Enabling Secure and Space-Efficient Metadata Management in Encrypted Deduplication

Jingwei Li, Suyu Huang, Yanjing Ren, Zuoru Yang, Patrick P. C. Lee, Xiaosong Zhang, and Yao Hao

Abstract—Encrypted deduplication combines encryption and deduplication in a seamless way to provide confidentiality guarantees for the physical data in deduplicated storage, yet it incurs substantial metadata storage overhead due to the additional storage of keys. We present a new encrypted deduplication storage system called Metadedup, which suppresses metadata storage by also applying deduplication to metadata. Its idea builds on indirection, which adds another level of metadata chunks that record metadata information. We find that metadata chunks are highly redundant in real-world workloads and hence can be effectively deduplicated. We further extend Metadedup to incorporate multiple servers via a distributed key management approach, so as to provide both fault-tolerant storage and security gaurantees. We extensively evaluate Metadedup from performance and storage efficiency perspectives. We show that Metadedup achieves high throughput in writing and restoring files, and saves the metadata storage by up to 93.94% for real-world backup workloads.

Index Terms—Encrypted deduplication, metadata management, cloud storage

1 Introduction

Chunk-based deduplication is widely used in modern primary [31] and backup [26], [39], [42] storage systems to achieve high storage savings. It stores only a single physical copy of duplicate chunks, while referencing all duplicate chunks to the physical copy by small-size references. Prior studies show that deduplication can effectively reduce the storage space of primary storage by 50% [31] and that of backup storage by up to 98% [39]. This motivates the wide deployment of deduplication in various commercial cloud storage services (e.g., Dropbox, Google Drive, Bitcasa, Mozy, and Memopal) to reduce substantial storage costs [18].

To provide confidentiality guarantees, *encrypted deduplication* [7], [8] adds an encryption layer to deduplication, such that each chunk, before being written to deduplicated storage, is deterministically encrypted via symmetric-key encryption by a key derived from the chunk content (e.g., the key is set to be the cryptographic hash of chunk content [14]). This ensures that duplicate chunks have identical content even after encryption, and hence we can still apply deduplication to the encrypted chunks for storage savings. Many studies (e.g., [5], [7], [25], [33], [36]) have designed various encrypted deduplication schemes to efficiently manage outsourced data in cloud storage.

In addition to storing non-duplicate data, a deduplicated storage system needs to keep *deduplication metadata*. There are two types of deduplication metadata. To check if identical chunks exist, the system maintains a *fingerprint index* that tracks the fingerprints of all chunks that have already been stored. Also, to allow a file to be reconstructed, the system maintains a *file recipe* that holds the mappings from the chunks in the file to the references of the corresponding physical copies.

Deduplication metadata is notoriously known to incur high storage overhead [11], [21], [30], especially for the highly redundant workloads (e.g., backups) as the metadata storage overhead becomes more dominant. In this work, we argue that encrypted deduplication incurs even higher metadata storage overhead, as it additionally keeps *key metadata*, such as the key recipes that track the chunk-to-key mappings to allow the decryption of individual files. Since the key recipes contain sensitive key information, they need to be managed separately from file recipes, encrypted by the master keys of file owners, and individually stored for different file owners. Such high metadata storage overhead can negate the storage effectiveness of encrypted deduplication in real deployment.

Contributions. To address the storage overhead of both deduplication metadata and key metadata, we design and implement Metadedup, a new encrypted deduplication system that effectively suppresses metadata storage. Our contributions are summarized as follows.

- Metadedup builds on the idea of *indirection*. Instead of directly storing all deduplication and key metadata in both file and key recipes (both of which dominate the metadata storage overhead), we group the metadata in the form of *metadata chunks* that are stored in encrypted deduplication storage. Thus, both file and key recipes now store references to metadata chunks, which now

- *An earlier conference version of this paper appeared at the 35th International Conference on Massive Storage Systems and Technology (MSST 2019) [23]. In this journal version, we propose a distributed key management approach to improve security and fault tolerance. We further evaluate our new distributed key management approach in terms of write/restore performance, data/metadata storage overheads, and storage load balance.*
- *J. Li, S. Huang, Y. Ren and X. Zhang are with the Center for Cyber Security, University of Electronic Science and Technology of China, China (Emails: {jwli, johnsonzxs}@uestc.edu.cn, {gabrielf977q, tinoryj}@gmail.com).*
- *Z. Yang and P. Lee are with Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (Emails: {zryang,pclee}@cse.cuhk.edu.hk).*
- *Y. Hao is with Science and Technology on Communication Security Laboratory, China, and also with China Electronics Technology Cyber Security Co., Ltd, China (Email: haoyao30@163.com).*
- *Corresponding author: Patrick P. C. Lee.*

contain references to *data chunks* (i.e., the chunks of file data). If Metadedup stores nearly identical files regularly (e.g., periodic backups [39]), the corresponding file and key metadata are expected to have long sequences of references that are in the same order. This implies that the metadata chunks are highly redundant and hence can be effectively deduplicated.

- We propose a distributed key management approach that adapts Metadedup into a multi-server architecture for fault-tolerant data storage. We generate the key of each data chunk from one of the servers, encode it via secret sharing, and distribute the resulting shares to remaining servers. This ensures fault-tolerant storage of data chunks, while being robust against adversarial compromise on a number of servers through our decoupled management of keys and shares.
- We implement a Metadedup prototype and evaluate its performance in a networked setup. Compared to the network speed of our Gigabit LAN testbed, we show that Metadedup incurs only 13.09% and 3.06% of throughput loss in writing and restoring files, respectively.
- Finally, we conduct trace-driven simulation on two real-world datasets. We show that Metadedup achieves up to 93.94% of metadata storage savings in encrypted deduplication. We also show that Metadedup maintains the storage load balance among all servers.

The source code of our Metadedup prototype is now available at **http://adslab.cse.cuhk.edu.hk/software/metadedup**.

The rest of this paper proceeds as follows. Section 2 motivates the need of mitigating metadata storage overhead in encrypted deduplication via mathematical analysis and trace-driven simulation. Section 3 reviews related work. Section 4 presents the design of Metadedup. Section 5 extends Metadedup with distributed key management. Section 6 presents the implementation details of our Metadedup prototype. Section 7 presents our evaluation results. Finally, Section 8 concludes this paper. In the digital supplementary file, we present security analysis on Metadedup and additional evaluation results on the storage usage of Metadedup.

2 Background and Motivation

2.1 Encrypted Deduplication Storage

Deduplication is a technique for space-efficient data storage (see [40] for a complete survey of deduplication). It partitions file data into either fixed-size or variable-size chunks, and identifies each chunk by the cryptographic hash, called *fingerprint*, of the corresponding content. Suppose that the probability of fingerprint collision against different chunks is practically negligible [10]. Deduplication stores only one physical copy of duplicate chunks, and refers the duplicate chunks that have the same fingerprint to the physical copy by small references.

Encrypted deduplication augments *plain deduplication* (i.e., deduplication without encryption) with an encryption layer that operates on the chunks before deduplication, and provides data confidentiality guarantees in deduplicated storage. It implements the encryption layer based on *message-locked encryption (MLE)* [7], [8], which encrypts each chunk with a symmetric key (called the *MLE key*) derived from the chunk content; for example, the MLE key can be computed as the cryptographic hash of the chunk content in convergent encryption [14]. This ensures that the encrypted chunks derived from duplicate chunks still have identical content, thereby being compatible with deduplication.

Historical MLE [8] builds on some *publicly available* function (e.g., cryptographic hash function [14]) to generate MLE keys. It provides security protection for *unpredictable* chunks, meaning that the chunks are drawn from a sufficiently large message set, such that the content of a chunk cannot be easily predicted; otherwise, if a chunk is predictable and known to be drawn from a finite set, historical MLE is vulnerable to the *offline brute-force attack* [8]. Specifically, given a target encrypted chunk, an adversary samples each possible chunk from the finite message set, derives the corresponding MLE key (e.g., by applying the cryptographic hash function to each sampled chunk [14]), and encrypts each sampled chunk with such a key. If the encryption result is equal to the target encrypted chunk, the adversary can infer that the sampled chunk is the original input of the target encrypted chunk.

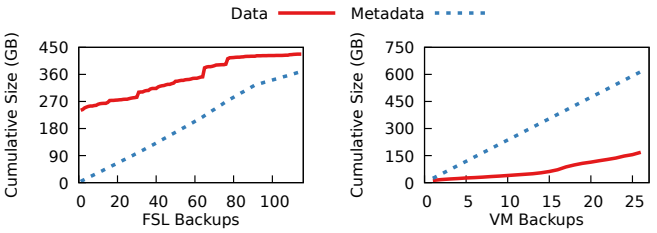
To defend against the offline brute-force attack, DupLESS [7] implements *server-aided MLE*, which introduces a global secret and protects the key generation process against public access. Server-aided MLE leverages a dedicated *key manager* to maintain a *global secret* that is protected from an adversary. It derives the MLE key of each chunk based on both the global secret and the cryptographic hash (a.k.a., fingerprint) of the chunk, such that an adversary cannot feasibly derive the MLE keys of any sampled chunks without knowing the global secret. Thus, if the global secret is secure, server-aided MLE is robust against the offline brute-force attack, and achieves security for both predictable and unpredictable chunks; otherwise, if the global secret is compromised, it preserves the same security guarantees for unpredictable chunks as in historical MLE [8]. DupLESS [7] further implements two mechanisms to strengthen security: (i) the *oblivious pseudorandom function (OPRF)* protocol, which allows the client to submit the fingerprint of each chunk (to the key manager) in a *blinded* manner, such that the key manager can successfully generate MLE keys without learning the actual fingerprints; and (ii) the *rate-limiting* mechanism, which proactively controls the key generation rate of the key manager, so as to defend the *online brute-force attack* from the compromised clients that try to issue too many key generation requests.

In this paper, we focus on mitigating the metadata storage overhead in MLE-based (including both historical MLE and server-aided MLE) encrypted deduplication storage systems.

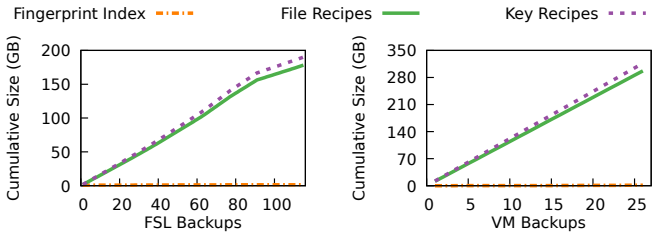
2.2 Metadata Storage Overhead

Section 1 reviewed the metadata components (i.e., deduplication metadata and key metadata) of encrypted deduplication. We now show the high metadata storage overhead in encrypted deduplication via both mathematical analysis and trace-driven simulation.

Mathematical analysis. We first model the metadata storage overhead in encrypted deduplication. We refer to the data before and after deduplication as *logical data* and *physical data*, respectively. Suppose that L is the size of logical data, P is the size of physical data, and f is the ratio of the deduplication metadata size to the chunk size. Plain deduplication incurs



(a) Physical data vs. metadata in FSL and VM datasets



(b) Breakdown of metadata in FSL and VM datasets

Fig. 1: Cumulative data and metadata storage of encrypted deduplication in two real-world datasets of backup workloads FSL and VM. The x-axis shows the number of FSL/VM backups issued to encrypted deduplication storage, and the y-axis shows the cumulative data/metadata sizes after issuing each backup.

$f \times (L+P)$ of metadata storage [38], [39], where $f \times L$ is the size of file recipes and $f \times P$ is the size of the fingerprint index. Encrypted deduplication has additional metadata storage for keys. It incurs a total of $f \times (L+P) + k \times L$ of metadata storage, where k is the ratio of the key metadata size to the chunk size, and $k \times L$ is the size of key recipes.

Based on the above analysis, we show via an example how the metadata storage overhead becomes problematic in encrypted deduplication. Suppose that the size of deduplication metadata is 30 bytes [39], the size of key metadata is 32 bytes (e.g., for AES-256 encryption keys), and the chunk size is 8 KB [39]. Then $f = \frac{30 \text{ bytes}}{8 \text{ KB}} \approx 0.0037$ and $k = \frac{32 \text{ bytes}}{8 \text{ KB}} \approx 0.0039$. If the deduplication factor (i.e., L/P) is 50× [39] and $L = 50$ TB, then plain deduplication and encrypted deduplication incur 191.25 GB and 391.25 GB of metadata, or equivalently 18.67% and 38.21% additional storage for 1 TB of physical data, respectively.

Trace-driven simulation. Our trace-driven simulation on two real-world datasets of backup workloads, namely FSL and VM (see Section 7.2 for the dataset details), further validates the high metadata storage overhead in encrypted deduplication. As in our mathematical analysis, we set the size of deduplication metadata as 30 bytes per chunk and the size of key metadata as 32 bytes per chunk.

We measure the cumulative metadata storage as we issue backups to encrypted deduplication storage. Figure 1(a) shows that the cumulative size of metadata (including the fingerprint index, file recipes, and key recipes) increases with the number of backups, and even exceeds that of physical data in the VM dataset. For example, after 26 VM backups, the cumulative data and metadata consume 168.24 GB and 615.18 GB, respectively. Figure 1(b) further presents the breakdown of metadata storage. We observe that the dominant components are the file recipes and key recipes, which contribute to 99.58% and 99.81% of the overall metadata storage in the FSL and VM datasets, respectively.

3 Related Work

Some studies organize metadata in efficient ways to improve deduplication performance [9], [26], [29], [42] or storage efficiency [27]. For example, DDFS [42], Sparse Indexing [26], and Extreme Binning [9] are designed to effectively cache a subset of fingerprint index entries, and mitigate the performance bottleneck of disk access to the fingerprint index on disk. Mandal *et al.* [29] transfer application metadata to block-layer deduplication, so as to accelerate the deduplication speed. Lin *et al.* [27] separate metadata from data to improve the storage efficiency of deduplication. While the above studies address metadata management, they do not consider how to mitigate metadata storage overhead in deduplication.

Considering the high metadata storage overhead, several studies reduce the amount of metadata in plain deduplication. We discuss their limitations in encrypted deduplication.

- **Grouping and re-chunking.** Fingerdiff [11] starts with small chunks, and groups adjacent duplicate small chunks into a big chunk for space-efficient metadata management. FBC [28] and Subchunk [35] apply deduplication on big chunks to reduce the amount of deduplication metadata, and re-chunk the non-duplicate big chunks into small ones for fine-grained deduplication. Bimodal [21] generalizes grouping and re-chunking to operate in data regions. However, these approaches depend on the prior knowledge of deduplication results (e.g., whether some chunks are duplicates), which can be abused to extract secret information [17], [18], [32] against encrypted deduplication. In addition, they cannot compress key metadata, since each chunk still needs to be encrypted by the key derived from its own content.
- **Compression.** Meister *et al.* [30] propose four approaches to replace the fingerprints in file recipes by short codewords, so as to compress deduplication metadata (see Section 7.2 for details). However, they either cannot apply to the key recipe that is encrypted by the file owner's master key, or only reduce the metadata of zero chunks.
- **Key reduction.** Dekey [22] applies deduplication to the keys directly to reduce the amount of key metadata. However, since the size of a key is often comparable to the size of the additional reference (both are of tens of bytes) to the corresponding physical copy, the storage saving of key metadata can be negated by such additional deduplication metadata in key-based deduplication. SecDep [41] and REED [33] generate one MLE key for a group of chunks to reduce the total number of keys. However, since duplicate chunks are possibly encrypted with different keys (i.e., the resulting encrypted chunks become different and cannot be deduplicated), these systems [33], [41] degrade the storage efficiency achieved by deduplication.

This paper is also related to some existing encrypted deduplication designs. Lamassu [36] implements transparent metadata management in encrypted deduplication. It places metadata into some reserved sections of file data, so as to be compatible with different applications. However, these metadata sections are randomly encrypted, and cannot be deduplicated along with data for storage savings. TEDStore

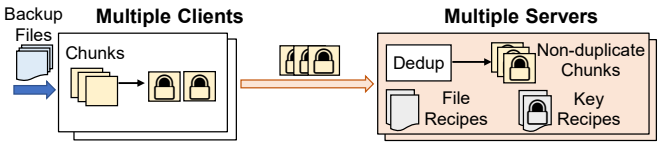


Fig. 2: Metadedup architecture. In this section, we focus on a single server for data and metadata management, and discuss the extension to multiple servers in Section 5.

[24] balances the storage efficiency and data confidentiality via tunable key management in encrypted deduplication, while this paper targets metadata storage.

4 METADEDUP

Metadedup is designed for an organization that outsources the storage of users' data to a remote shared storage system. It focuses on the storage of backup workloads, which are known to have high content similarity [39]. It applies deduplication to remove content redundancies of both data (i.e., the file data from users' backup workloads) and metadata (i.e., deduplication metadata and key metadata), so as to improve the overall storage efficiency.

Metadedup builds on the client-server model (see Figure 2). A *client* is a software interface for users to process backup files. It partitions a backup file into multiple chunks, encrypts each chunk, and uploads the encrypted chunks to a remote *server* that employs encrypted deduplication. The server performs chunk-based deduplication, and only stores the non-duplicate chunks that have unique content with existing chunks. It also keeps the file recipe and key recipe of each backup file (note that the key recipe is further encrypted by the master key of the client that owns the backup file) for the reconstruction of the backup file. Here, we assume that the communication channels are carefully protected (e.g., via SSL/TLS), so as to address network eavesdropping. To this end, Metadedup aims for the following goals:

- **Low storage overhead of metadata.** It suppresses the storage space of both file recipes and key recipes (which dominate the metadata storage overhead as shown in Section 2.2), while incurring only small storage overhead to the fingerprint index as we also apply deduplication to metadata.
- **Security for data and metadata.** It preserves the security guarantees of underlying encrypted deduplication for both data and metadata storage.
- **Limited performance overhead.** It adds small performance overhead on writing (restoring) files to (from) deduplicated storage.

In the following, we define the threat model of Metadedup, and present its metadata deduplication design in details. We also present the security analysis of Metadedup on the protection of metadata chunks in Section 1 of the digital supplementary file.

4.1 Threat Model

We consider an honest-but-curious adversary that exactly follows the storage protocol, but attempts to learn the original content of the data and metadata in storage. Specifically, the adversary may take the following actions.

- It can compromise the server and access the fingerprint index, file recipes, key recipes and physical copies of the chunks that are kept by the server. It aims to infer the original information of *any* data or metadata by observing the server storage.
- In addition to the server, it can compromise some clients and further access the original data or metadata of the compromised clients. It aims to infer the original information of *unauthorized* data or metadata that belong to other non-compromised clients and are not permitted for access by the compromised clients.

We ensure that our Metadedup design is *compatible* with existing countermeasures [6], [17], [20], [25] that address different threats against encrypted deduplication systems. For example, a malicious client may abuse client-side deduplication to learn whether other users have already stored certain files [17], [18], and Metadedup can defeat the side-channel leakage by adopting the server-side deduplication on cross-user data [25] (see Section 6). A malicious server may modify or even delete stored files to destroy the availability of outsourced files, and Metadedup is compatible with the availability countermeasure that disperses data across servers via deduplication-aware secret sharing [25] (see Section 5).

We do not consider the threats that exploit the leakage of access patterns [19], although Metadedup can work in conjunction with the related countermeasures [37]. Metadedup can also be deployed with a private server to tolerate Byzantine faults [13], or with data auditing protocols [6], [20] to efficiently check the integrity of outsourced files against malicious corruptions.

4.2 Metadata Deduplication

Metadedup builds on indirection to preserve storage efficiency, while minimizing the index overhead. Recall that before deduplication, we first partition file data into chunks, which we now call *data chunks*. After the data chunks are encrypted by MLE (i.e., historical or server-aided MLE), Metadedup collects the metadata of multiple regions of adjacent encrypted data chunks into *metadata chunks* (each of which corresponds to a region of encrypted data chunks). Both file recipes and key recipes now store the information of metadata chunks, which reference the physical copies of data chunks in encrypted deduplication storage. Metadedup further applies deduplication to metadata chunks as well. Our observation is that identical data chunks across backups tend to be clustered together, and form regions of duplicates [21]. Thus, we can keep only one copy of metadata chunks for such repeated regions of data, thereby mitigating metadata storage overhead. Since Metadedup operates on metadata chunks, which have significantly larger sizes than fingerprints, it introduces limited storage overhead (e.g., 0.33-1.94%; see Experiment C.1 in Section 2 of the digital supplementary file) to the fingerprint index. In the following, we describe the design decisions in Metadedup.

Segmentation. Metadedup works after the encryption procedure on the client side and collects metadata information of the encrypted data chunks to be stored. It partitions the stream of encrypted data chunks into multiple coarse-grained data units called *segments*. A simple partitioning algorithm is *fixed-size segmentation*, which fixes a segment size and puts a

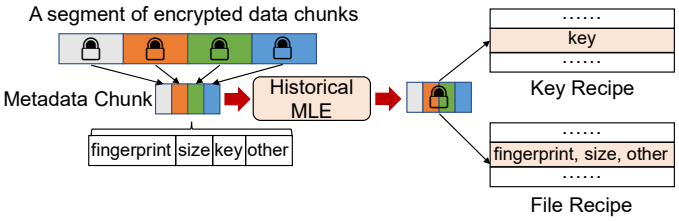


Fig. 3: Overview of metadata management in Metadedup.

segment boundary on every offset that is equal to a multiple of the segment size. Fixed-size segmentation is fast, but is vulnerable to the *boundary-shift problem* [16]. Since Metadedup deduplicates the metadata of segments, the boundary-shift problem can lead to many distinct segments and degrades the effectiveness of metadata deduplication.

Thus, Metadedup adopts *variable-size segmentation* [26], [33] to achieve high effectiveness for metadata deduplication. Variable-size segmentation works on the fingerprints of the encrypted data chunks, and configures the minimum, average, and maximum segment sizes, where the average segment size indicates a pre-defined divisor for segmentation. It sequentially traverses each data chunk, and identifies a segment boundary after a data chunk if (i) the size of the new segment is larger than the minimum segment size, and (ii) the fingerprint modulo the pre-defined divisor is equal to a fixed constant (e.g., 1) or the inclusion of the encrypted data chunk makes the size of the new segment larger than the maximum segment size. By default, we fix the minimum and maximum segment sizes as half and double of the average segment size, respectively.

Metadata management. For each segment obtained from the segmentation algorithm, Metadedup creates a metadata chunk that keeps the fingerprint, size, key, and other necessary metadata information derived from each encrypted data chunk within the segment (see Figure 3). This enables us to retrieve and decrypt a segment of data chunks based on a metadata chunk.

To protect metadata chunks, Metadedup adopts historical MLE (see Section 2.1) for both confidentiality and deduplication capabilities. Specifically, it computes the cryptographic hash of each metadata chunk as a key, and uses the hash key to encrypt this metadata chunk. The design decision is driven from both performance and security perspectives. From the performance perspective, historical MLE avoids the interaction with the key manager in server-aided MLE (see Section 2.1). From the security perspective, we argue that the protection by historical MLE is sufficient for metadata chunks, because it is much more computationally expensive to launch the offline brute-force attack against metadata chunks than against data chunks (see Section 1 of the digital supplementary file).

Given the encrypted metadata chunks, Metadedup creates both file and key recipes. Each entry of the file recipe keeps the fingerprint, size and other metadata information of an encrypted metadata chunk, while each entry of the key recipe keeps the corresponding key. It further encrypts the key recipe with the file owner's master key for protection.

Metadedup applies deduplication to both encrypted data and metadata chunks. Note that it does not further compress the data and metadata chunks after deduplication, since they are encrypted and less likely to benefit from compression. Also, it does not further apply deduplication to file recipes and key recipes. Both file recipes and key recipes are now used to reference metadata chunks, and their redundancies are relatively low. The benefits of eliminating the recipe redundancies may be negated by the additional deduplication metadata.

4.3 Basic Operations

We show how we incorporate variable-size segmentation and metadata management into basic operations. We first summarize the major notations used in the presentation of Metadedup. Suppose that a client uses its individual master key key to write and restore a target file. We denote the file recipe and the key recipe of the target file as $fRecipe$ and $kRecipe$, respectively. We also denote a data chunk and a metadata chunk by $dChunk$ and $mChunk$, as well as corresponding MLE keys as $dkey$ and $mkey$, respectively. We use $[X]_Y$ to denote the encryption output of an object X (that can be $kRecipe$, $dChunk$ or $mChunk$) encrypted with a key Y (that can be key, $dkey$ or $mkey$) using symmetric-key encryption (e.g., AES).

Algorithm 1 shows the interaction between a client and a server when writing a target file into storage. The client first divides the target file into data chunks and encrypts each data chunk (Lines 2-5). It creates segments based on encrypted data chunks (Line 6), collects the metadata in each segment into a metadata chunk (Lines 9-10), and encrypts the metadata chunk using historical MLE (Lines 11-12). It adds the deduplication metadata and key metadata of the metadata chunk into the file and key recipes, respectively (Lines 13-14). It further encrypts the key recipe with its master key (Line 15), and uploads the following information to the server (Line 16): (i) the file recipe and encrypted key recipe, (ii) the encrypted data chunks, and (iii) the encrypted metadata chunks.

The server performs deduplication on received (encrypted) data and metadata chunks, and stores the unique ones (Lines 18-19). It also stores the file recipe and encrypted key recipe (Line 20).

Algorithm 2 shows the two-round interactions for restoring a target file. In the first round, the client requests the metadata of the file based on its full pathname (Line 1); the server retrieves and sends the file recipe, encrypted key recipe and encrypted metadata chunks (Lines 2-5). In the second round, the client decrypts the key recipe and metadata chunks (Lines 7-10), and requests file data (Line 11); the server retrieves and sends the encrypted data chunks back to the client (Lines 12-14). The client decrypts each data chunk based on the corresponding key in metadata chunks (Lines 16-18), and finally assembles the data chunks to reconstruct the original file (Line 19).

5 Distributed Key Management

One limitation of Metadedup is that it does not ensure the fault tolerance of data chunks since the server is a single-point-of-failure. To this end, we adopt a quorum-based design [25], and extend Metadedup to incorporate multiple

Algorithm 1 Write operation of Metadedup

Client input: target file $file$, client's master key key
Initialize file and key recipes: $fRecipe, kRecipe$
Divide $file$ into data chunks $\{dChunk\}$
for each $dChunk$ **do**
  $dkey \leftarrow$ hash of $dChunk$
  $[dChunk]_{dkey} \leftarrow$ encryption of $dChunk$ with $dkey$
Divide $\{[dChunk]_{dkey}\}$ into segments $\{Seg\}$
for each Seg **do**
  Initialize metadata chunk $mChunk$
  for each $[dChunk]_{dkey}$ in Seg **do**
    Add metadata of $[dChunk]_{dkey}$ into $mChunk$
  $mkey \leftarrow$ cryptographic hash of $mChunk$
  $[mChunk]_{mkey} \leftarrow$ encryption of $mChunk$ with $mkey$
  Add deduplication metadata of $[mChunk]_{mkey}$ into $fRecipe$
  Add $mkey$ into $kRecipe$
$[kRecipe]_{key} \leftarrow$ encryption of $kRecipe$ with key
Upload: $fRecipe, [kRecipe]_{key}, \{[dChunk]_{dkey}\}, \{[mChunk]_{mkey}\}$

Server input: fingerprint index
Receive: $fRecipe, [kRecipe]_{key}, \{[dChunk]_{dkey}\}, \{[mChunk]_{mkey}\}$
Deduplicate $\{[dChunk]_{dkey}\}$ and $\{[mChunk]_{mkey}\}$
Store unique $\{[dChunk]_{dkey}\}$ and $\{[mChunk]_{mkey}\}$
Store $fRecipe$ and $[kRecipe]_{key}$

Algorithm 2 Restore operation of Metadedup

Client input: full pathname $name$ of the target file
Request metadata based on $name$

Server input: $fRecipe, \{[mChunk]_{mkey}\}$ and $[kRecipe]_{key}$ of each stored file
Receive $name$
Retrieve the corresponding $fRecipe$ and $[kRecipe]_{key}$ based on $name$
Retrieve $\{[mChunk]_{mkey}\}$ based on $fRecipe$
Send $fRecipe, \{[mChunk]_{mkey}\}$ and $[kRecipe]_{key}$

Client input: client's master key key
Receive $fRecipe, \{[mChunk]_{mkey}\}$ and $[kRecipe]_{key}$
$kRecipe \leftarrow$ decryption of $[kRecipe]_{key}$ with key
for each $[mChunk]_{mkey}$ **do**
  $mkey \leftarrow$ corresponding key in $kRecipe$
  $mChunk \leftarrow$ decryption of $[mChunk]_{mkey}$ with $mkey$
Request data chunks using deduplication metadata in $\{mChunk\}$

Server input: $\{[dChunk]_{dkey}\}$
Receive deduplication metadata of data chunks
Retrieve $\{[dChunk]_{dkey}\}$ based on deduplication metadata
Send $\{[dChunk]_{dkey}\}$

Client input: $\{mChunk\}$
Receive $\{[dChunk]_{dkey}\}$
for each $[dChunk]_{dkey}$ **do**
  Retrieve corresponding $dkey$ in $\{mChunk\}$
  $dChunk \leftarrow$ decryption of $[dChunk]_{dkey}$ with $dkey$
Assemble $\{dChunk\}$ to original file

servers. Specifically, we treat each data chunk as a secret, and encode it on the client side into s shares that are stored by s distinct servers using a (s, t)-*deduplication-aware secret sharing algorithm* (where $s \geq t > 0$), in which the random factor in traditional secret sharing is replaced by the cryptographic hash of the input secret. This enables three properties: (i) *reliability*, i.e., the data chunk can be correctly rebuilt if any t shares are available; (ii) *security*, i.e., the data chunk remains confidential if no more than $t - 1$ shares are compromised; and (iii) *deduplication-aware*, i.e., identical data chunks are encoded into identical shares that can be deduplicated on the server side.

However, the deduplication-aware secret sharing algorithm [25] relies on the assumption that each secret is *unpredictable* (see Section 2.1); otherwise, it is susceptible to the offline brute-force attack as in historical MLE. In other words, an adversary that compromises a share can learn the original data chunk, by finding the data chunk that is encoded into the share (i.e., like the offline brute-force attack in Section 2.1).

In this section, we argue that existing key management approaches have limitations in addressing the unpredictability assumption in the multi-server architecture. Thus, we propose a distributed key management approach to ensure data robustness against adversarial compromise, while preserving secure and space-efficient metadata management.

5.1 Limitations of Existing Approaches

To address the unpredictability assumption, a simple way is to extend the server-aided approach [7], in which we introduce a global secret and generate the MLE key of each data chunk based on the global secret in addition to the chunk hash (see Section 2.1). Instead of using the MLE key directly for encryption, we feed the MLE key and the corresponding data chunk as input to the deduplication-aware secret sharing algorithm [25], which returns the shares that are in essence protected by the global secret. If the global secret remains confidential, we can provide security guarantees for all data chunks without relying on the unpredictability assumption.

However, the above server-aided approach has two limitations. First, the global secret is a single-point-of-failure, in that the security level of all data chunks is inevitably degraded if it is compromised. Second, the implementation of the server-aided approach [7] protects the global secret in a dedicated key manager, yet the key manager easily becomes a performance bottleneck as the number of clients scales up. Although we can replicate multiple key managers for load balance, this introduces additional risk for the leakage of the global secret.

Building on the server-aided approach, Duan [15] proposes to distribute the global secret across multiple key managers. Specifically, each key manager keeps a share of the global secret, and returns a key share (based on its secret share) upon key generation request. After collecting a threshold number of key shares, a client can rebuild the MLE key of a data chunk. This approach preserves security even if a number of key managers are compromised. It also achieves load balance, since the client can choose some unoccupied key managers for key generation. However, the distributed approach [15] builds on an expensive cryptographic primitive (i.e., threshold signature) that is theoretically proven but is not yet readily implemented in practice.

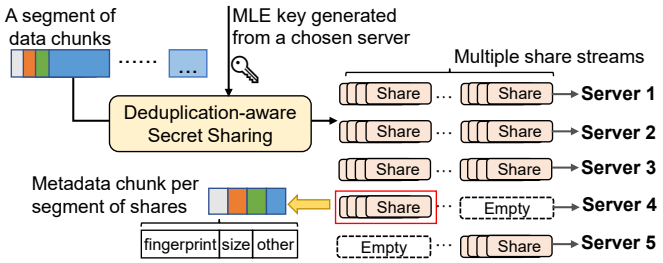


Fig. 4: Overview of our distributed key management approach. Here, we deploy five servers under (4, 3)-deduplication-aware secret sharing. For example, for the input segment, we choose the fifth server (based on the minimum fingerprint) to perform key generation and upload the shares to the first four servers.

5.2 Our Proposed Key Management Approach

We propose a new distributed key management approach in a multi-server architecture. To be compatible with (s, t)-deduplication-aware secret sharing, we deploy $s + 1$ servers, so as to generate the MLE key of each data chunk from one server (using the server-aided MLE approach [7]; see Section 2.1) and distribute its resulting shares to the remaining s servers (i.e., except the one that generates the MLE key). Our rationale is to preserve security of the data chunks against *any* single compromised server. The reason is that an adversary cannot learn the original information based on either a global secret or a single share of a data chunk from the compromised server.

In addition, instead of arranging a dedicated server to generate the MLE keys for all data chunks (e.g., the simple approach in Section 5.1), we allow any server to perform key generation. Specifically, we maintain an independent global secret in each server (i.e., there are $s + 1$ global secrets in total). For each data chunk, we leverage the chunk content to choose the server to generate its MLE key. This ensures that if more servers are compromised, the security level of the overall system only gracefully degrades (see Theorem 2 in Section 5.3), since each server just generates the MLE keys of a subset of all data chunks.

In the following, we elaborate the design decisions of implementing our key management approach (see Figure 4) in Metadedup.

Key generation. To generate MLE keys, one approach is to run an oblivious pseudorandom function (OPRF) protocol [7] on a per-chunk basis, such that the MLE keys can be successfully generated without leaking the underlying chunk fingerprints or the global secret. However, the OPRF protocol incurs substantial performance overhead when there is a huge number of data chunks [33], [41].

To mitigate the key generation performance overhead, Metadedup uses the similarity-based approach [33] to generate coarse-grained MLE keys. Specifically, it performs variable-size segmentation on the original data chunks (as opposed to our basic design that applies segmentation to encrypted data chunks; see Section 4.2). It chooses the data chunk that has the *minimum fingerprint* in each segment, as well as the server identified by the value of the minimum fingerprint modulo $s + 1$. Then it runs the OPRF protocol [7] with the chosen server, so as to generate a per-segment MLE key based on the minimum fingerprint.

Our design choice only slightly degrades the deduplication effectiveness of data chunks. The reason is that similar segments share a large fraction of identical data chunks. They are likely to have the same minimum fingerprint [9], [12] and hence the same MLE key (recall that an identical minimum fingerprint is always directed to the same server for key generation). More importantly, our design does not affect the deduplication effectiveness of metadata chunks, since Metadedup creates metadata chunks on a per-segment basis and identical segments still lead to identical metadata chunks (see details below).

Metadata chunk organization. For each segment, Metadedup encodes its data chunks by the corresponding MLE key (i.e., as in the simple approach in Section 5.1) and generates s share streams, which are then written to the s servers except the one that is applied in key generation. Since we distribute the key generation workload across all $s + 1$ servers, each of the $s + 1$ servers will receive a share stream if it is not selected for key generation. Thus, we have $s + 1$ share streams in total for all segments, while the MLE key generation is also distributed across all $s + 1$ servers.

For each of the $s + 1$ share streams, Metadedup creates metadata chunks. Each metadata chunk keeps the fingerprints, sizes, and other necessary metadata information of the shares from a segment. We do not include MLE keys in metadata chunks, since they are not necessary for the reconstruction of original data chunks [25]. Also, we do not need to encrypt metadata chunks (as opposed to our basic design in Section 4.2), since the metadata chunks now only include deduplication metadata for shares. This information is not helpful for inferring original data chunks. For each of the $s+1$ streams of metadata chunks, Metadedup creates a file recipe that keeps the fingerprints, sizes, sequence numbers, and the corresponding share indexes of metadata chunks. Then it distributes the shares, metadata chunks, and the file recipe in each stream to the corresponding server.

5.3 Robustness Analysis

We analyze the robustness of Metadedup from both reliability and security perspectives. We first show that Metadedup maintains the fault tolerance capability of the underlying deduplication-aware secret sharing algorithm.

Theorem 1. *Suppose that* Metadedup *deploys $s + 1$ servers to work with the (s, t)-deduplication-aware secret sharing algorithm. If at most $s - t$ servers fail, it can recover all data chunks from the remaining $t + 1$ available servers.*

Proof. We prove the theorem by contradiction. Suppose the contrary that at least one data chunk M cannot be recovered from the $t+1$ available servers. We denote the set of available servers by S_a, and the set of servers that store any share of M by S_r. Since M cannot be recovered, the servers in S_a do not have the sufficient number (i.e., t) of shares of M. This implies that $|S_a \cap S_r| < t$.

In addition, we have the number of servers in S_a as $|S_a| = t + 1$, as well as $|S_r| = s$ due to the (s, t)-secret-sharing of M. Thus, we have $|S_a \cup S_r| = |S_a| + |S_r| - |S_a \cap S_r| > t + 1 + s - t = s + 1$. This contradicts the hypothesis that the total number of severs is $s + 1$. □

We now consider the security of Metadedup under distributed key management. Our goal is to show that even an adversary compromises more servers, the security level of Metadedup does not vanish completely; instead, it still preserves the security of a large fraction of data chunks.

Theorem 2. *Suppose that* Metadedup *deploys $s+1$ servers to work with the (s,t)-deduplication-aware secret sharing algorithm, and each server generates the MLE keys of $\frac{1}{s+1}$ data chunks and stores the shares of the remaining $1-\frac{1}{s+1}$ data chunks. Suppose now that an adversary compromises any k out of the $s+1$ servers, where $k \leq t$.* Metadedup *provides security guarantees under three cases:*

- *If $k=1$, it ensures the security for all data chunks.*
- *If $1<k<t$, it ensures the security for $1-\frac{k}{s+1}$ data chunks. It also protects the remaining $\frac{k}{s+1}$ data chunks if they are unpredictable.*
- *If $k=t$, it protects $\frac{t}{s+1}$ data chunks, if they are unpredictable.*

Proof. When $k=1$, the adversary can access shares and global secret from a single compromised server. Since Metadedup separates the management of the shares and corresponding global secrets in different servers, the compromised shares and the global secret in one server are unrelated. The adversary cannot use either of them to infer the original information.

When $1<k<t$, the adversary can access k global secrets that are used to generate the MLE keys of $\frac{k}{1+s}$ data chunks. Since the global secrets corresponding to $1-\frac{k}{1+s}$ data chunks remain confidential, Metadedup ensures the security for these data chunks. In addition, for each of the remaining $\frac{k}{1+s}$ data chunks, the adversary can access the corresponding global secret and at most $k-1$ shares of each of the data chunks. Metadedup protects these data chunks if they are unpredictable, since the adversary does not have the sufficient number of shares to recover any original contents.

When $k=t$, the adversary can access at most $t-1$ shares of the data chunks, whose MLE keys are generated from the compromised servers. In other words, for each of the $\frac{t}{s+1}$ data chunks, the adversary only learns its $t-1$ shares and the corresponding global secret. Thus, Metadedup ensures the security, if these data chunks are unpredictable. □

6 Implementation

We implement a Metadedup prototype in C++ based on our previously built system CDStore [25], a multi-cloud storage system that supports encrypted deduplication via deduplication-aware secret sharing (see Section 5). CDStore also applies client-side deduplication on the data from the same client, followed by server-side deduplication on that from all clients, so as to be robust against side-channel leakage. On the other hand, CDStore does not address metadata management in the multi-cloud architecture; while the goal of Metadedup is to augment CDStore with new approaches to improve its security and storage efficiency.

We follow the modular approach as in CDStore to implement Metadedup. Figure 5 shows how Metadedup adds new modules to CDStore. We use OpenSSL 1.1.1 [3] to implement the cryptographic operations in Metadedup. Our current Metadedup prototype adds about 7,400 lines of code to the original CDStore prototype.

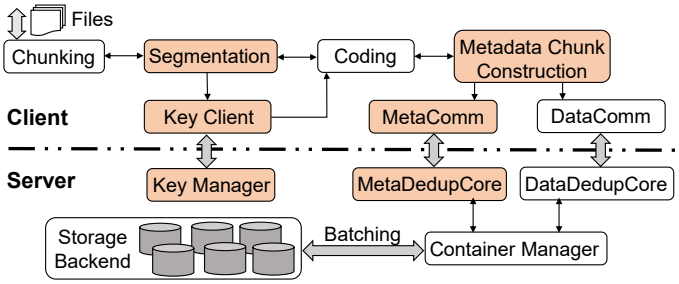


Fig. 5: Implementation of the Metadedup prototype (the modules that are newly added to CDStore are colored in orange).

6.1 Modules

We elaborate how Metadedup augments CDStore to realize the write and restore operations with metadata deduplication and distributed key management.

Write. As in CDStore, a client writes a file by partitioning it into data secrets via the *chunking* module, which implements Rabin fingerprinting [34] for variable-size chunking. Rabin fingerprinting takes the minimum, average and maximum sizes of chunks as inputs, which we now fix as 2 KB, 8 KB and 16 KB, respectively. It groups chunks into variable-size segments (see Section 4.2) via the *segmentation* module.

In the *key client* and *key manager* modules, Metadedup implements the RSA-based OPRF [7], which uses RSA blind signature to protect the original fingerprints against each server. It configures each key manager (that is deployed in each server) with a 1024-bit RSA public/private key pair. In key generation, (i) the key client sends the blinded minimum chunk fingerprint of each segment to corresponding key manager; (ii) the key manager computes an RSA signature on the blinded fingerprint; (iii) the key client unblinds the result to form the segment-level MLE key.

In the *coding* module, Metadedup encodes each data chunk into s shares, and organizes the shares of the chunks from different segments into $s+1$ share streams.

Metadedup introduces a new *metadata chunk construction* module. In each of the $s+1$ streams, it constructs a metadata chunk based on the shares, whose original data chunks are in the same segment. It also prepares a file recipe to keep the metadata of the metadata chunks that correspond to the same stream (i.e., we have $s+1$ file recipes in total).

Metadedup adds a *MetaComm* module for the communication of metadata chunks and file recipes with servers, in addition to the original *DataComm* module for data communication in CDStore. Specifically, in MetaComm, it performs intra-user deduplication on metadata chunks, and only sends (i) unique metadata chunks, and the (ii) file recipe to corresponding server. To mitigate the network transmission overhead, we batch the uploaded content in an in-memory buffer of size 4 MB, and upload the buffered content when the buffer is full.

On the server side, when a server receives the shares and metadata chunks from a client, it applies deduplication to them in the *DataDedupCore* and *MetaDedupCore* modules, respectively. Each module maintains an independent fingerprint index implemented based on the key-value store levelDB [2], which maps a fingerprint to an ID of a container

(see below) that stores the corresponding data share or metadata chunk.

In the *container manager*, the server writes the unique content, as well as the file recipe, in the units of containers. Each container is now configured with a fixed size of 4 MB. This mitigates the disk access overhead due to the frequent accesses to the data shares or metadata chunks that have smaller sizes of several kilobytes (e.g., 8 KB).

Restore. To restore a file, a client connects to any $t+1$ out of $s+1$ servers to request for the shares of the file (recall that Metadedup allows to recover all data chunks from any $t+1$ servers, see Theorem 1 in Section 5.3). Each server returns the file recipe, metadata chunks, and shares. The client extracts the deduplication metadata of shares, and recovers each data chunk from the corresponding t shares. Finally, it assembles the recovered data chunks to the original file.

6.2 Discussion

We discuss several implementation details in our prototype.

Parallelization. We follow CDStore to parallelize the operations of Metadedup through multi-threading. We first parallelize the processing of different modules, and transfer the outputs of the paralleled modules via a high-performance lock-free message queue [1]. In addition, we apply multi-threading to the encoding and decoding operations for the secret sharing algorithm (see [25] for details).

Filename protection. Our current implementation uses the full pathname of a file to write and restore the corresponding file data. We can use an obfuscated name (e.g., encoded by a salted hash function) to access the file.

7 Evaluation

7.1 Performance

Experiment A.1 (Microbenchmarks). We first implement the *metadata workflow* of our metadata deduplication algorithms (see Section 4.3) and study the computational performance of each processing step. Here, we do not consider the client-server communication as in our prototype evaluation (which will be addressed in Experiment A.2).

We create 10 GB of unique data without any content redundancy, which enables us to perform stress-tests with the maximum amount of metadata. We generate the corresponding data chunks and their metadata that are to be processed by the metadata deduplication algorithms. We conduct microbenchmarks on a local machine equipped with 2.40 GHz Intel(R) Xeon(R) E5-2620 v3 and 32 GB memory.

We measure the time consumed in each step of data write and restore procedures. Specifically, the write procedure includes: (i) *segmentation*, which groups data chunks into segments; (ii) *metadata handling*, which creates metadata chunks and file recipes; (iii) *recipe handling*, which collects both file and key recipes and further encrypts the key recipe. The restore procedure includes: (i) *recipe restore*, which reconstructs both the file recipe and the key recipe; and (ii) *metadata restore*, which recovers all metadata chunks.

Table 1 presents the evaluation results averaged over 10 runs, including the 90% confidence intervals for the throughput results. We observe that the most time-consuming step in the write procedure is metadata handling, which takes 42.65-78.69% of the overall time. In addition, the write throughput generally increases with the average segment size, since the number of metadata chunks to be handled is reduced with fewer segments. For example, when the average segment size is at least 1 MB, the write throughput is capable to achieve above 9 GB/s.

TABLE 1: Metadata deduplication microbenchmarks of metadata flow for the average segment size from 512 KB to 4 MB. Note that the write and restore throughput results are computed based on original data size (i.e., 10 GB).

Procedures/Steps		512KB	1MB	2MB	4MB
Write	Segmentation	0.394s	0.391s	0.395s	0.404s
	Metadata handling	3.084s	0.632s	0.611s	0.627s
	Recipes handling	0.441s	0.427s	0.425s	0.439s
	Throughput (GB/s)	**4.84±0.12**	**9.45±0.01**	**9.65±0.00**	**9.39±0.01**
Restore	Recipes restore	0.005s	0.003s	0.001s	0.001s
	Metadata restore	5.085s	2.664s	1.437s	0.858s
	Throughput (GB/s)	**1.96±0.00**	**3.75±0.00**	**6.95±0.00**	**11.65±0.01**

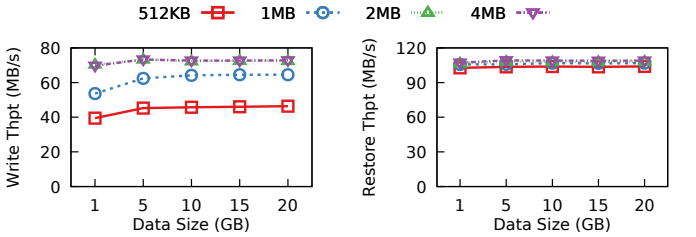


Fig. 6: Write throughput and restore throughput of Metadedup under different average segment sizes. We omit the performance variances across different runs since they are very small (e.g., within 0.07 MB/s).

In the restore procedure, the performance bottleneck is metadata restore, which takes more than 99% of the total time. When the average segment size is 4 MB, the restore throughput achieves 11.65 GB/s.

Experiment A.2 (Prototype performance). We now study the performance of the Metadedup prototype (that enables both metadata deduplication and distributed key management) in a networked setting. Our evaluation setting is as follows. We deploy a client instance on a machine that has a six-core 2.40GHz Intel(R) Xeon(R) CPU E5-2620 v3 and 32 GB RAM, and five server instances on a different machine that has a 10-core 2.40GHz Intel(R) Xeon(R) CPU E5-2640 v4 and 32 GB RAM. We distinguish different server instances in the same machine by distinct ports. Both client and server machines are connected via a 1 Gb/s switch. We configure the coder module (of Metadedup) with two threads to boost performance. We do not consider more threads, because our results (see Figure 6) suggest that two encoding threads are sufficient for achieving the required performance.

We vary the size of test data, and evaluate Metadedup under different average segment sizes. Specifically, we create and upload a certain size of unique data to five servers, and then download them. We evaluate the throughput of both write and restore operations, and present the average results over 10 runs.

Figure 6 shows the performance results. In the write operation, a larger average segment size leads to higher performance of Metadedup. The reason is that it generates a fewer number of MLE keys. When the average segment size

increases to 4 MB, the write throughput reaches 73.33 MB/s for processing 5 GB of unique data. To examine the performance overhead, we find that the effective network speed of our testbed is about 112.5 MB/s. Since (s, t)-deduplication-aware secret sharing adds slightly more than s/t times redundancies of the original data chunks [25], the upper bound of the effective transfer speed of the shares is $112.5 \times t/s$ = 84.38 MB/s, where $s = 4$ is the number of shares encoded by a data chunk and $t = 3$ is the number of necessary shares for the reconstruction of a data chunk. Thus, the write operation incurs about 13.09% throughput loss to the maximum transfer speed of the shares.

In the restore operation, the performance of Metadedup (slightly) increases with the average segment size, because the number of metadata chunks to be restored decreases. When the average segment size is 4 MB, Metadedup achieves 109.06 MB/s for restoring 10 GB of unique data, only 3.06% less than our effective network speed (Metadedup only needs to retrieve t shares for restoring each data chunk). Note that the overall restore performance significantly outperforms the restore performance that we previously reported [23], which achieves only about 50 MB/s when the average segment size is 512 KB. The reason is that we now do not need to encrypt metadata chunks (see Section 5.2), and hence avoid the serial retrieval of metadata and data.

7.2 Storage Efficiency

We evaluate the storage efficiency of overall Metadedup that enables metadata deduplication (see Section 4.2) and distributed key management (see Section 5.2). We assume that Metadedup stores data in five servers via (4, 3)-deduplication-aware secret sharing. We use two real-world datasets.

- **FSL:** This is a public dataset collected by the File systems and Storage Lab (FSL) at Stony Brook University [4], [38]. We focus on the `fslhomes`, which contains the snapshots of students' home directories from a shared network file system. Each FSL snapshot is represented by 48-bit fingerprints of variable-size chunks, as well as corresponding metadata information. We pick all snapshots from January 22 to June 17, 2013, aggregate them in a daily basis and obtain 115 daily backups (that are not continuous). The dataset takes 56.20 TB of data before deduplication.
- **VM:** This is our private dataset collected by ourselves and is also used in the evaluation of our previous work [25], [33]. It consists of the virtual machine (VM) image snapshots that capture the three-month programming activities of students enrolling in a university programming course. It includes 156 VM image snapshots, each of which is of 10 GB and represented by the SHA-1 fingerprints of 4 KB fixed-size chunks. We aggregate all snapshots on a daily basis, and obtain 26 full daily backups for the VM images. The dataset contains 39.61 TB of data before deduplication.

We build a trace-driven simulator based on the metadata size settings in Section 2.2 and the chunk data sizes in each trace. The simulator adds the FSL or VM backups to storage in the order of their creation times, and computes three metrics: (i) *data storage saving*, the percentage of the total data size reduced by data deduplication; (ii) *metadata storage saving*, the percentage of the total metadata size (excluding the fingerprint index) reduced by metadata deduplication; and (iii) *index overhead*, the percentage of additional storage cost to the fingerprint index.

In addition to FSL and VM backups, we also include a new real-world dataset from Microsoft [31] for our evaluation (see Section 2 of the digital supplementary file). We show that the metadata deduplication approach (see Section 4.2) achieves at least 67% of metadata storage savings under file system snapshots.

Experiment B.1 (Overall storage efficiency). Table 2 shows the simulation results of data and metadata storage savings and index overhead after storing all FSL and VM backups, where *raw* denotes the resulting data and metadata size after applying exact data deduplication only (i.e., without Metadedup). Metadedup degrades the data storage savings by 0.3-0.8% from exact deduplication. The reasons are two-fold. First, it introduces redundancies (via secret sharing) for the goal of fault-tolerant data storage. Second, it only achieves near-exact deduplication for data chunks for performance consideration (i.e., using similarity-based key generation [33]).

Metadedup generally preserves high metadata storage saving such as about 90% in the FSL dataset and 88-93% in the VM dataset. Specifically, in addition to metadata deduplication, Metadedup does not need to maintain key recipes, since it protects data chunks via secret sharing [25] that is a keyless security algorithm; on the other hand, Metadedup amplifies metadata storage due to the distributed storage across multiple servers.

Metadedup adds high storage overhead to the fingerprint index. For example, when we vary the average segment size from 512 KB to 4 MB, the index overheads increase from 478% to 537% and from 481% to 580% in the FSL and VM datasets, respectively. The reason is that a larger average segment size introduces more non-duplicate shares, thereby increasing the storage overhead of the fingerprint index (see below).

We argue that the index overhead is *acceptable* in practice, since index storage only takes a small fraction (e.g., less than 0.5%, see Section 2.2) in overall metadata storage. For example, without Metadedup, we only need up to 1.39 GB of index storage to process tens of TB backup workloads (e.g., FSL and VM, see Section 7.2); even Metadedup amplifies the index storage by about five times, it takes up to 8.86 GB. More importantly, the index storage is shared by multiple servers, and each server only spends about 1.77 GB of index storage on average; this can be fit for most commodity machines.

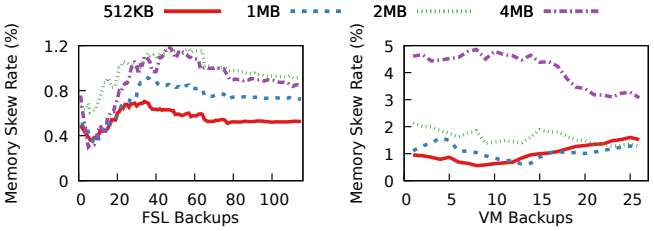
In addition, the index overhead is contributed by (i) the non-duplicate data shares introduced by near-exact deduplication and (ii) the unique metadata chunks in metadata deduplication. Our metadata deduplication approach only introduces *negligible* part (e.g., less than 1.94% in a single server; see Experiment C.1 in Section 2 of the digital supplementary file) of the index overhead.

Experiment B.2 (Load balance analysis). We analyze the storage load distribution across servers. Our goal is to show that while not explicitly address this issue, Metadedup can balance the storage load across all servers. We assume that each server stores file recipes, metadata chunks and data chunks in disk for long-term backup, while keeps fingerprint index in memory for high deduplication performance [42]. To measure load balance, we define *memory (disk) skew rate* as

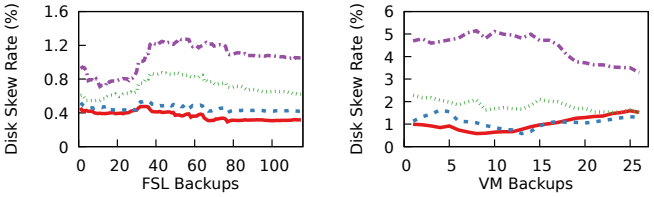
TABLE 2: Overall storage efficiency of Metadedup in FSL and VM datasets. *Raw* denotes the resulting data and metadata size after applying exact data deduplication only (i.e., without Metadedup). For Metadedup, we consider the average segment size that ranges from 512 KB to 4 MB.

	Components/Metrics	Raw	512KB	1MB	2MB	4MB
	Total logical data size (GB)	57548.3				
FSL	**Total unique data size (GB)**	**431.90**	**750.28**	**779.13**	**810.80**	**846.21**
FSL	**Data Storage saving**	**99.25%**	**98.70%**	**98.65%**	**98.59%**	**98.53%**
FSL	File recipes (GB)	178.19	7.73	3.86	1.92	0.86
FSL	Key recipes (GB)	190.07	–	–	–	–
FSL	Metadata chunks (GB)	–	27.22	29.48	32.16	35.25
FSL	Fingerprint index (GB)	1.39	8.03	8.26	8.54	8.86
FSL	**Total metadata size (GB)**	**369.65**	**42.98**	**41.60**	**42.62**	**44.97**
FSL	**Metadata storage saving**	**–**	**90.55%**	**90.98%**	**90.78%**	**90.23%**
FSL	**Index overhead**	**–**	**478%**	**494%**	**514%**	**537%**
	Total logical data size (GB)	40560.0				
VM	**Total data size (GB)**	**168.24**	**321.29**	**332.04**	**352.14**	**380.14**
VM	**Data storage saving**	**99.59%**	**99.21%**	**99.18%**	**99.13%**	**99.06%**
VM	File recipes (GB)	297.07	4.77	2.38	1.19	0.59
VM	Key recipes (GB)	316.88	–	–	–	–
VM	Metadata chunks (GB)	–	32.42	39.17	49.10	67.30
VM	Fingerprint index (GB)	1.23	7.15	7.35	7.77	8.37
VM	**Total metadata size (GB)**	**615.18**	**44.34**	**48.90**	**58.06**	**76.26**
VM	**Metadata storage saving**	**–**	**93.94%**	**93.23%**	**91.81%**	**88.94%**
VM	**Index overhead**	**–**	**481%**	**498%**	**532%**	**580%**

Note: (i) The results of Metadedup are *aggregated* from all five servers; (ii) The total metadata size excludes the size of fingerprint index.



(a) Memory skew rates in FSL and VM datasets. The ranges of the y-axis are from zero to 1.2% and 5% for the FSL and VM datasets, respectively.



(b) Disk skew rates in FSL and VM datasets. The ranges of the y-axis are from zero to 1.6% and 6% for the FSL and VM datasets, respectively.

Fig. 7: Load balance analysis of Metadedup under different average segment sizes.

the percentage of the difference between the maximum and minimum memory (disk) usage among all servers by their average memory (disk) usage. Clearly, the lower the skew rate is, the better load balance Metadedup achieves.

Figure 7 presents the memory and disk skew rates after storing each FSL/VM backup. A larger average segment size generally leads to a higher skew rate (for both memory and disk). The reason is that the minimum chunk fingerprints in large segments incur a skew distribution on the usage of the servers for key generation. In addition, compared to the FSL dataset, the VM dataset suffers from a (relatively) higher skew rate. The reason is that the VM dataset includes a large fraction of zero chunks, which naturally affects the skew rate. Nevertheless, Metadedup achieves a good storage load balance across servers. For example, after storing all FSL backups, its memory skew rate is below 0.91%, while disk skew rate is below 1.05%.

8 Conclusion

We present Metadedup, which exploits the power of indirection to realize deduplication to metadata. It significantly mitigates the metadata storage overhead in encrypted deduplication, while preserving confidentiality guarantees for both data and metadata. We extend Metadedup for secure and fault-tolerant storage in a multi-server architecture; in particular, we propose a distributed key management approach, which preserves security guarantees even a number of servers are compromised. We extensively evaluate Metadedup from performance and storage efficiency aspects. We show that Metadedup significantly suppresses the storage space of metadata, while incurring limited performance overhead compared to network speed.

Acknowledgments

This work was supported in part by grants by the National Key R&D Program of China (2017YFB0802300), the National Natural Science Foundation of China (61972073), the Key Research Funds of Sichuan Province (2020YFG0298, 2021YFG0167), Sichuan Science and Technology Program (2020JDTD0007), and Innovation and Technology Fund (ITS/315/18FX).

References

[1] Boost c++ libraries. https://www.boost.org/.
[2] Leveldb. https://github.com/google/leveldb.
[3] Openssl. https://www.openssl.org.
[4] FSL traces and snapshots public archive. http://tracer.filesystems.org/, 2014.
[5] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proc. of ACM CCS*, 2015.
[6] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS*, 2007.
[7] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.
[8] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EUROCRYPT*, 2013.
[9] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. of IEEE MASCOTS*, 2009.
[10] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX ATC*, 2006.
[11] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage*, 2(4):424–448, 2006.
[12] A. Z. Broder. On the resemblance and containment of documents. In *Proc. of SEQUENCES*, 1997.
[13] D. Dobre, P. Viotti, and M. Vukolič. Hybris: Robust hybrid cloud storage. In *Proc. of ACM SoCC*, 2014.
[14] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.
[15] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *Proc. of ACM CCSW*, 2014.
[16] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. Technical report, HPL-2005-30R1, 2005.

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

[18] 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.

[19] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proc. of NDSS*, 2012.

[20] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS*, 2007.

[21] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *Proc. of USENIX FAST*, 2010.

[22] J. Li, X. Chen, M. Li, J. Li, P. P. C. Lee, and W. Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE Transactions on Parallel Distributed Systems*, 25(6):1615–1625, 2014.

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

[24] J. Li, Z. Yang, Y. Ren, P. P. C. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proc. of Eurosys*, 2020.

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

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

[27] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace. Metadata considered harmful ... to deduplication. In *Proc. of USENIX HotStorage*, 2015.

[28] G. Lu, Y. Jin, and D. H. Du. Frequency based chunking for data de-duplication. In *Proc. of IEEE MASCOTS*, 2010.

[29] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok. Using hints to improve inline block-layer deduplication. In *Proc. of USENIX FAST*, 2016.

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

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

[32] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proc. of USENIX Security*, 2011.

[33] 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):9:1–9:30, 2017.

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

[35] B. Romański, L. Heldt, W. Kilian, K. Lichota, and C. Dubnicki. Anchor-driven subchunk deduplication. In *Proc. of ACM SYSTOR*, 2011.

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

[37] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious ram protocol. In *Proc. of ACM CCS*, 2013.

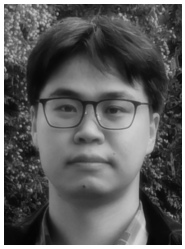
[38] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok. Cluster and single-node analysis of long-term deduplication patterns. *ACM Transactions on Storage*, 14(2):13:1–13:25, 2018.

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

[40] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

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

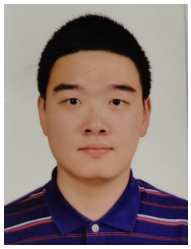
[42] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.



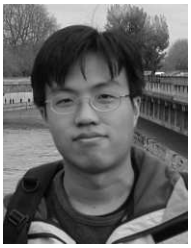
Jingwei Li is an associate professor in the University of Electronic Science and Technology of China. His research interest is to apply cryptographic technologies to build secure systems for large-scale data processing. His current focus is secure deduplication.

Suyu Huang is a master student in the University of Electronic Science and Technology of China. His research interest is to build encrypted deduplication storage systems.

Yanjing Ren is a master student in the University of Electronic Science and Technology of China. His research interest is to build encrypted deduplication storage systems.



Zuoru Yang received the B.Eng. degree in Computer Science from Xi'an Jiaotong University in 2018. He is currently a Ph.D. student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include data deduplication, security, etc.



Patrick P. C. Lee is an associate professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, operating systems, dependability, and security.

Xiaosong Zhang received his master and PhD degrees from University of Electronic Science and Technology of China, Chengdu, China, in 1999 andpdf 2011, respectively. Since 2011 he has been a professor in information security at University of Electronic Science and Technology of China. His research interest includes cryptography, dynamic program analysis and information security.

Yao Hao is a senior engineer of Science and Technology on Communication Security Laboratory, China and China Electronics Technology Cyber Security Co., Ltd. His research interests include applied cryptography, big data security and secure storage.