications for computing $\kappa_j \cdot x \cdot \{h_i(M)\}_{i=1}^{r}$ in Equation (16), as well as $r-1$ multiplications and r modular exponentiations for computing the homomorphic hash function $\mathbf{H}'(\cdot)$ (Equation (13)). Also, the client incurs $v-1$ multiplications and v modular exponentiations to combine all key seeds into k (Equation (17)).

Although the modular exponentiations are known to incur severe computational overhead when the base and exponent are large [64], we argue that the performance overhead is acceptable in our case. As opposed to the previous work [43] that uses the whole data block of 16 KiB as the exponent, each exponent in our quorum-based scheme is a short hash that has only 32 bits by default (§5). Thus, our quorum-based scheme has limited performance overhead due to homomorphic hashing. In Experiment B.2 (§6.3), we show that the quorum-based scheme incurs mild performance overhead in the key generation of TED.

5 Implementation

To show the applicability of TED, we built an encrypted deduplication prototype called TEDStore, which realizes TED for chunk encryption. TEDStore is written in C++ on Linux. It uses OpenSSL 1.1.1c [63] for cryptographic operations, MurmurHash3 [6] for the short hash operations (§3.3 and §3.4), and GNU MP 6.2.0 [31] for homomorphic hashing (§4.2). Our prototype now contains around 5.7K lines of code.

Deduplication. Each client now implements content-defined chunking based on FastCDC [82], which takes the minimum, average, and maximum chunk sizes as inputs (by default, we set them as 4 KiB, 8 KiB, and 16 KiB, 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 [29] 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 KiB each) in fixed-size *containers* (on the order of MiB each), such that the I/O operations are in units of containers [53]. We now fix the container size as 8 MiB.

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). Also, we choose SHA-256 for $\mathbf{H}(\cdot)$ in the key seed generation (Equation 2) and key derivation (Equation 4), as well as AES-256 for chunk encryption. Furthermore, for quorum-based key generation (§4.2), we follow the previous work [43] to configure the big prime number $\mathcal{P}$ in homomorphic hashing with 1,024 bits. We also configure $\mathcal{Q}$ with 33 bits to fit the size of each short hash (i.e., 32 bits).

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. To improve the download performance, the provider maintains an in-memory least-recently-used cache (1 GiB by default) to hold the most recently restored containers. For each download request, the provider first searches for the containers in the cache, and retrieves the containers from disk only if they are not in the cache. In addition, we deploy AES-256 in Galois/Counter Mode (GCM), and leverage the Intel AES New Instructions set to speed up encryption and decryption through hardware acceleration [37].

Prototype limitations. Our TEDStore prototype currently focuses on only the deduplication of data chunks, but not metadata (e.g., file recipes [57]). Also, it does not address the fault tolerance of the provider, yet we can implement a quorum-based design for storage [52]. Furthermore, it focuses on confidentiality, and does not support fine-grained access control (e.g., attribute-based encryption [30]).

Currently, TEDStore builds on server-aided key generation [12], which disables client-side key generation as in CE [25] (§2.1). While server-aided key generation protects against offline brute-force attacks, it also incurs performance overhead of interacting with the key manager, especially in geo-distributed environments. One possible performance optimization is to cache the most recently generated chunk-based keys on the client side, so that the client can reuse the cached keys for a large proportion of identical chunks from previous uploads [49]. However, the caching approach can complicate the tunable design of TEDStore, since it causes identical chunks to be directly encrypted with the same cached keys. How to design a proper caching approach is an open issue.

6 Evaluation

We conduct trace-driven evaluation to study the storage-confidentiality trade-off of TED (§6.2) and the performance of TEDStore in networked settings (§6.3). 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 accelerates key generation. The unanimity-based and quorum-based schemes achieve at least 31× and 14× speedups over existing approaches, respectively. When both schemes are deployed in TEDStore, they only incur at most 3.4% and 19.8% performance drops in uploads compared to the single-key-manager baseline, respectively.

6.1 Datasets

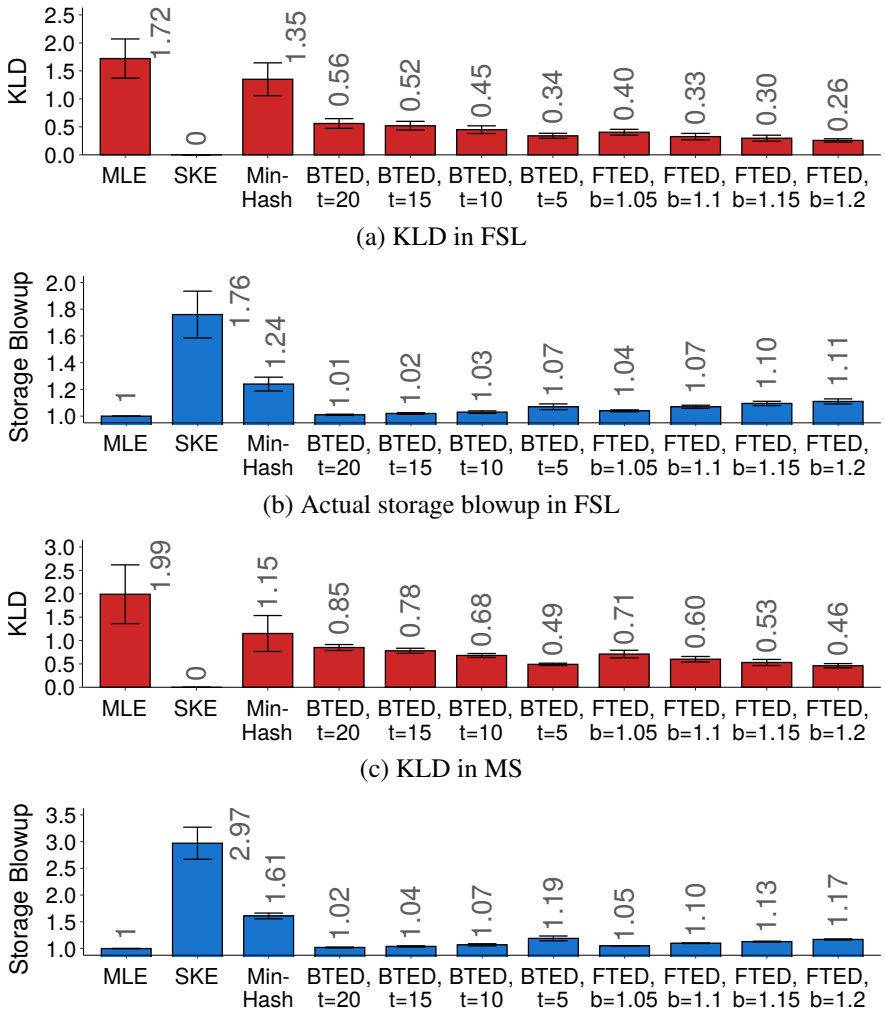
We consider the following datasets to derive our evaluation.

- *FSL.* This dataset is collected by the File systems and Storage Lab (FSL) at Stony Brook University [28]. 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 KiB. 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 GiB to 251.01 GiB. Our dataset covers a total of 3.08 TiB of pre-deduplicated data. The data size reduces to 1.54 TiB if we perform deduplication on each snapshot.
- *MS.* This dataset contains the Windows file system snapshots collected by Microsoft [59]. 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 KiB. We sample 30 snapshots, each of which is of size around 100 GiB. Our dataset covers a total of 3.91 TiB of pre-deduplicated data. The data size reduces to 1.34 TiB if we perform deduplication on each snapshot.
- *ZIPF.* This dataset contains a list of chunk fingerprints that are synthetically generated by ourselves based on a Zipf distribution, which is commonly found in backup workloads [84, 85]. We have developed a trace generator that takes the number of logical chunks (assuming that the chunk size is fixed), the deduplication ratio, and the Zipfian constant that defines a specific Zipf distribution as inputs. Suppose that each chunk can be identified by a fingerprint. We prepare a *candidate fingerprint set* of unique 48-bit fingerprints based on the expected number of unique chunks. For example, if the number of logical chunks is 6,553,600 (i.e., 50 GiB of pre-deduplicated data for a chunk size of 8 KiB) and the deduplication ratio is 10× [78], then the expected number of unique chunks is 655,360. For each logical chunk, we sample its fingerprint from the candidate fingerprint set based on the Zipf distribution defined by the input Zipfian constant, so that the frequencies of all unique chunks will also follow the target distribution. Finally, the trace generator outputs a list of chunk fingerprints for all logical chunks. We generate different traces for different Zipfian constants, and evaluate the impact of frequency skewness on the storage-confidentiality trade-off (Experiment A.6).

6.2 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).

By default, we apply deduplication to each snapshot (i.e., no deduplication across snapshots in a dataset),



(a) KLD in FSL

(b) Actual storage blowup in FSL

(c) KLD in MS

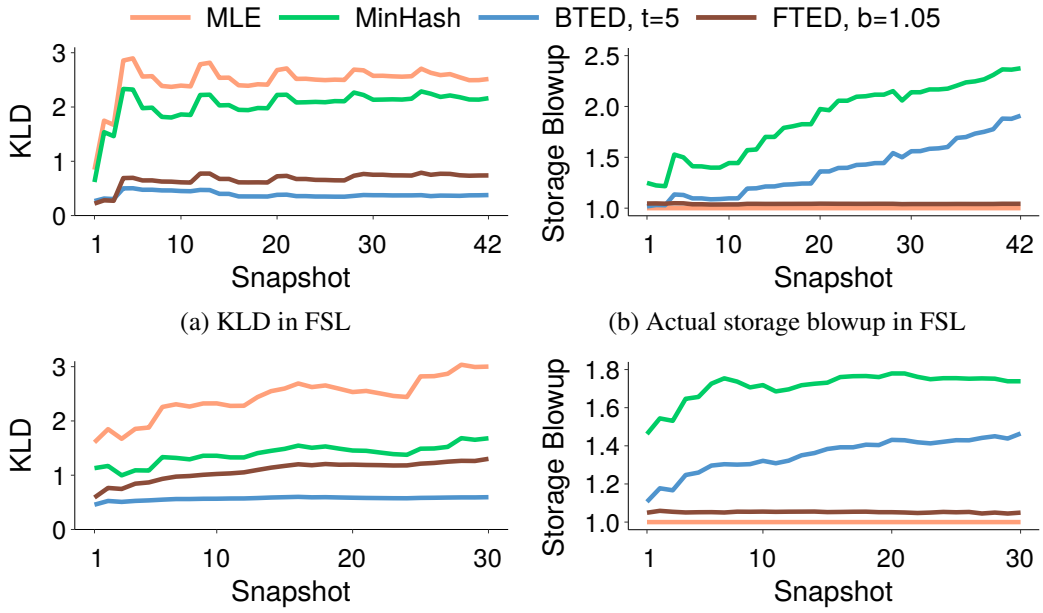
(d) Actual storage blowup in MS

Figure 2: Experiment A.1: The storage-confidentiality trade-offs of different encryption approaches for processing each snapshot in both FSL and MS datasets. Each error bar represents the 95% confidence interval across different snapshots in each encryption approach.

and present the average results of all snapshots in a dataset. In Experiment A.1, we also consider a long-term scenario, in which we apply deduplication to all snapshots that have already been stored.

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 deduplication for each snapshot in both FSL and MS datasets. 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 KiB, 1 MiB, and 2 MiB [50]; 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 (§6.1), 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.



(a) KLD in FSL

(b) Actual storage blowup in FSL

(c) KLD in MS

(d) Actual storage blowup in MS

Figure 3: Experiment A.1: The storage-confidentiality trade-offs of different encryption approaches after each snapshot is stored in both FSL and MS datasets. The x-axis represents the snapshots based on their upload orders.

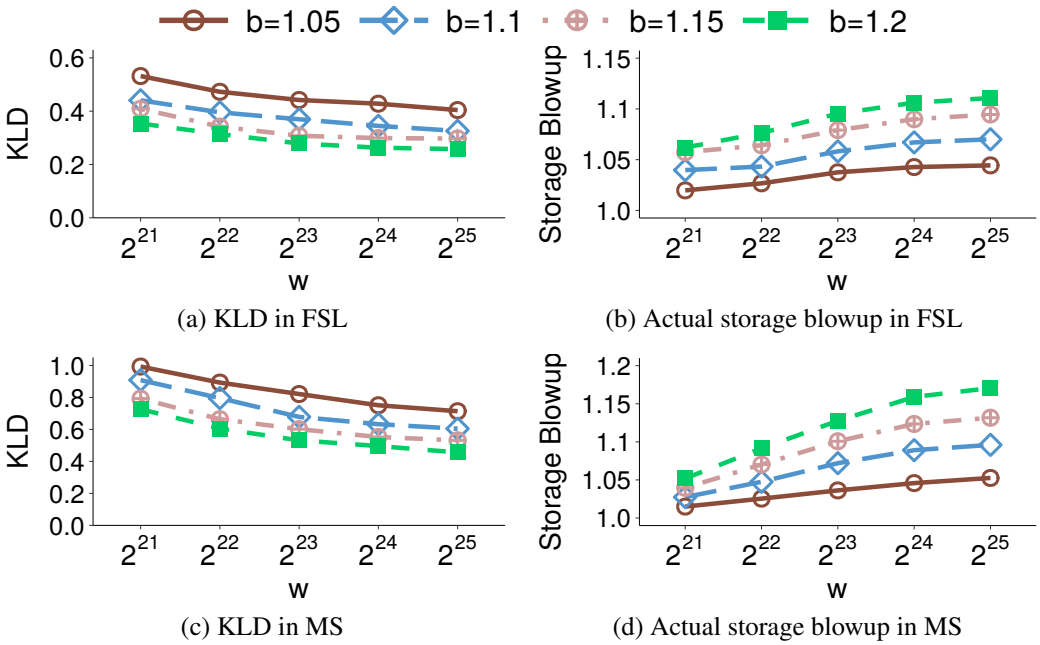
Figure 2 shows the average results over all snapshots in both FSL and MS datasets, with the 95% confidence intervals. MLE achieves exact deduplication (i.e., its actual storage blowup is always one), but incurs the highest KLD due to deterministic encryption. SKE has the minimum KLD (zero), but incurs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

In addition, we investigate the storage-confidentiality trade-off in a long-term storage scenario. We upload snapshots in the order of their creation times: for FSL, we order the snapshots by dates, and by user IDs if the snapshots belong to the same date; for MS, we order the snapshots by IDs. We measure the KLD and actual storage blowup after each snapshot is stored (i.e., deduplication is applied across the already stored snapshots). We consider MLE, MinHash encryption, BTED, and FTED for comparison. For BTED, we set $t = 5$. For FTED, we set $b = 1.05$.

Figure 3 shows the KLD and the actual storage blowup after each snapshot is stored, in both FSL and MS datasets. In FSL, the KLDs of all schemes show periodic fluctuations. The reason is that the snapshots



(a) KLD in FSL

(b) Actual storage blowup in FSL

(c) KLD in MS

(d) Actual storage blowup in MS

Figure 4: Experiment A.2: Analysis of sketch-based frequency counting. A larger w implies more accurate counting, at the expense of more memory usage.

from the same FSL users have similar chunk frequency distribution (and hence similar KLDs) and we upload the snapshots ordered by creation dates followed by user IDs. In MS, the KLDs of all schemes gradually increase, since the accumulated frequency distribution is more skewed as more snapshots are uploaded. Note that TED significantly reduces the KLDs in both FSL and MS datasets. For example, after storing the last snapshot, the KLD of MinHash encryption is 2.16 (1.68) in FSL (MS), while BTED and FTED reduce the KLD to 0.38 (0.60) and 0.74 (1.30), respectively.

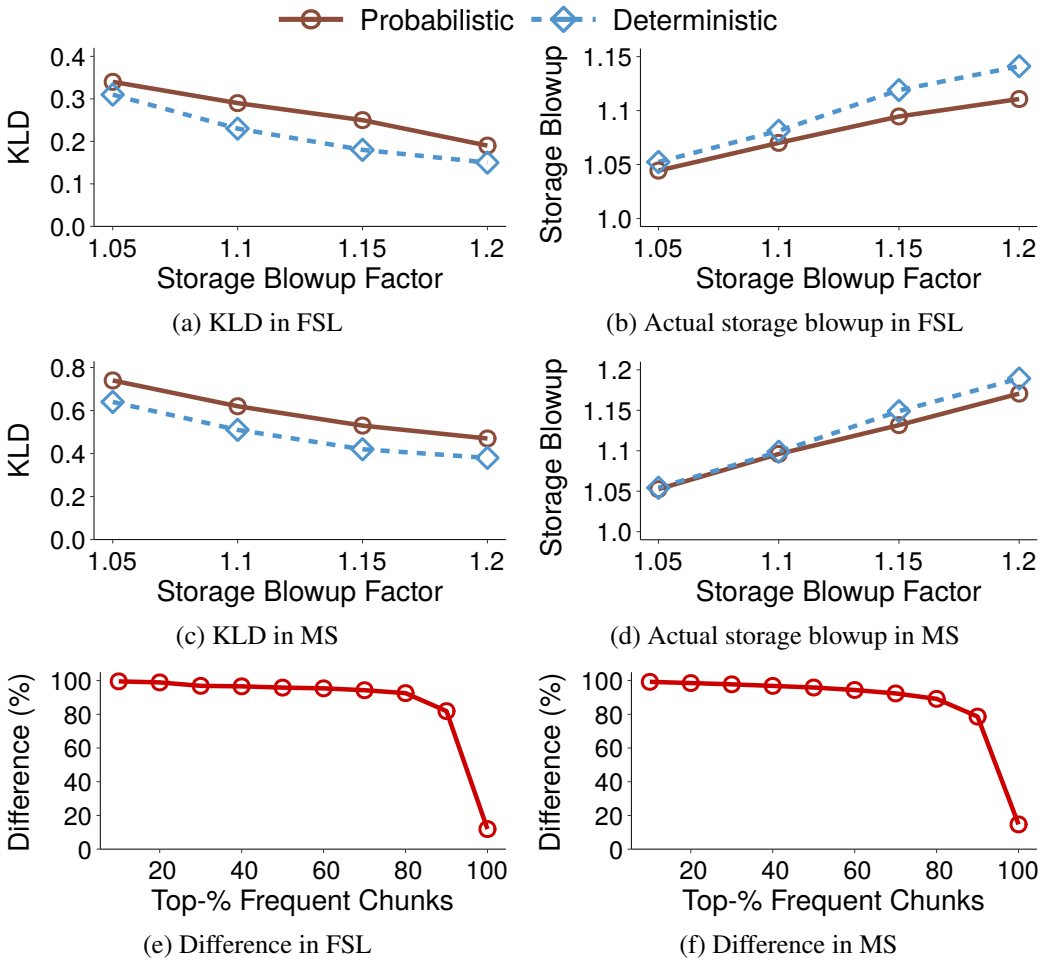
Also, the actual storage blowups of MinHash and BTED increase as more snapshots are stored. On the other hand, FTED effectively bounds the actual storage blowup with respect to b. For example, after the last FSL (MS) snapshot is stored, the actual storage blowup of FTED is 1.04 (1.05).

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 MiB to 512 MiB).

Figure 4 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 ciphertext chunks for deduplication. For example, in MS, 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 5(a)-5(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.



(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

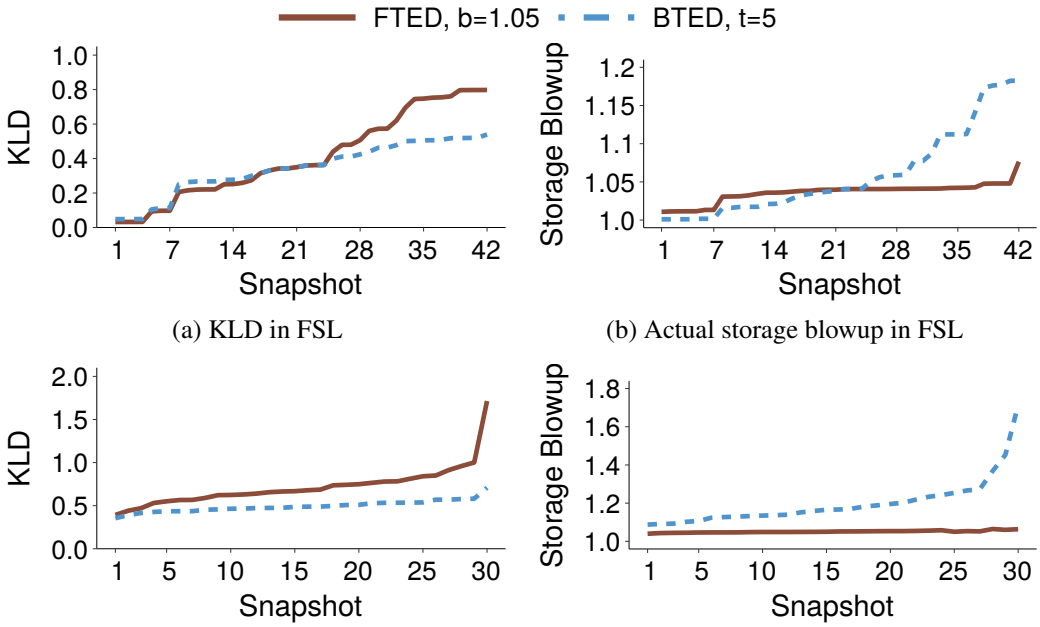
Figure 5: Experiment A.3: Analysis of probabilistic key generation, which we compare with deterministic key generation.

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 5(e) and 5(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



(a) KLD in FSL

(b) Actual storage blowup in FSL

(c) KLD in MS

(d) Actual storage blowup in MS

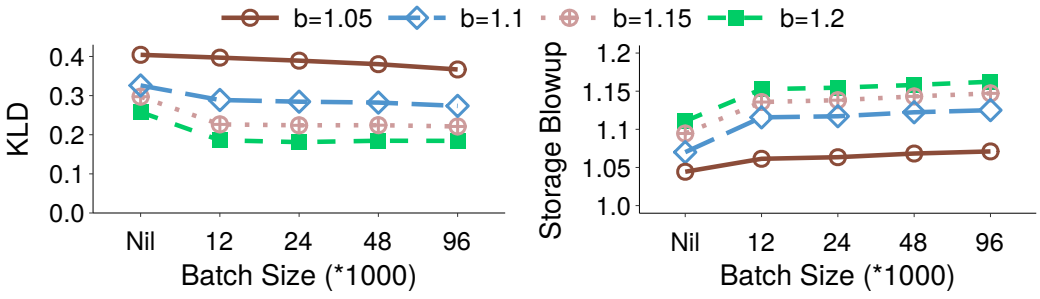
Figure 6: Experiment A.4: Comparison between BTED ($t = 5$) and FTED ($b = 1.05$) in the controllability of the actual storage blowup. Here, the x-axis refers to the snapshots sorted in the ascending order of their y-axis values.

BTED and $b = 1.05$ for FTED. The results are similar for other BTED and FTED variants. We apply BTED and FTED to each snapshot and present the results of all snapshots, sorted in ascending order of their y-axis values.

Figure 6 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.

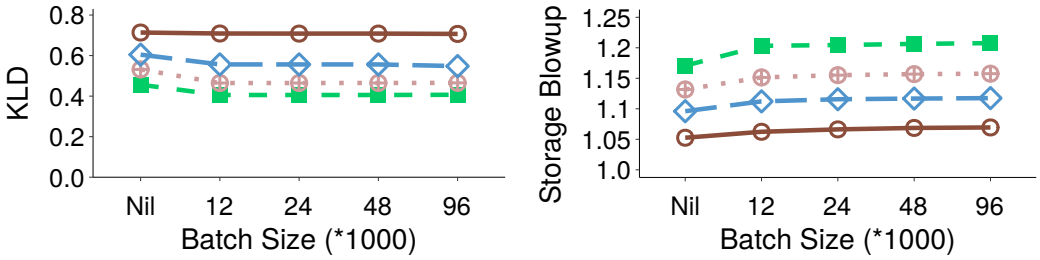
Experiment A.5 (Impact of batch size). To efficiently configure t in practice, a client issues key generation requests for a batch of plaintext chunks (§3.5). We study how batching affects the KLD and the actual storage blowup. Recall that with batching, TED initializes $t = 1$ and adjusts t on a per-batch basis. We focus on FTED with varying b.

Figure 7 shows the results versus the batch size (varied from 12,000 to 96,000) in both 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).



(a) KLD in FSL

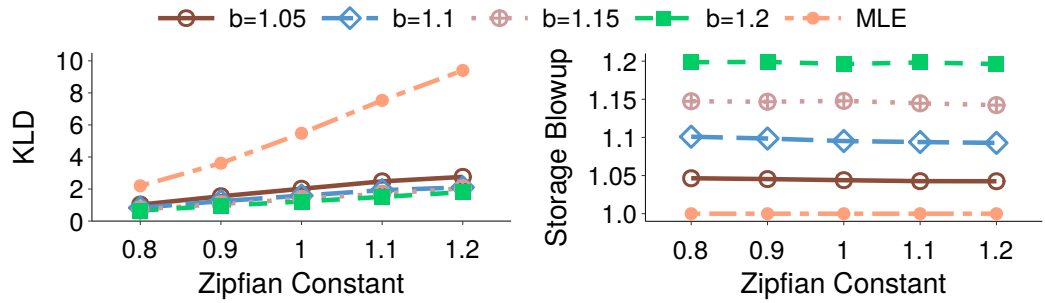
(b) Actual storage blowup in FSL



(c) KLD in MS

(d) Actual storage blowup in MS

Figure 7: 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.



(a) KLD

(b) Actual storage blowup

Figure 8: Experiment A.6: Impact of the distribution skewness on KLD and the actual storage blowup.

Experiment A.6 (Impact of distribution skewness). We study the impact of the frequency distribution skewness of plaintext chunks using the ZIPF dataset. We vary the Zipfian constant (§6.1) from 0.8 to 1.2 to simulate Zipf distributions with different skewness in real-world workloads [83]; a larger Zipfian constant implies a more skewed Zipf distribution. We fix the number of logical chunks as 6,553,600 and a deduplication ratio as 10×. We focus on FTED with varying b, and include the results of MLE as the baseline.

Figure 8 shows the results. When the Zipfian constant increases from 0.8 to 1.2, the KLD of MLE dramatically increases from 2.2 to 9.4, since a more skewed distribution leads to a higher KLD (i.e., more deviated from the uniform distribution). In contrast, the KLD of TED for different b increases at a much slower rate as the Zipfian constant increases; for example, when $b = 1.05$, the KLD of TED increases from 1.0 to 2.8 as the Zipfian constant increases from 0.8 to 1.2. Thus, TED remains effective in reducing the KLD for different skewness. Also, TED consistently keeps the actual storage blowup with respect to b for different Zipfian constants, since it adaptively configures t with respect to different workloads.

6.3 Performance Evaluation on TEDStore

We evaluate TEDStore in networked environments using both synthetic and real-world workloads. We consider three TED key generation schemes, including the single-key-manager scheme, the unanimity-based scheme, and the quorum-based scheme. By default, we choose the following parameters when realizing TED in TEDStore: $b = 1.05$, $r = 4$ and $w = 2^{21}$ (i.e., 32 MiB memory) for sketch-based frequency counting (§3.3), and a batch size of 48,000 chunks for key generation (§3.5). Based on the prior studies on evaluating quorum-based schemes [14, 15, 52, 58, 70, 76], we choose $u = 4$ key managers for both the unanimity-based and quorum-based schemes, while choosing $v = 3$ for the quorum-based scheme for tolerating the failures of any single key manager. We will evaluate different configurations of (u, v) in Experiment B.4 as well as the performance impact of geo-distributed key managers in Experiment B.5.

6.3.1 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, 32 GiB RAM and is installed with Linux Ubuntu 16.04. We run the clients, key managers, and the provider on distinct machines that are connected with 10 GbE. We also deploy TEDStore on Amazon EC2 for our evaluation in a geo-distributed environment (see Experiment B.5 for the detailed configuration).

We generate a synthetic dataset with a set of 2 GiB files, each of which comprises globally unique chunks (i.e., no duplicates). We let one or multiple clients issue a 2 GiB file of unique data to the provider simultaneously. 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 the microbenchmark evaluation by deploying a client, one or multiple key managers (depending on the key generation scheme being used), and a provider in distinct machines. We measure the computational time (excluding the communication time) of different steps when the client uploads a 2 GiB 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) *random factor generation*, in which the client picks a random factor; (v) *key seeding*, in which each of the key managers performs frequency counting, solves the optimization problem, and return a key seed for each chunk; (vi) *key derivation*, in which the client derives the key of each chunk; and (vii) *encryption*, in which the client encrypts each chunk.

Table 1 shows the breakdown of the computational time (per 1 MiB of uploads) for different key generation schemes. Fingerprinting and encryption are the most time-consuming steps, since they perform cryptographic operations on all file data. In contrast, the key generation of the single-key-manager scheme, including hashing, key seeding, and key derivation, takes only 5.2% of the overall computational time. In addition, the key generation of the unanimity-based scheme adds a random factor generation step to synchronize the key seed generation across all key managers, and it takes 7.1% of the overall computational time. The quorum-based scheme takes 20.2% of the overall computational time, due to the additional overhead of homomorphic hashing and secret sharing. In general, TED incurs limited computational overhead under attack-resistant key management.

Compared to the evaluation in our conference paper [51], we reduce the chunking time by 25% by replacing Rabin fingerprinting [69] by FastCDC [82]. We also reduce the encryption time by 47%, as we speed up encryption and decryption via hardware acceleration (§5).

TEDStore Instances	Single	Unanimity	Quorum
Chunking	0.60ms		
Fingerprinting	2.3ms		
Hashing	0.19ms		
Random factor generation	-	0.11ms	
Key seeding	0.05ms		0.60ms
Key derivation	0.06ms	0.07ms	0.49ms
Encryption	2.6ms		

Table 1: Experiment B.1: Breakdown of computational time per 1 MiB of uploads in synthetic workloads. In the single-key-manager scheme, the client does not require random factor generation. In both unanimity-based and quorum-based schemes, each key manager executes key seeding individually, and we measure the time of the slowest one in our evaluation.

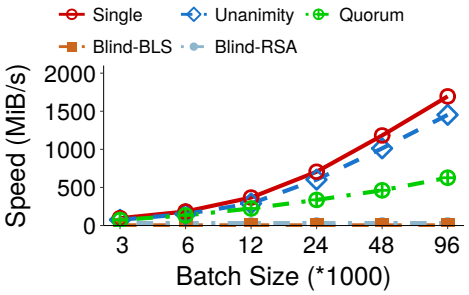


Figure 9: Experiment B.2: Key generation performance.

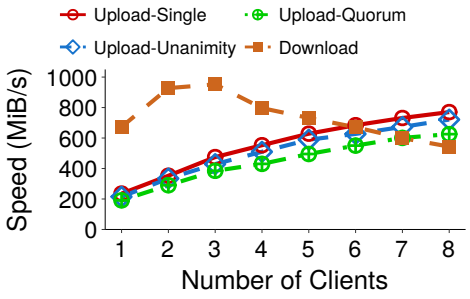


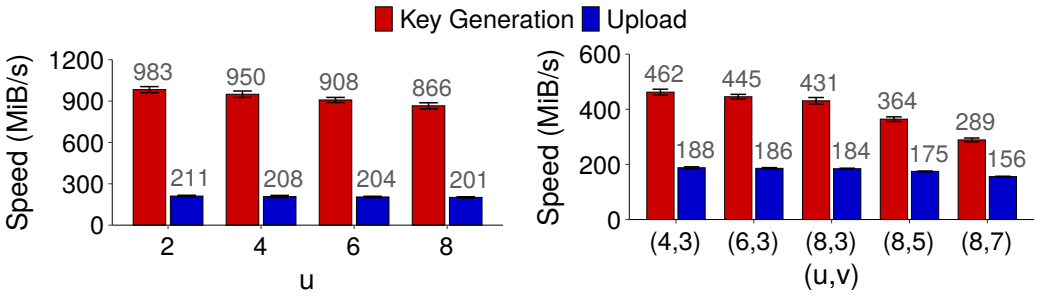
Figure 10: Experiment B.3: Multi-client performance.

Experiment B.2 (Key generation performance). We evaluate the overall key generation performance of TEDStore in a networked setting. We compare the three TED key generation schemes with two blinded key generation protocols (§2.1): (i) *blind RSA* [12] and (ii) *blind BLS* [7]. We focus on a single client, and vary the batch size in key generation (§3.5). We measure the *key generation speed* as the ratio between the file data size (i.e., 2 GiB) and the total running time from when the client computes the short hashes (in TED key generation schemes) or blinded fingerprints (§2.1) (in blind RSA and blind BLS) for all chunks until it obtains all key seeds from the key managers and derives the final encryption keys.

Figure 9 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. All TED key generation schemes achieve much higher key generation speeds than blind RSA and blind BLS, since they use lightweight hash computations instead of expensive blinded key generation operations. For example, when the batch size is 48,000, the unanimity-based and the quorum-based schemes achieve the key generation speeds of 1,012.5 MiB/s, and 462.1 MiB/s, respectively, while those of blind RSA and blind BLS are only 32.5 MiB/s and 2.3 MiB/s, respectively (i.e., at least 31× and 14× speedups in the unanimity-based and quorum-based schemes, respectively). Also, the speeds of all TED key generation schemes increase with the batch size, as we solve the optimization problem fewer times. For example, when the batch size is 96,000, the unanimity-based and quorum-based schemes achieve 1,452.7 MiB/s and 627.2 MiB/s, respectively.

Note that the unanimity-based scheme adds limited key generation performance overhead (e.g., up to 14.3%) over the single-key-manager scheme. The quorum-based scheme incurs extra key generation performance overhead (e.g., up to 56.8%) over the unanimity-based scheme for fault tolerance.

Experiment B.3 (Multi-client performance). We evaluate the performance of TEDStore by having multiple clients upload or download data concurrently. Each client uploads a 2 GiB file of unique data to the provider, and then downloads the 2 GiB file from the provider. We issue the concurrent uploads or downloads



(a) Unanimity-based scheme

(b) Quorum-based scheme

Figure 11: Experiment B.4: The key generation speeds and upload speeds of different cases in the unanimity-based and quorum-based schemes. Each error bar represents the 95% confidence interval based on student's t-distribution over five runs.

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 10 shows the results versus the number of clients (from one to eight). The aggregate upload speeds of the single-key-manager, unanimity-based, and quorum-based schemes increase with the number of clients and finally reach 770.2 MiB/s, 720.9 MiB/s, and 625.5 MiB/s, respectively. In particular, compared to the single-key-manager scheme, the unanimity-based scheme incurs only 6.3-9.7% of speed reduction, while the quorum-based scheme incurs 17.7-22.2% of speed reduction for reconstructing the key of each chunk in secret sharing.

The aggregate download speed (same for all TED key generation schemes) first increases to 951.2 MiB/s for three clients, followed by dropping to 543.8 MiB/s for eight clients due to the read contention and thread context switches across multiple clients. We can improve the download performance by pre-fetching appropriate chunks [21, 53].

Note that we increase the aggregate upload and download speeds of the single-key-manager scheme in our conference version [51] by up to 22.6% and 68.8%, respectively, since we optimize the performance of chunking, encryption, and decryption.

Experiment B.4 (Performance impact of u and v in attack-resistant key management). We evaluate the performance of TEDStore under different configurations of u and v in both the unanimity-based and quorum-based schemes (recall that u is the total number of key managers in both the unanimity-based and quorum-based schemes, while v is the minimum number of available key managers for successful key generation in the quorum-based scheme). We evaluate both the key generation speed (see Experiment B.2 for definition) and the upload speed (see Experiment B.3 for definition).

Figure 11 shows the results. For the unanimity-based scheme (Figure 11(a)), the key generation speed decreases with u, since it incurs more communication overhead for interacting with more key managers. This also leads to a slight drop in the upload speed. For example, when u increases from two to eight, the key generation speed and the upload speed decrease by 11.9% (from 983.2 MiB/s to 865.9 MiB/s) and 5.2% (from 210.8 MiB/s to 201.3 MiB/s), respectively.

For the quorum-based scheme, when we fix $v = 3$ (i.e., the left three groups of bars in Figure 11(b)), both the key generation speed and the upload speed slightly decrease with u, since the client needs to send key generation requests to all key managers. When we fix $u = 8$ (i.e., the right three groups of bars in Figure 11(b)) and increase v from three to seven, the key generation speed and the upload speed decrease by 32.9% (from 430.6 MiB/s to 288.8 MiB/s) and 15.2% (from 184.3 MiB/s to 155.8 MiB/s), respectively. The reason is that the client incurs more modular exponentiations to combine the key seeds.

Experiment B.5 (Performance on Amazon EC2). We evaluate the performance of TEDStore on Amazon EC2 in a geo-distributed environment. We deploy TEDStore in four regions in North America, namely

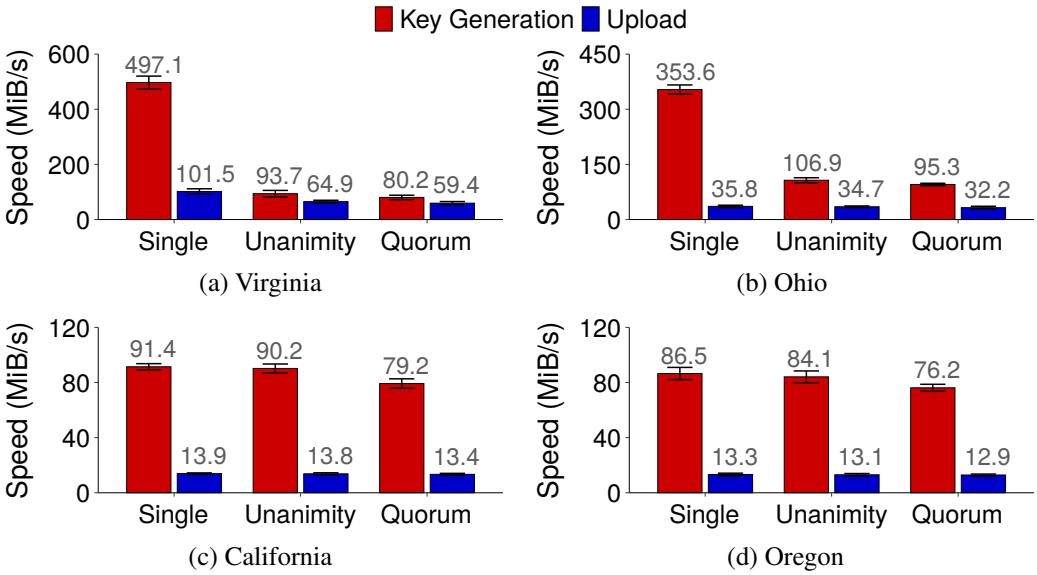


Figure 12: Experiment B.5: the key generation speeds and the upload speeds of TEDStore when the client resides in different regions. Each error bar represents the 95% confidence interval based on student's t-distribution over five runs.

Virginia, Ohio, California, and Oregon. We deploy one EC2 instance per region to host the key manager (i.e., four key managers in total) and a dedicated EC2 instance in Virginia to host the provider. For the single-key-manager scheme, we let the client use the key manager in Virginia. For both the unanimity-based and quorum-based schemes, we use all four key managers. Also, we deploy one EC2 instance in each region as a client. We configure all EC2 instances with type `t2.xlarge`, which has four vCPUs on a 2.3 GHz Intel Xeon E5-2686 CPU and 16 GiB RAM. Each EC2 instance is installed with Linux Ubuntu 16.04. We evaluate the key generation speed and the upload speed of TEDStore when the client is deployed in a different region.

Figure 12 shows the results. The key generation speed of the single-key-manager scheme heavily depends on the region in which the client resides. When both the client and the key manager reside in the same region (i.e., Virginia in Figure 12(a)), the key generation speed of the single-key-scheme reaches 497.1 MiB/s, but drops to 86.5 MiB/s when the client resides in Oregon (Figure 12(d)). On the other hand, both the unanimity-based and quorum-based schemes are less affected by the region in which the client resides, since the client needs to query all key managers and wait for their responses. For example, the key generation speed of the unanimity-based scheme lies in a relatively small range from 84.1 MiB/s to 106.9 MiB/s across different regions.

The key generation speed also affects the corresponding upload speed. For example, for the single-key-manager scheme, the upload speed reaches 101.5 MiB/s when the client resides in Virginia, but is only up to 35.8 MiB/s when the client resides in other regions.

We further provide an upload breakdown as in Experiment B.1 (which considers the computational time in a LAN testbed) with two exceptions: (i) we include the network transmission times for key generation requests and key seeds into the key seeding step, and (ii) we add the *data transmission* step, in which the client uploads the ciphertext chunks to the provider. We measure the processing time breakdown (per 1 MiB of uploads) when the client resides in Virginia and Oregon. Our rationale is to evaluate the cases where the client and the provider reside in the same or different regions.

Table 2 shows the results. For the single-key-manager scheme, data transmission dominates in the overall performance in both regions (e.g., 79.1% of the overall processing time in Oregon). For the unanimity-based and quorum-based schemes, when the client and the provider reside in different regions, data transmission

Steps	Single		Unanimity		Quorum	
	Virginia	Oregon	Virginia	Oregon	Virginia	Oregon
Chunking	1.4±0.1ms					
Fingerprinting	3.3±0.2ms					
Hashing	0.32 ± 0.03ms					
Random factor generation	-		0.30±0.03ms			
Key seeding	1.3±0.2ms	10.8±0.3ms	9.7±0.4ms	11.0±0.6ms	11.7±0.4ms	12.8±0.4ms
Key derivation	0.15±0.01ms		0.17±0.01ms		0.99±0.02ms	
Encryption	3.6±0.1ms					
Data transmission	9.1±0.5ms	74.0±4.7ms	9.2±0.5 ms	75.5±5.2ms	9.2±0.4ms	74.6±4.1ms

Table 2: Experiment B.5: Breakdown of processing time per 1 MiB of uploads in synthetic workloads on Amazon EC2. The variances represent the 95% confidence intervals based on student's t-distribution over five runs.

remains the bottleneck (e.g., 78.9% of the overall processing time for the unanimity-based scheme in Oregon) due to the low cross-region bandwidth. On the other hand, when both the client and the provider reside in the same region, key seeding dominates in the overall performance (e.g., 34.7% of the overall processing time for the unanimity-based scheme in Virginia), since the client needs to interact with the key managers in different regions.

6.3.2 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 §6.1.

Methodology. We deploy TEDStore in a LAN testbed with three machines, all of which run Debian 10.6.0. Specifically, we deploy a client on a machine with a 10-core 2.4 GHz Intel Xeon E5-2640v4 CPU and 64 GiB RAM, up to four key managers bounded to different ports in a machine with two six-core 2.4 GHz Intel Xeon E5-2620v3 CPUs and 32 GiB RAM, and a provider on a machine with a 16-core 2.1 GHz Intel Xeon E5-2683v4 CPU and 64 GiB RAM. The provider machine is attached with a RAID-5 array of four Western Digital Blue 4 TiB 5400 rpm SATA harddisks. All machines are connected with 10 GbE.

Recall that our datasets only contain the fingerprints and sizes of chunks (§6.1). 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.

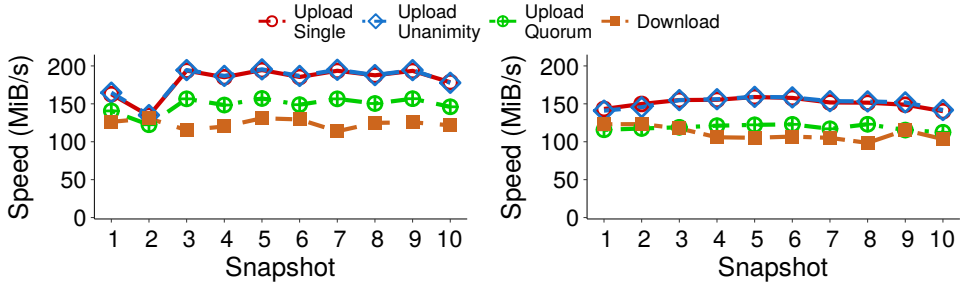
Experiment B.6 (Microbenchmarks). We conduct micro-benchmarks on real-world workloads in a networked setting. For each of the FSL and MS datasets, we choose the snapshot that has the medium size among all snapshots in the dataset. For FSL, the selected snapshot has 116.9 GiB of pre-deduplicated data (or 13.4 M chunks), while for MS, the selected snapshot has 97.8 GiB (or 16.2 M chunks) of pre-deduplicated data. We measure the upload time breakdown as in Experiment B.1 with three exceptions: (i) we do not include chunking due to our trace replay, (ii) we include the time of network communication in key seeding, and (iii) 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 3 presents the time breakdown (per 1 MiB 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 MiB of data. Overall, as in Experiment B.1, the TED key generation schemes of TEDStore remain efficient and do not slow down the overall upload operation.

Experiment B.7 (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

Steps	FSL			MS		
	Single	**Unanimity**	**Quorum**	**Single**	**Unanimity**	**Quorum**
Fingerprinting	2.6ms					
Hashing	0.25ms					
Random factor generation	-	0.14ms		-	0.22ms	
Key seeding	1.3ms	1.5ms	2.3ms	1.5ms	1.7ms	2.7ms
Key derivation	0.06ms	0.06ms	0.41ms	0.10ms	0.10ms	0.69ms
Encryption	3.0ms					
Write	5.4ms			5.8ms		

Table 3: Experiment B.6: Breakdown of processing time per 1 MiB of uploads in real-world workloads.



(a) FSL (b) MS

Figure 13: Experiment B.7: Upload/download speeds in real-world workloads. The x-axis represents the snapshots based on their upload/download orders.

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

Figure 13 shows the upload and download speeds for each snapshot. We observe that the upload speed remains stable for all key generation schemes. For example, for the quorum-based scheme, the upload speed is 122.4-157.0 MiB/s in FSL and 112.5-123.1 MiB/s in MS. Also, compared to the single-key-manager scheme, both the unanimity-based and quorum-based schemes only incur a small speed reduction (e.g., in FSL, 1.9-3.4% for the unanimity-based scheme, and 8.9-19.8% for the quorum-based scheme). Note that the upload speeds of all key generation schemes are lower than those in our evaluation on synthetic workloads (Experiment B.3), mainly due to the fingerprint index access overhead and disk I/O. In addition, the number of fingerprint index entries in the MS dataset is about 1.78× of that in the FSL dataset, so the upload speed of the MS dataset is less than that of the FSL dataset (e.g., by 20.0% under the quorum-based scheme).

The download speed keeps relatively stable in FSL (e.g., ranging from 113.8 MiB/s to 131.0 MiB/s), while having fluctuations in MS (e.g., between 98.5 MiB/s and 123.2 MiB/s). A possible reason is that some MS snapshots have more non-duplicate chunks and are likely to be stored in consecutive regions that can be accessed quickly via sequential reads.

Compared to our conference paper [51], we increase the average upload speeds of the single-key-manager scheme by 36.7% in FSL and 34.1% in MS, since we now apply hardware acceleration technology to encryption (§5). Also, we increase the average download speeds by 109.3% and 132.5% for FSL and MS, respectively, through hardware-accelerated decryption and container-based caching (§5). We can further improve the performance by optimizing the indexing techniques [81, 88], as well as leveraging rewriting and prefetching to mitigate chunk fragmentation [53].

7 Related Work

Encrypted deduplication. MLE [13] formalizes the theoretical framework of encrypted deduplication. Follow-up studies address different aspects of MLE from a theoretical perspective, including parameter dependency [1], data correlation [11], and updates [86]. Liu *et al.* [56] present a generalized security model for encrypted deduplication.

On the applied side, various encrypted deduplication systems (e.g., [2, 5, 23, 25, 40, 67, 75, 80]) realize the MLE construction via convergent encryption (CE) [25]. Some approaches augment CE with secret sharing [52] and transparent metadata management [72]. However, CE is vulnerable to offline brute-force attacks (§2.1).

DupLESS [12] 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 efficient key generation via cross-user file-level deduplication [87], and decentralized key agreement among users without a dedicated key manager [55]. Some studies augment encrypted deduplication with new functionalities, such as periodic verification of storage space [7], dynamic access control [68], bandwidth-efficient uploads [24], or space-efficient metadata management [47]. In particular, Duan [26] and Metadedup [47] propose fault-tolerant key management similar to our quorum-based scheme, but both approaches prohibit the frequency counting of chunks as they send blinded fingerprints for key generation. Note that 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 [12] can infer the original plaintext chunk of a ciphertext chunk by testing all candidate plaintext chunks, or learn the remaining content of a file [79]. Side-channel attacks [8, 24, 33, 34, 66, 89] can infer the content of an already stored file by examining if a chunk can be deduplicated via client-side deduplication. Ritzdorf *et al.* [71] 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 [71].

Our work focuses on defending against frequency analysis in encrypted deduplication. Li *et al.* [50] show how to increase the inference rate of frequency analysis (from an adversarial perspective) by exploiting chunk locality [54, 88]. 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 [42] as well as Lewi and Wu [46] propose to hide attribute frequencies by randomizing ciphertexts. Frequency-smoothing encryption [45] 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 [42, 46], or incur high performance overhead by using computational expensive cryptographic primitives (e.g., homomorphic encoding) [45].

8 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 extend TED with two attack-resilient key generation schemes, namely the unanimity-based scheme and the quorum-based scheme, so as to defend against the

compromise of a single key manager. The unanimity-based scheme has limited performance overhead, while the quorum-based scheme provides fault tolerance for the key managers with additional performance overhead. 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.

References

[1] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In *Proceedings of the Annual Cryptology Conference (CRYPTO'13)*, pages 374–391, 2013.

[2] 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 *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 1–14, 2002.

[3] I. A. Al-Kadit. Origins of cryptology: The arab contributions. *Cryptologia*, 16(2):97–126, 1992.

[4] G. Amvrosiadis and M. Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 151–164, 2015.

[5] P. Anderson and L. Zhang. Fast and secure laptop backups with encrypted de-duplication. In *Proceedings of the 24th USENIX International Conference on Large Installation System Administration (LISA'10)*, pages 1–8, 2010.

[6] A. Appleby. SMHasher. Retrieved from https://github.com/aappleby/smhasher, 2022.

[7] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 886–900, 2015.

[8] F. Armknecht, C. Boyd, G. T. Davies, K. Gjøsteen, and M. Toorani. Side channels in deduplication: Trade-offs between leakage and efficiency. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*, pages 266–274, 2017.

[9] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS'07)*, pages 598–609, 2007.

[10] T. Baignères, P. Junod, and S. Vaudenay. How far can we go beyond linear cryptanalysis? In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'04)*, pages 432–450, 2004.

[11] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Proceedings of the IACR International Workshop on Public Key Cryptography (PKC'15)*, pages 516–538, 2015.

[12] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proceedings of the 22nd USENIX Security Symposium (Security'13)*, pages 179–194, 2013.

[13] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'13)*, pages 296–312, 2013.

[14] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):1–33, 2013.

[15] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 169–180, 2014.

[16] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'09)*, pages 1–9, 2009.

[17] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment*, 11(11):1715–1728, 2018.

[18] J. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATC'06)*, pages 85–90, 2006.

[19] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[20] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Conference on Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1997.

[21] 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 *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 309–324, 2018.

[22] 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.

[23] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 285–298, 2002.

[24] H. Cui, C. Wang, Y. Hua, Y. Du, and X. Yuan. A bandwidth-efficient middleware for encrypted deduplication. In *Proceedings of the 2018 IEEE Conference on Dependable and Secure Computing (DSC'18)*, pages 1–8, 2018.

[25] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS'02)*, pages 617–624, 2002.

[26] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *Proceedings of the 2014 ACM on Cloud Computing Security Workshop (CCSW'14)*, pages 57–68, 2014.

[27] 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.

[28] File System and Storage Lab at Stony Brook University. FSL traces and snapshots public archive. Retrieved from http://tracer.filesystems.org/, 2022.

[29] S. Ghemawat and J. Dean. LevelDB. Retrieved from https://github.com/google/leveldb, 2022.

[30] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM SIGSAC Conference on Computer and Communications Security (CCS'06)*, pages 89–98, 2006.

[31] T. Granlund. GNUMP: GNU multiple precision arithmetic library. Retrieved from https://gmplib.org/, 2022.

[32] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P'17)*, pages 655–672, 2017.

[33] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, pages 491–500, 2011.

[34] 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.

[35] HIPAA Journal. Hard drive theft sees data of 1 million individuals exposed. Retrieved from https://www.hipaajournal.com/hard-drive-theft-sees-data-1-million-individuals-exposed-8859/, 2022.

[36] IDC. Data age 2025. Retrieved from https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2022.

[37] Intel Corporation. Intel advanced encryption standard (AES) new instructions set. Retrieved from https://www.intel.com.bo/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf, 2022.

[38] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the 2009 ACM International Conference on Systems and Storage (SYSTOR'09)*, pages 1–12, 2009.

[39] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS'07)*, pages 584–597, 2007.

[40] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'03)*, pages 29–42, 2003.

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

[42] F. Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 656–667, 2015.

[43] M. N. Krohn, M. J. Freedman, and D. Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy (S&P'04)*, pages 226–240, 2004.

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

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

[46] K. Lewi and D. J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, pages 1167–1178, 2016.

[47] J. Li, S. Huang, Y. Ren, Z. Yang, P. P. Lee, X.-s. Zhang, and Y. Hao. Enabling secure and space-efficient metadata management in encrypted deduplication. *IEEE Transactions on Computers*, pages 1–1, 2021.

[48] J. Li, P. P. Lee, C. Tan, C. Qin, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. *ACM Transactions on Storage*, 16(1):1–30, 2020.

[49] J. Li, C. Qin, P. P. C. Lee, and J. Li. Rekeying for encrypted deduplication storage. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, pages 618–629, 2016.

[50] J. Li, C. Qin, P. P. C. Lee, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*, pages 1–12, 2017.

[51] J. Li, Z. Yang, Y. Ren, P. P. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*, pages 1–15, 2020.

[52] M. Li, C. Qin, and P. P. C. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 111–124, 2015.

[53] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 183–197, 2013.

[54] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 111–123, 2009.

[55] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 874–885, 2015.

[56] J. Liu, L. Duan, Y. Li, and N. Asokan. Secure deduplication of encrypted data: Refined model and new constructions. In *Proceedings of the 2018 Cryptographers' Track at the RSA Conference (CT-RSA'18)*, pages 374–393, 2018.

[57] D. Meister, A. Brinkmann, and T. Süß. File recipe compression in data deduplication systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 175–182, 2013.

[58] R. Mendes, T. Oliveira, V. V. Cogo, N. F. Neves, and A. N. Bessani. Charon: A secure cloud-of-clouds system for storing and sharing big data. *IEEE Transactions on Cloud Computing*, 9(4):1349–1361, 2021.

[59] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage*, 7(4):1–20, 2011.

[60] 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 *Proceedings of the 20th USENIX Conference on Security (Security'11)*, pages 5–5, 2011.

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

[62] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 644–655, 2015.

[63] OpenSSL. OpenSSL: Cryptography and SSL/TLS toolkit. Retrieved from https://www.openssl.org/, 2022.

[64] P. Paillier. Low-cost double-size modular exponentiation or how to stretch your cryptoprocessor. In *Proceedings of the IACR International Workshop on Public Key Cryptography (PKC'99)*, pages 223–234, 1999.

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

[66] Z. Pooranian, K.-C. Chen, C.-M. Yu, and M. Conti. RARE: Defeating side channels based on data-deduplication in cloud storage. In *Proceedings of the 2018 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS'18)*, pages 444–449, 2018.

[67] P. Puzio, R. Molva, M. Önen, and S. Loureiro. ClouDedup: Secure deduplication with encrypted data for cloud storage. In *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'13)*, pages 363–370, 2013.

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

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

[70] J. Resch and J. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, pages 14–14, 2011.

[71] H. Ritzdorf, G. O. Karame, C. Soriente, and S. Čapkun. On information leakage in deduplicated storage systems. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop (CCSW'16)*, pages 61–72, 2016.

[72] P. Shah and W. So. Lamassu: Storage-efficient host-side encryption. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 333–345, 2015.

[73] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[74] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In *Proceedings of International Conference on Financial Cryptography and Data Security (FC'14)*, pages 99–118, 2014.

[75] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS'08)*, pages 1–10, 2008.

[76] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS-a secure, recoverable, long-term archival storage system. *ACM Transactions on Storage*, 5(2):1–35, 2009.

[77] 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 *Proceedings of the 32nd IEEE Symposium on Mass Storage Systems and Technologies (MSST'16)*, pages 1–7, 2016.

[78] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, pages 4–4, 2012.

[79] Z. Wilcox-O'Hearn. Drew perttula and attacks on convergent encryption. Retrieved from https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html, 2022.

[80] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS'08)*, pages 21–26, 2008.

[81] 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 *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*, pages 26–30, 2011.

[82] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. FastCDC: a fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 101–114, 2016.

[83] Y. Yang and J. Zhu. Write skew and zipf distribution: Evidence and implications. *ACM Transactions on Storage*, 12(4):1–19, 2016.

[84] W. Zhang, D. Agun, T. Yang, R. Wolski, and H. Tang. VM-centric snapshot deduplication for cloud data backup. In *Proceedings of the 31st IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, pages 1–12, 2015.

[85] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng. Multi-level selective deduplication for VM snapshots in cloud storage. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD'12)*, pages 550–557, 2012.

[86] Y. Zhao and S. S. Chow. Updatable block-level message-locked encryption. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*, pages 449–460, 2017.

[87] 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 *Proceedings of the 31st IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, pages 1–14, 2015.

[88] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, pages 269–282, 2008.

[89] 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 *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, pages 1153–1162, 2018.

Appendix: Optimization for Automated Parameter Configuration

We present the detailed algorithm of finding the set of frequencies of ciphertext chunks, i.e., $\{f_i^*\}_{i=1}^{n^*}$, by solving the optimization problem in Equation (6), as shown in §3.5. To solve the optimization problem, we leverage the classical simplex algorithm [19], which finds the solution by performing an iterative search in extreme points.

Algorithm 1 presents the pseudo-code of the algorithm. It takes the frequencies $\{f_i\}_{i=1}^{n}$ of plaintext chunks and the number n^* of distinct ciphertext chunks as inputs. Initially, it assigns each f_i^* with the extreme point $\sum_{i=1}^{n} f_i/n^*$ (Lines 2-5). It then performs iterative adjustment by traversing i from 1 to n. If $f_i^* > f_i$, it sets $f_i^* = f_i$ to meet the constraint $f_i^* \leq f_i$ (Line 8) and updates $\{f_j^*\}_{j=i+1}^{n^*}$ for $i+1 \leq j \leq n^*$ (Lines 9-12). Otherwise, if $f_i^* \leq f_i$, it leaves the for-loop (Line 14). The algorithm finally returns the frequency distribution of ciphertext chunks, as given by Equation (7) (Line 17).

We prove that Algorithm 1 provides an optimal solution to the problem in Equation (6).

Theorem 1. *The solution of Algorithm 1, as shown in Equation (7), is a globally optimal solution to the problem in Equation (6)).*

Proof. We simplify our optimization problem and show that the solution in Equation (7) satisfies the Karush-Kuhn-Tucker (KKT) conditions [19] of the problem.

Algorithm 1 Optimization for Automated Parameter Configuration

```
procedure MAIN({f_i}_{i=1}^n, n*)
    F ← Σ_{i=1}^n f_i
    for i ← 1 to n* do
        f_i* ← F / n*
    end for
    for i ← 1 to n do
        if f_i* > f_i then
            f_i* ← f_i
            F ← F − f_i*
            for j ← i + 1 to n* do
                f_j* ← F / (n* − i)
            end for
        else
            break
        end if
    end for
    return {f_i*}_{i=1}^{n*}
end procedure
```

Let $\sum_{i=1}^{n^*} f_i^* = \sum_{i=1}^{n} f_i = F$ be the total number of plaintext chunks (which is also the total number of ciphertext chunks). We expand the objective function of the optimization problem as follows:

$$\begin{aligned}\mathbf{KLD}(\{f_i^*\}_{i=1}^{n^*}) &= \mathbf{log}n^* + \sum\nolimits_{i=1}^{n^*} \frac{f_i^*}{F}\mathbf{log}\frac{f_i^*}{F} \\ &= \mathbf{log}n^* - \mathbf{log}F + \frac{1}{F}\sum\nolimits_{i=1}^{n^*} f_i^*\mathbf{log}f_i^* \\ &= \mathbf{log}n^* - \mathbf{log}F - \mathbf{ln}2 + \frac{1}{F}\sum\nolimits_{i=1}^{n^*} f_i^*\mathbf{ln}f_i^*.\end{aligned} \tag{18}$$

Since $(\mathbf{log}n^* - \mathbf{log}F - \mathbf{ln}2)$ and F are constants, we can reduce the objective of the optimization problem to minimizing the objective function $\mathbf{F}(\{f_i^*\}_{i=1}^{n^*}) = \sum_{i=1}^{n^*} f_i^*\mathbf{ln}f_i^*$. Thus, we express the optimization problem as follows:

$$\begin{aligned}&\text{minimize } \mathbf{F}(\{f_i^*\}_{i=1}^{n^*}) \\ &\text{subject to } \mathbf{P}(\{f_i^*\}_{i=1}^{n^*}) = 0 \text{ and} \\ &\qquad \mathbf{Q}_j(\{f_i^*\}_{i=1}^{n^*}) \leq 0 \text{ for } \forall j \in [1, n+n^*],\end{aligned} \tag{19}$$

where

$$\mathbf{P}(\{f_i^*\}_{i=1}^{n^*}) = \sum\nolimits_{i=1}^{n^*} f_i^* - \sum\nolimits_{i=1}^{n} f_i, \tag{20}$$

$$\mathbf{Q}_j(\{f_i^*\}_{i=1}^{n^*}) = \begin{cases} -f_j^*, & 1 \leq j \leq n^*, \\ f_{j-n^*}^* - f_{j-n^*}, & n^*+1 \leq j \leq n+n^*. \end{cases} \tag{21}$$

Since equality constraint function $\mathbf{P}(\cdot)$ and inequality constraint functions $\{\mathbf{Q}_j(\cdot)\}_{j=1}^{n+n^*}$ are affine functions, the optimization problem satisfies the condition of linearity constraints.

Let $x^* = \{f_i^*\}_{i=1}^{n^*}$ be the solution of Algorithm 1 in the form of Equation (7). We can find the constant KKT multipliers β and $\{\mu_j\}_{j=1}^{n+n^*}$ as:

$$\beta = -(1 + \mathbf{ln}f_{m+1}^*), \tag{22}$$

$$\mu_j = \begin{cases} 0, & 1 \le j \le n^*, \\ \mathbf{ln} f^*_{m+1} - \mathbf{ln} f^*_{j-n^*}, & n^*+1 \le j \le n^*+m, \\ 0, & n^*+m+1 \le j \le n^*+n, \end{cases} \tag{23}$$

where m is the maximum integer such that $f_m \le \frac{\sum_{i=m+1}^{n} f_i}{n^*-m}$. Based on these coefficients, we will verify the KKT conditions of x^* listed in Equations (24), (25), and (26):

$$\mathbf{P}(x^*) = 0, \tag{24}$$

$$\sum\nolimits_{j=1}^{n+n^*} \mu_j \mathbf{Q}_j(x^*) = 0, \tag{25}$$

$$\frac{\partial \mathbf{F}}{\partial f_i^*} + \sum\nolimits_{j=1}^{n+n^*} \mu_j \frac{\partial \mathbf{Q}_j}{\partial f_i^*} + \beta \frac{\partial \mathbf{P}}{\partial f_i^*} = 0, \ \ \forall i \in [1, n^*]. \tag{26}$$

To verify Equation (24), we expand $\mathbf{P}(x^*)$:

$$\begin{aligned} \mathbf{P}(x^*) &= \sum\nolimits_{i=1}^{m} f_i^* + \sum\nolimits_{i=m+1}^{n^*} \frac{\sum_{j=m+1}^{n} f_j}{n^*-m} - \sum\nolimits_{i=1}^{n} f_i \\ &= \sum\nolimits_{i=1}^{m} f_i^* + \sum\nolimits_{j=m+1}^{n} f_j - \sum\nolimits_{i=1}^{n} f_i. \end{aligned} \tag{27}$$

Since $f_i^* = f_i$ for $1 \le i \le m$ in x^* (see Equation (7)), we have $\mathbf{P}(x^*) = 0$.

To verify Equation (25), we only need to consider the case $n^*+1 \le j \le n^*+m$, in which $\mu_j \ne 0$. Specifically, we have:

$$\mu_j \mathbf{Q}_j(x^*) = (\mathbf{ln} f^*_{m+1} - \mathbf{ln} f^*_{j-n^*})(f^*_{j-n^*} - f_{j-n^*}). \tag{28}$$

Note that for x^*, in this case $1 \le j - n^* \le m$, we have $f^*_{j-n^*} - f_{j-n^*} = 0$ and hence $\mu_j \mathbf{Q}_j(x^*) = 0$. Thus, $\sum_{j=1}^{n+n^*} \mu_j \mathbf{Q}_j(x^*) = 0$.

To verify Equation (26), we first compute the component:

$$\frac{\partial \mathbf{Q}_j}{\partial f_i^*} = \begin{cases} -1, & j = i, \\ 1, & j = n^*+i, \\ 0, & \text{otherwise.} \end{cases} \tag{29}$$

Since $\mu_j = 0$ for $1 \le j \le n^*$ (see Equation (23)), we have $\mu_j \frac{\partial \mathbf{Q}_j}{\partial f_i^*} = \mathbf{ln} f^*_{m+1} - \mathbf{ln} f^*_{j-n^*}$ only when $j = n^*+i$. Then we can verify Equation (26):

$$\begin{aligned} \frac{\partial \mathbf{F}}{\partial f_i^*} + \sum\nolimits_{j=1}^{n+n^*} \mu_j \frac{\partial \mathbf{Q}_j}{\partial f_i^*} + \beta \frac{\partial \mathbf{P}}{\partial f_i^*} &= \mathbf{ln} f_i^* + 1 + \mathbf{ln} f^*_{m+1} - \mathbf{ln} f^*_{j-n^*} + \beta \\ &= 0. \end{aligned} \tag{30}$$

Finally, since x^* satisfies all KKT conditions shown in Equations (24), (25), and (26), we conclude that x^* is the globally optimal solution of the optimization problem. □