Metadedup: Deduplicating Metadata in Encrypted Deduplication via Indirection

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

***Abstract*—Encrypted deduplication combines encryption and deduplication in a seamless way to provide confidentiality guarantees for the physical data in deduplication 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. In addition, metadata chunks can be protected under the same encrypted deduplication framework, thereby providing confidentiality guarantees for metadata as well. We evaluate Metadedup through microbenchmarks, prototype experiments, and trace-driven simulation. Metadedup has limited computational overhead in metadata processing, and only adds 6.19% of performance overhead on average when storing files in a networked setting. Also, for real-world backup workloads, Metadedup saves the metadata storage by up to 97.46% at the expense of only up to 1.07% of indexing overhead for metadata chunks.**

I. Introduction

Chunk-based deduplication is widely used in modern primary [32] and backup [26], [42], [45] 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% [32] and that of backup storage by up to 98% [42]. This motivates the wide deployment of deduplication in various commercial cloud storage services (e.g., Dropbox, Google Drive, Bitcasa, Mozy, and Memopal) for maintenance cost savings [19].

To provide confidentiality guarantees, *encrypted deduplication* [9], [10] adds an encryption layer to deduplication, such that each chunk, before being written to deduplication 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 [16]). This ensures that duplicate chunks remain to have identical content even after encryption, and hence we can still apply deduplication to the encrypted chunks for storage savings. Many studies (e.g., [6], [9], [25], [34], [37]) have designed various encrypted deduplication schemes to efficiently manage outsourced data in cloud storage.

In addition to storing non-duplicate data, a deduplication 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 [13], [23], [31], 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.

A. 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 contain references to *data chunks* (i.e., the chunks of file data). If Metadedup stores nearly identical files regularly (e.g., periodic backups [42]), the corresponding file and key metadata are expected to have long sequences of references that are in identical orders. This implies that the metadata chunks are highly redundant and hence can be effectively deduplicated.
- We show how metadata chunks can also be protected by encrypted deduplication like the data chunks. We

show via security analysis that Metadedup preserves the confidentiality guarantees for both data and metadata chunks; in particular, we can apply a weak form of encryption [16] on metadata chunks to achieve high performance, yet we show that the metadata chunks remain highly robust against the offline brute-force attack.
- We evaluate Metadedup via microbenchmarks, and show that it incurs limited computational overhead in metadata processing.
- We implement a Metadedup prototype and evaluate its performance in a networked setup. Our prototype experiments show that Metadedup only adds small performance overhead when writing files to encrypted deduplication storage. For example, the average performance overhead of storing unique file data is 6.19% on average for different sizes ranging from 1 GB to 20 GB.
- Finally, we conduct trace-driven simulation on two real-world datasets. We show that Metadedup achieves up to 97.46% of metadata storage savings in encrypted deduplication, while incurring only up to 1.07% of indexing overhead for metadata chunks. Our savings of metadata storage are significantly higher than those of existing compression approaches [31].

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

B. Paper Organization

The rest of this paper is organized as follows. Section II motivates the need of addressing metadata storage overhead in encrypted deduplication via mathematical analysis and trace-driven simulation. Section III reviews related work. Section IV presents the design of Metadedup. Section V shows the implementation details of our Metadedup prototype. Section VI presents our evaluation results. Finally, Section VII concludes this paper.

II. Background and Motivation

We first introduce the background of encrypted deduplication. We then motivate our work by showing the high metadata storage overhead in encrypted deduplication storage systems via mathematical analysis and trace-driven simulation.

A. Encrypted Deduplication Storage

Deduplication is a technique for space-efficient data storage (see [43] 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 [12]. 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-size 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 deduplication storage. It implements the encryption layer based on *message-locked encryption (MLE)* [9], [10], 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 [16]. This ensures that the encrypted chunks derived from duplicate chunks still have identical content, thereby being compatible with deduplication.

Historical MLE [10] builds on some *publicly available* function (e.g., cryptographic hash function [16]) 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* [10]. 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 [16]), 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, server-aided MLE [9] introduces a global secret and protects the key generation process against public access. Its idea is to derive the MLE key of each chunk based on both the global secret and the cryptographic hash of this 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 [10].

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.

B. Metadata Storage Overhead

Section I reviews 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 $f \times (L+P)$ of metadata storage [40]–[42], 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 [42], the size of key metadata is 32 bytes (e.g., for AES-256 encryption keys), and the chunk size is 8 KB [42]. 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 $50\times$ [42] 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 VI-C 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.

III. Related Work

Some studies organize metadata in efficient ways to improve deduplication performance [11], [26], [30], [45] or storage efficiency [27]. For example, DDFS [45], Sparse Index [26], and Extreme Binning [11] 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.* [30] 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 [13] starts with small chunks, and groups adjacent duplicate small chunks into a big chunk for space-efficient metadata management. FBC [29] and Subchunk [36] 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 [23] 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 [18], [19], [33] 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.* [31] propose four approaches to replace the fingerprints in file recipes by short codewords, so as to compress deduplication metadata (see Section VI-C 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 management.** Dekey [24] 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 [44] and REED [34] 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 [34], [44] degrade the storage efficiency achieved by deduplication.

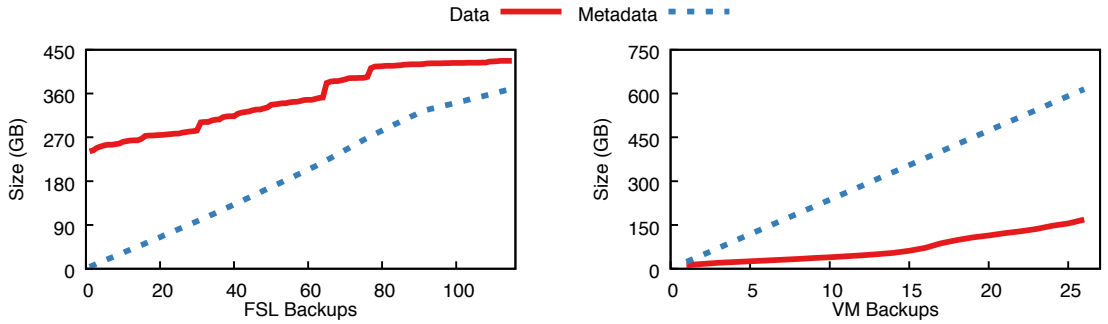
This paper is also related to Lamassu [37] that implements transparent metadata management in encrypted deduplication storage. Lamassu places metadata into some reserved sections of file data, so as to be compatible with different applications without significant changes. However, these metadata sections are randomly encrypted, and cannot be deduplicated along with data for storage savings.

Some encrypted deduplication schemes [5], [8], [28], [38] combine encryption and deduplication in different ways than MLE, yet they incur high performance overhead and are not readily implemented. This paper targets metadata storage in MLE-based encrypted deduplication.

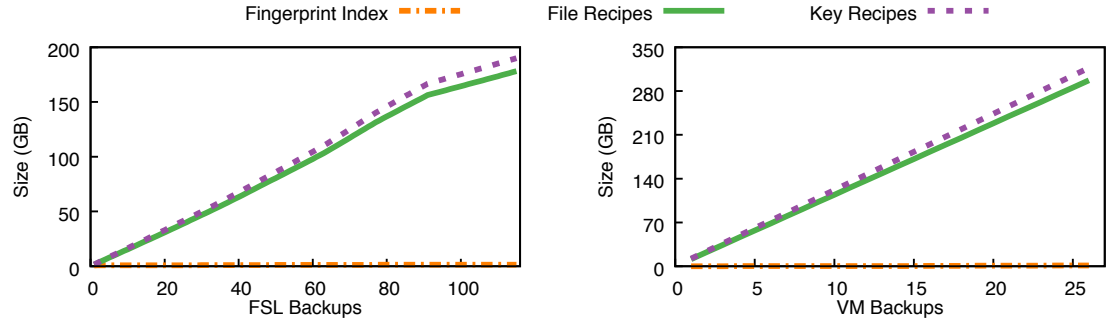
IV. Metadedup

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 [42]. 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.

Figure 2 presents the architecture of Metadedup, which builds on the client-server model. Each user installs a *client* on its co-located machine for processing backup files. It uploads the encrypted file data, as well as the corresponding deduplication metadata and key metadata, to a remote storage system that employs encrypted deduplication. Here, we assume that the communication channels are carefully protected (e.g., via



(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.

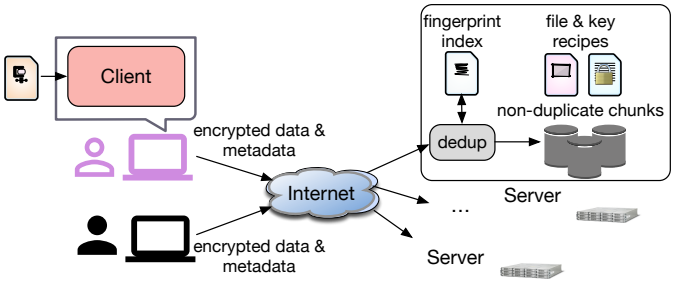


Fig. 2: Metadedup architecture.

SSL/TLS), so as to address network eavesdropping. The remote storage system deploys *servers* for data management. The server maintains a fingerprint index for chunk-based deduplication, and only stores the non-duplicate chunks that have unique content with existing chunks. It also keeps file recipes and key recipes for the reconstruction of original files.

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 II-B), 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 files to deduplication storage, compared to existing encrypted deduplication storage systems without metadata suppression.

In the following, we define the threat model of Metadedup, and present its design in details. Finally, we present the security analysis of Metadedup design based on our threat model.

A. 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 (see Figure 2). 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 [7], [14], [18], [22], [25] that address different threats against encrypted deduplication systems (see

Section V). For example, a malicious client may abuse client-side deduplication to learn whether other users have already stored certain files [18], [19], and Metadedup can defeat the side-channel leakage by adopting the countermeasure that enforces server-side deduplication on cross-user data [25]; 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 V).

We do not consider the threats that exploit the leakage of access patterns [20], although Metadedup can work in conjunction with the related countermeasures [39]. Metadedup can also be deployed with a private server to tolerate Byzantine faults [15], or with data auditing protocols [7], [22] to efficiently check the integrity of outsourced files against malicious corruptions.

B. Design

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 [23]. 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 overhead 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 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* [17]. 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], [34] 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.

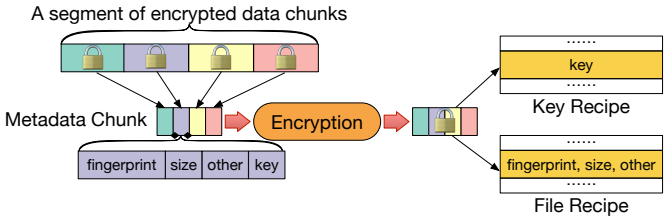


Fig. 3: Overview of metadata management in Metadedup.

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 II-A) 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 interactions in key generation [9], [34], [44], so as to preserve high performance. 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 IV-D).

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 have random data patterns that are less likely to be further compressed. Also, it does not apply deduplication to file recipes or key recipes.

C. Basic Operations

We show how we incorporate variable-size segmentation and metadata management into basic operations. We first summarize

Algorithm 1 Write operation of Metadedup

Client input: target file $file$, client's master key key
1: Initialize file and key recipes: $fRecipe, kRecipe$
2: Divide $file$ into data chunks $\{dChunk\}$
3: **for** each $dChunk$ **do**
4: $dkey \leftarrow$ key derived from $dChunk$
5: $[dChunk]_{dkey} \leftarrow$ encryption of $dChunk$ with $dkey$
6: **end for**
7: Divide $\{[dChunk]_{dkey}\}$ into segments $\{Seg\}$
8: **for** each Seg **do**
9: Initialize metadata chunk $mChunk$
10: **for** each $[dChunk]_{dkey}$ in Seg **do**
11: Add metadata of $[dChunk]_{dkey}$ into $mChunk$
12: **end for**
13: $mkey \leftarrow$ cryptographic hash of $mChunk$
14: $[mChunk]_{mkey} \leftarrow$ encryption of $mChunk$ with $mkey$
15: Add deduplication metadata of $[mChunk]_{mkey}$ into $fRecipe$
16: Add $mkey$ into $kRecipe$
17: **end for**
18: $[kRecipe]_{key} \leftarrow$ encryption of $kRecipe$ with key
19: Upload:
$fRecipe, [kRecipe]_{key}, \{[dChunk]_{dkey}\}, \{[mChunk]_{mkey}\}$

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

Algorithm 2 Restore operation of Metadedup

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

Server input: $fRecipe, \{[mChunk]_{mkey}\}$ and $[kRecipe]_{key}$
2: Receive $name$
3: Retrieve $fRecipe$ and $[kRecipe]_{key}$ based on $name$
4: Retrieve $\{[mChunk]_{mkey}\}$ based on $fRecipe$
5: Send $fRecipe, \{[mChunk]_{mkey}\}$ and $[kRecipe]_{key}$

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

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

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

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 2-6). It creates segments based on encrypted data chunks (Line 7), collects the metadata in each segment into a metadata chunk (Lines 9-12), and encrypts the metadata chunk using historical MLE (Lines 13-14). It adds the deduplication metadata and key metadata of the metadata chunk into the file and key recipes, respectively (Lines 15-16). It further encrypts the key recipe with its master key (Line 18), and uploads the following information to the server (Line 19): (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 store the unique ones (Lines 21-22). It also stores the file recipe and encrypted key recipe (Line 23).

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 3-5). In the second round, the client decrypts the key recipe and metadata chunks (Lines 7-11), and requests file data (Line 12); the server retrieves and sends the encrypted data chunks back to the client (Lines 14-15). The client decrypts each data chunk based on the corresponding key in metadata chunks (Lines 17-20), and finally assembles the data chunks to reconstruct the original file (Line 21).

D. Security Analysis

In Metadedup, each data chunk remains protected by a key derived from its content, so the confidentiality guarantees for data chunks are retained in the context of encrypted deduplication. Specifically, depending on how to derive the MLE keys of data chunks, Metadedup achieves two security levels (see Section II-A): If server-aided MLE is applied, it provides confidentiality guarantees for both predictable and unpredictable data chunks; otherwise, if historical MLE is applied, it provides confidentiality guarantees for unpredictable chunks. In the following, we analyze the confidentiality for metadata chunks in both cases.

Case 1: Metadata confidentiality under server-aided MLE. In this security level, the adversary cannot infer any original metadata information from the encrypted data chunks. Thus, we only need to ensure that the stored metadata chunks also do not leak metadata information.

Suppose that an adversary can access any metadata content that includes the key recipes, file recipes, metadata chunks, and fingerprint index. Since the adversary cannot compromise any master key (that is used to encrypt the key recipes) or any hash key (that is used to encrypt a metadata chunk), the encrypted key recipes and metadata chunks cannot be reverted. Although the fingerprint index and file recipe are not encrypted, they include the deduplication metadata only for encrypted chunks and do not leak any information about the original content.

Suppose that the adversary now further obtains the master keys and hash keys (of some metadata chunks) from some compromised clients. Nevertheless, these compromised keys cannot be used to decrypt other unauthorized key recipes and metadata chunks, because this metadata information is protected by independent keys (e.g., a per-client key for key recipes and a per-chunk key for each metadata chunk).

Case 2: Metadata confidentiality under historical MLE. The above analysis has shown that metadata chunks are secure if their corresponding data chunks are fully protected. However, in the security level under historical MLE, an adversary can derive metadata (e.g., keys) from data chunks and arbitrarily construct metadata chunks. Since Metadedup protects metadata chunks using historical MLE, the adversary can launch the offline brute-force attack (see Section II-A) to infer the original contents in target metadata chunks.

We argue that the offline brute-force attack against encrypted metadata chunks is much more computationally expensive than against encrypted data chunks. Recall that each metadata chunk consists of the metadata of multiple encrypted data chunks. Thus, an adversary needs to include different combinations of encrypted data chunks to construct a potential encrypted metadata chunk for the offline brute-force attack, yet the number of combinations is exhaustively high. In contrast, to launch the offline brute-force attack against encrypted data chunks, the adversary only needs to sample each possible data chunk to test (see Section II-A).

In the following, we conduct a simple analysis to justify that the offline brute-force attack against a metadata chunk incurs a huge time cost and hence is computationally infeasible in practice. Suppose that each data chunk is known to be drawn from a finite set that includes a total of n distinct data chunks. Let c be the average number of data chunks in a segment.

To compute the metadata of an encrypted data chunk, the adversary applies a hash function once to derive the MLE key, encrypts the data chunk, and applies the hash function again to derive the fingerprint. We estimate the running time of computing the metadata of each data chunk as:

$$T_{\mathsf{meta}} = 2 \times T_{\mathsf{hash}} + T_{\mathsf{enc}},$$

where T_{hash} and T_{enc} denote the running times of the hash and encryption functions, respectively.

To construct a metadata chunk, the adversary assembles the metadata of c encrypted data chunks in order. In fact, each data chunk in the finite set may contribute metadata, and we assume that the metadata contribution of a data chunk does not affect that of any other data chunk (i.e., the events are mutually independent). Here, we consider the total number of combinations of c *distinct* encrypted data chunks as:

$$N_{\mathsf{assemble}} = \prod_{i=0}^{c-1}(n-i) = \frac{n!}{(n-c)!},$$

where $n!$ and $(n-c)!$ are the factorials of n and $n-c$, respectively. Each combination corresponds to a metadata chunk, and thus the adversary needs to test N_{assemble} possible metadata chunks to see if any of them is encrypted to the target encrypted metadata chunk based on historical MLE. Note that the adversary may test more metadata chunks than N_{assemble} in practice, in order to address the case that metadata chunks include the metadata of duplicate encrypted data chunks. Thus, N_{assemble} can be viewed as the *lower bound* of the number of metadata chunks that the adversary needs to construct, and the total construction time is:

$$T_{\mathsf{construct}} = N_{\mathsf{assemble}} \times T_{\mathsf{meta}}.$$

For each constructed metadata chunk, the adversary checks whether it is the original input of the target encrypted metadata chunk, and each check requires one hash (to derive the MLE key) and one encryption. This implies that the running time of the equality check for all metadata chunks:

$$T_{\mathsf{check}} = N_{\mathsf{assemble}} \times (T_{\mathsf{hash}} + T_{\mathsf{enc}}).$$

We can now estimate the average running time of the offline brute-force attack against metadata chunks as:

$$T_{\mathsf{attack}} = T_{\mathsf{construct}} + T_{\mathsf{check}}.$$

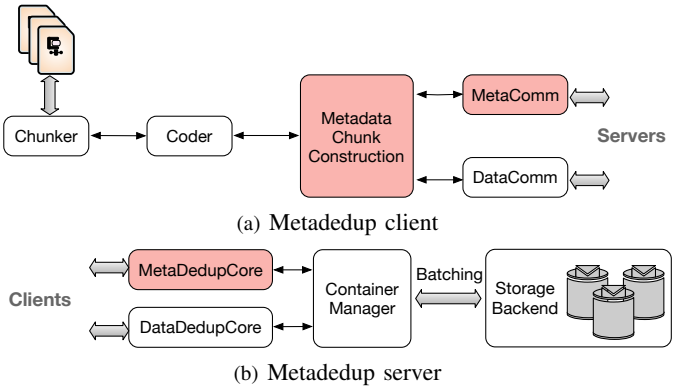
We consider an example to understand how large T_{attack} is. Suppose that the average segment size is 1 MB, and the average chunk size is 8 KB. Then, $c = \frac{1\,\mathrm{MB}}{8\,\mathrm{KB}} = 128$. According to [1], we assume that it takes $T_{\mathsf{enc}} = 48\mu s$ and $T_{\mathsf{hash}} = 37\mu s$ to perform the encryption and hash operations on a chunk of 8 KB, respectively (the equivalent encryption and hash speeds are 163 MB/s and 212 MB/s, respectively). We estimate T_{attack} as follows.

$$\begin{aligned} T_{\mathsf{attack}} &= (3 \times T_{\mathsf{hash}} + 2 \times T_{\mathsf{enc}}) \times \frac{n!}{(n-c)!} \\ &\geq (3 \times T_{\mathsf{hash}} + 2 \times T_{\mathsf{enc}}) \times c! \\ &\approx 7.94 \times 10^{211} s. \end{aligned}$$

If the attack is implemented serially, the total running time is at least $7.94 \times 10^{211} s$, which is more than 10^{204} years. Even the attack can be implemented in parallel, it is computationally infeasible to work in reasonable time.

V. 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. CDStore encodes each data secret on the client side into s shares via a (s,t)-deduplication-aware secret sharing algorithm (where $s \geq t > 0$) that has three properties: (i) *reliability*, i.e., the data chunk can



(a) Metadedup client

(b) Metadedup server

Fig. 4: Implementation of the Metadedup prototype (the modules that are newly added to CDStore are colored by red).

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. The s shares are stored in s distinct servers. In our experiments, we set $s = 4$ and $t = 3$. 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. Note that CDStore does not apply deduplication to metadata; the goal of Metadedup is to augment CDStore with metadata deduplication.

In our Metadedup implementation, we treat each share as an encrypted data chunk. Thus, we generate s share streams that are written to s servers. For each of the s share streams, we generate metadata chunks as well as the corresponding file recipe and key recipe (i.e., we have s file recipes and s key recipes in total).

We follow the modular approach as in CDStore to implement Metadedup. Figure 4 shows how Metadedup adds new modules to CDStore. We use OpenSSL 1.0.2p [3] to implement the cryptographic operations in Metadedup. The current Metadedup prototype, including the original CDStore modules, contains about 7,500 lines of code.

A. Modules

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

Write. As in CDStore, a client writes a file by partitioning it into data secrets via the *chunker* module, which implements Rabin fingerprinting [35] for variable-size chunking. Rabin fingerprinting takes the minimum, average and maximum sizes of chunking as inputs, which we now fix as 2 KB, 8 KB and 16 KB, respectively. Also, the client encodes each data secret into s shares via the *coder* module.

Metadedup introduces a new *metadata chunk construction* module. It takes s share streams as input, where each share is treated as an encrypted data chunk. It partitions each of the share streams into segments using variable-size segmentation, and generates a metadata chunk for each segment. It encrypts metadata chunks using historical MLE. It finally prepares a file recipe and a key recipe for each of the s share streams, where each key recipe is encrypted by the client's master key.

Metadedup adds a *MetaComm* module for the communication of metadata chunks with servers, in addition to the original *DataComm* module for data communication in CDStore. Specifically, in MetaComm, it performs intra-user deduplication on encrypted metadata chunks, and only sends (i) unique encrypted metadata chunks, (ii) file recipes, and (iii) encrypted key recipes to corresponding servers. 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 and key recipes, 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 out of s servers to request for the shares of the file in a two-round manner (see Algorithm 2 in Section IV-C). Each server first returns the metadata, including the file recipe, the encrypted key recipe, and the encrypted metadata chunks, to the client. The client decrypts the key recipe and metadata chunks, and reconstructs the deduplication metadata and key metadata of data shares. Then the client retrieves the data shares from the server. It recovers the data secret from t shares. Finally, it assembles the recovered data secrets to the original file.

B. Discussion

We discuss several implementation details in our prototype implementation.

Parallelization. We follow CDStore to parallelize the operations of Metadedup through multi-threading. We first parallelize the processing of different modules. 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, while including the real filename into the key recipe that is protected together with the key metadata.

Restore optimization: Our current implementation needs two rounds of interactions between client and servers to restore a file (i.e., the client first retrieves the metadata, followed by

TABLE I: Metadedup's microbenchmarks of metadata flow for different average segment sizes.

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**

Note: The write and restore throughputs are computed based on original data size (i.e., 10 GB).

retrieving the shares). How to further optimize the restore performance is our future work.

VI. Evaluation

We conduct microbenchmarks, prototype experiments, and trace-driven simulation on Metadedup. Our evaluation goal is to answer three high-level questions:

- What is the performance penalty of Metadedup compared to the conventional encrypted deduplication approach that does not apply metadata deduplication?
- Can Metadedup achieve storage savings through metadata deduplication?
- Can Metadedup further improve storage savings when being combined with existing compression approaches?

A. Microbenchmarks

We first implement the metadata workflow of Metadedup algorithms and study the computational performance of each processing step. Here, we do not consider the client-server communication as in our prototype evaluation (which is addressed in Section VI-B).

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 Metadedup 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 metadata write and restore procedures. Specifically, the write procedure includes: (i) *segmentation*, which groups data chunks into segments; (ii) *metadata handling*, which creates, encrypts, and deduplicates metadata chunks; and (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 I 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 above 9 GB/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.

B. Prototype Experiments

We now study the performance of our Metadedup prototype and compare it with CDStore, which does not support metadata deduplication. Our evaluation setting of both Metadedup and CDStore is as follows. We deploy a client instance on a machine that has a 24-core 2.40GHz Intel(R) Xeon(R) CPU E5-2620 v3 and 32 GB RAM, and four server instances on a different machine that has a 20-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.

Like the prior work [25], we configure the coder module (of both Metadedup and CDStore) with two threads to boost performance. For performance tests, we present the average results over 10 runs with the 90% confidence intervals.

Experiment A.1 (Performance impact of segment size). We evaluate Metadedup under different average segment sizes, and compare the results with those of CDStore. We create and upload 10 GB of unique data to four servers, and then download them from any three servers. We evaluate the write and restore speeds.

Figure 5 presents the comparison results. The write speed of Metadedup approximates that of CDStore. For example, when the average segment size is 1 MB, Metadedup achieves the write speed of 60.33 MB/s, only 2.54 MB/s less than that of CDStore; this implies that the additional overhead is 4.05%.

In addition, the restore speed of Metadedup increases with the average segment size, because the number of metadata chunks to be restored is reduced. When we increase the average segment size to 4 MB, the restore speed of Metadedup achieves 93.62 MB/s, slower than that of CDStore by 13.85%. This reason is that Metadedup needs to serially retrieve metadata, followed by data.

Experiment A.2 (Scalability to data size). We now evaluate the scalability of Metadedup, and also compare the results with those of CDStore. We fix the average segment size of Metadedup as 1 MB, and examine the write and restore speeds for processing the unique data under different sizes.

Figure 6 shows the evaluation results. We observe that both write and restore speeds of Metadedup degrade with the total size of unique data, because Metadedup needs to process more metadata chunks. For example, when we vary the size of unique data from 1 GB to 20 GB, the write and restore speeds drop from 63.30 MB/s and 87.92 MB/s to 45.51 MB/s and 70.69 MB/s, respectively. Although Metadedup suffers from speed degradation as the size of unique data increases, when being compared to CDStore, Metadedup only adds small write overhead (e.g., by 6.19% on average) and medium restore overhead (e.g., by 23.23% on average).

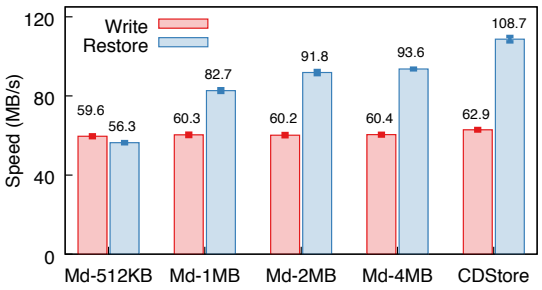


Fig. 5: Experiment A.1 (Performance impact of segment size). The notation Md-X denotes the Metadedup instance configured with an average segment size of X.

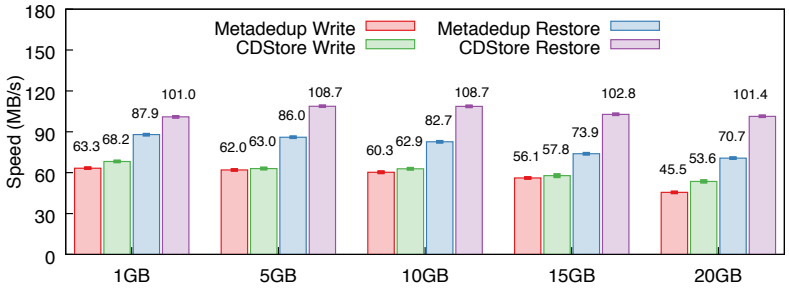


Fig. 6: Experiment A.2 (Scalability to data size). The x-axis indicates the sizes of unique data that are tested for write and restore in the evaluation.

C. Trace-driven Simulation

We now evaluate the storage savings of Metadedup via trace-driven simulation.

Datasets. 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], [40], [41]. 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], [34]. 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.

Methodology. To conduct trace-driven simulation, we build a simulator based on the metadata size assumptions in Section II-B. The simulator adds the FSL or VM backups to storage in the order of their creation times, and evaluates two metrics: (i) *storage saving*, the percentage of the total size of metadata (excluding the fingerprint index) reduced by the approaches we consider; and (ii) *index overhead*, the percentage of additional storage cost to the fingerprint index.

Experiment B.1 (Storage impact of segment size). We first study the impact of the average segment size in Metadedup. Table II presents the simulation results of storage saving and index overhead after storing all backups, where *raw* denotes the original metadata size without deduplication or compression.

TABLE II: Experiment B.1 (Storage impact of segment size).

Components/Metrics		Raw	512KB	1MB	2MB	4MB
FSL	File recipes (GB)	178.191	1.932	0.965	0.481	0.240
	Key recipes (GB)	190.070	2.061	1.030	0.513	0.256
	Fingerprint index (GB)	1.385	1.412	1.400	1.393	1.390
	Metadata chunks (GB)	–	6.806	7.372	8.041	8.818
	Total (GB)	**369.646**	**12.211**	**10.767**	**10.428**	**10.704**
	Storage saving	**–**	**97.07%**	**97.46%**	**97.55%**	**97.47%**
	Index overhead	**–**	**1.94%**	**1.07%**	**0.60%**	**0.33%**
VM	File recipes (GB)	297.070	0.579	0.290	0.145	0.073
	Key recipes (GB)	316.875	0.618	0.309	0.155	0.077
	Fingerprint index (GB)	1.232	1.256	1.244	1.241	1.237
	Metadata chunks (GB)	–	24.985	25.138	36.359	40.817
	Total (GB)	**615.177**	**27.438**	**26.981**	**37.900**	**42.204**
	Storage saving	**–**	**95.74%**	**95.81%**	**94.03%**	**93.33%**
	Index overhead	**–**	**1.91%**	**0.96%**	**0.70%**	**0.39%**

For both datasets, the storage saving first increases with the average segment size, because a larger segment size (and hence larger metadata chunks) reduces the metadata of metadata chunks. For example, when the average segment sizes are 2 MB and 1 MB, the FSL and VM datasets achieve the highest storage savings of 97.55% and 95.81%, respectively. The storage saving decreases due to a small deduplication factor for large metadata chunks. Nevertheless, the storage savings under all average segment sizes are higher than 93%. In addition, the index overheads decrease with the average segment size and are lower than 2%.

Experiment B.2 (Storage comparison with compression approaches). We fix the average segment size of Metadedup at 1 MB, and compare its storage efficiency and index overhead with those of file recipe compression approaches [31]. We do not consider other approaches for comparison, as they either require the deduplication information of chunks [13], [23], [29], [36] that leads to side-channel leakage in encrypted deduplication or add additional metadata [24] and data [34], [44] overheads (see Section III). We elaborate how we configure the baseline compression approaches [31], followed by the evaluation results.

- *Zero compression (ZC)* replaces the metadata of zero-filled chunks by one-byte special codes, so as to reduce the sizes of file recipe and key recipe.
- *Page-based compression (PC)* assumes the availability of fingerprint index, and replaces the deduplication metadata

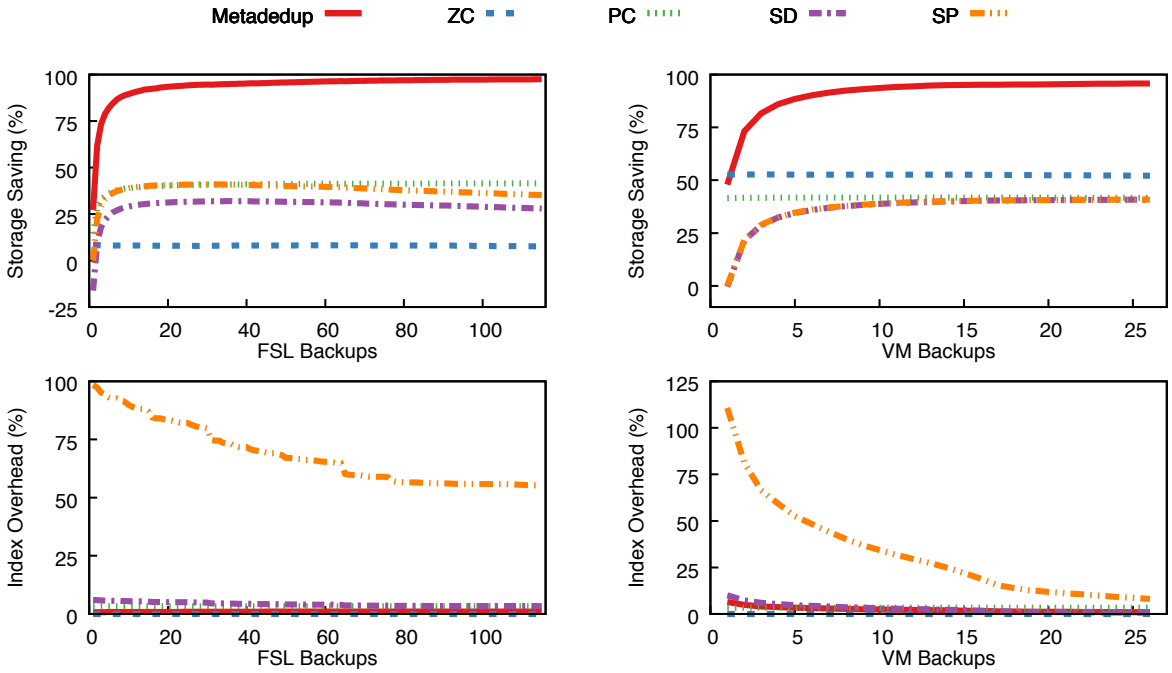


Fig. 7: Experiment B.2 (Storage comparison with compression approaches).

of each chunk by a codeword derived from its index offset. We configure the length of each codeword at 4 bytes [31].

- *Statistical directory (SD)* encodes the deduplication metadata of low-entropy chunks by fixed-size codewords. Like the prior work [31], we derive the entropy information from the first backup, and allocate each codeword with 3 bytes.
- *Statistical prediction (SP)* exploits the locality of neighboring chunks under the logical order. For each chunk, it stores u codewords that are mapped from the deduplication metadata of the most likely neighbors of this chunk. For compression, it replaces the deduplication metadata of these neighbors by corresponding codewords. Like the prior work [31], we derive the neighboring information from the first backup, and set u to 2.

Figure 7 shows the cumulative storage savings and index overheads of all considered approaches for the FSL and VM datasets. The storage savings of Metadedup increase with the number of backups and significantly outperform those of baseline approaches. For example, they finally achieve 97.46% and 95.81% for the FSL and VM datasets, respectively. In the baseline approaches, the savings of ZC are almost unchanged, such as 7.58-8.29% for the FSL dataset and 52.14-52.74% for the VM dataset. The savings of PC gradually increase during backup periods, since some metadata in later backups have already been encoded, and they finally achieve 41.51% and 41.71% for the FSL and VM datasets, respectively. SD and SP only have savings after the initial backups, since they need to first extract the entropy and neighboring information, respectively [31]. One special note is that SD even incurs additional overheads of 16.11% and 0.42% for the initial FSL and VM backups, respectively. The reason is that it maintains a codeword index to map assigned codewords back to the corresponding deduplication metadata [31]. Such overhead can be covered in following backups, and SD finally achieves the savings of 27.99% and 40.79% for the FSL and VM datasets, respectively. The corresponding final savings of SP also reach 35.20% and 40.81%, respectively.

In addition, we observe that Metadedup, PC, and ZC incur low index overheads, such as less than 3.33% in both datasets, during the whole backup time. SD and SP incur relatively high index overheads in initial backups, since they need to store some fingerprint-to-codeword mappings in the fingerprint index. For example, SP stores u mappings in each fingerprint index entry to map the fingerprints of some chunks that are most likely to come after the corresponding chunk to short codewords; this leads to the index overheads of 97.34% and 110.76% for the FSL and VM datasets, respectively. Such overhead can be amortized in following backups. The index overheads of SP finally decrease to 55.35% and 8.04% for the FSL and VM datasets, respectively, while those of SD drop down to 3.38% and 0.73%.

Experiment B.3 (Combined with compression). We finally examine the effectiveness of combining Metadedup with the baseline compression approaches [31] to reduce the size of recipes. We focus on two combined approaches: (i) Metadedup + ZC and (ii) Metadedup + PC, which apply Metadedup first and then use ZC and PC to suppress the metadata of metadata chunks, respectively. We do not consider other combination options as they either incur high index overhead (e.g., combined with SP) or lead to small storage savings (e.g., combined with SD). We fix the average segment size of Metadedup as 1 MB.

TABLE III: Experiment B.3 (Combined with compression).

Components/Metrics		FSL	VM
Metadedup only	File recipes (GB)	0.965	0.290
	Key recipes (GB)	1.030	0.309
	Fingerprint index (GB)	1.400	1.244
	Metadata chunks (GB)	7.372	25.138
	Total (GB)	**10.767**	**26.981**
	Storage saving	**97.46%**	**95.81%**
	Index overhead	**1.07%**	**0.96%**
Metadedup + ZC	File recipes (GB)	0.923	0.145
	Key recipes (GB)	0.984	0.155
	Fingerprint index (GB)	1.400	1.244
	Metadata chunks (GB)	7.372	25.138
	Total (GB)	**10.679**	**26.682**
	Storage saving	**97.48%**	**95.86%**
	Index overhead	**1.07%**	**0.96%**
Metadedup + PC	File recipes (GB)	0.129	0.039
	Key recipes (GB)	1.030	0.309
	Fingerprint index (GB)	1.401	1.244
	Metadata chunks (GB)	7.372	25.138
	Decoding mapping (GB)	0.017	0.013
	Total (GB)	**9.949**	**26.730**
	Storage saving	**97.68%**	**95.85%**
	Index overhead	**1.11%**	**1.00%**

Note that PC needs to store page offsets in fingerprint index entries for encoding, and maintain a reverse mapping for decoding [31].

Table III presents the simulation results after storing all backups, and we also include the results of Metadedup only for reference. By combining Metadedup with ZC, we can further reduce the sizes of both file recipe and key recipe, from 0.97 GB and 1.03 GB to 0.92 GB and 0.98 GB in the FSL dataset, as well as from 0.29 GB and 0.31 GB to 0.15 GB and 0.16 GB in the VM dataset, respectively. Such recipe reduction is more effective (e.g., about 50%) for the VM dataset, as VM images include large regions of zero chunks [21]. This only leads to negligible storage savings of metadata, such as 0.02% and 0.05% for the FSL and VM datasets, respectively. Similarly, the combination of Metadedup and PC brings few additional savings by about 0.22% for the FSL dataset and 0.04% for the VM dataset.

Our results suggest that Metadedup can only be marginally improved by compression approaches, as compression cannot apply to the physical metadata chunks that take more than 60% of overall metadata in Metadedup (see Table II). Thus, Metadedup itself sufficiently achieves high storage saving of metadata.

VII. 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 extensively evaluate Metadedup from microbenchmarks, prototype experiments, and trace-driven simulation. We show that Metadedup significantly suppresses the storage space of metadata, while incurring limited performance and indexing penalties.

Acknowledgments

This work was supported in part by grants by the National Key R&D Program of China (Grant No. 2017YFB0802300), the National Natural Science Foundation of China (Grant No. 61602092), the Open Research Project of the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences (Grant No. 2019-MS-05), and the Research Grants Council of Hong Kong (CRF C7036-15G).

References

[1] "Bearssl performance," https://bearssl.org/speed.html.
[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] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev, "Message-locked encryption for lock-dependent messages," in *Proc. of CRYPTO*, 2013.
[6] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud," in *Proc. of ACM CCS*, 2015.
[7] 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.
[8] M. Bellare and S. Keelveedhi, "Interactive message-locked encryption and secure deduplication," in *Proc. of PKC*, 2015.
[9] M. Bellare, S. Keelveedhi, and T. Ristenpart, "DupLESS: Server-aided encryption for deduplicated storage," in *Proc. of USENIX Security*, 2013.
[10] ——, "Message-locked encryption and secure deduplication," in *Proc. of EUROCRYPT*, 2013.
[11] 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.
[12] J. Black, "Compare-by-hash: A reasoned analysis," in *Proc. of USENIX ATC*, 2006.
[13] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage*, vol. 2, no. 4, pp. 424–448, 2006.
[14] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Proc. of USENIX ATC*, 2013.
[15] D. Dobre, P. Viotti, and M. Vukolič, "Hybris: Robust hybrid cloud storage," in *Proc. of ACM SoCC*, 2014.
[16] 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.
[17] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," HPL-2005-30R1, Tech. Rep., 2005.
[18] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proc. of ACM CCS*, 2011.
[19] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010.
[20] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. of NDSS*, 2012.
[21] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proc. of ACM SYSTOR*, 2009.
[22] A. Juels and B. S. Kaliski, Jr., "PORs: Proofs of retrievability for large files," in *Proc. of ACM CCS*, 2007.
[23] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *Proc. of USENIX FAST*, 2010.
[24] 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*, vol. 25, no. 6, pp. 1615–1625, 2014.
[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] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers," in *Proc. of ACM CCS*, 2015.
[29] G. Lu, Y. Jin, and D. H. Du, "Frequency based chunking for data de-duplication," in *Proc. of IEEE MASCOTS*, 2010.
[30] 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.
[31] D. Meister, AndréBrinkmann, and T. Süß, "File recipe compression in data deduplication systems," in *Proc. of USENIX FAST*, 2013.
[32] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *Proc. of USENIX FAST*, 2011.
[33] 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.
[34] 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*, vol. 13, no. 1, pp. 9:1–9:30, 2017.
[35] M. O. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technology, Harvard University. Tech. Report TR-CSE-03-01, 1981.
[36] B. Romański, L. Heldt, W. Kilian, K. Lichota, and C. Dubnicki, "Anchor-driven subchunk deduplication," in *Proc. of ACM SYSTOR*, 2011.
[37] P. Shah and W. So, "Lamassu: Storage-efficient host-side encryption," in *Proc. of USENIX ATC*, 2015.
[38] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A secure data deduplication scheme for cloud storage," in *Proc. of FC*, 2014.
[39] 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.
[40] 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*, vol. 14, no. 2, pp. 13:1–13:25, 2018.
[41] ——, "A long-term user-centric analysis of deduplication patterns," in *Proc. of IEEE MSST*, 2016.
[42] 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.
[43] 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*, vol. 104, no. 9, pp. 1681–1710, 2016.
[44] 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.
[45] 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.