Encrypted Data Reduction: Removing Redundancy from Encrypted Data in Outsourced Storage

JIA ZHAO, The Chinese University of Hong Kong, China
ZUORU YANG, The Chinese University of Hong Kong, China
JINGWEI LI, University of Electronic Science and Technology of China, China
PATRICK P. C. LEE, The Chinese University of Hong Kong, China

Storage savings and data confidentiality are two primary goals for outsourced storage. However, encryption by design destroys the content redundancy within plaintext data, so there exist design tensions when combining encryption with data reduction techniques (i.e., deduplication, delta compression, and local compression). We present EDRStore, an outsourced storage system that realizes encrypted data reduction to achieve both storage savings and data confidentiality. EDRStore's core idea is a careful design of the encryption and data reduction workflows. It proposes new key generation and encryption schemes to preserve the content similarity of encrypted data for deduplication and delta compression. It further proposes selective local compression based on content similarity, so as to achieve storage savings of encrypted data from both delta compression and local compression. Evaluation on real-world datasets shows that EDRStore achieves higher storage savings than existing encrypted storage approaches and incurs moderate performance overhead compared with plaintext storage.

CCS Concepts: • **Information systems** → **Cloud based storage**; **Deduplication**; • **Security and privacy** → **Management and querying of encrypted data**.

Additional Key Words and Phrases: Encrypted data reduction, cloud storage

ACM Reference Format:
Jia Zhao, Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. 2024. Encrypted Data Reduction: Removing Redundancy from Encrypted Data in Outsourced Storage. *ACM Trans. Storage* 1, 1, Article 1 (January 2024), 29 pages. https://doi.org/0001.0001

1 INTRODUCTION

The global data volume reached 64.2 zettabytes in 2020 and is expected to be more than doubled in the next five years [30]. Such unprecedented data growth motivates enterprises and individuals to increasingly outsource storage management to public clouds [57]. There are two key requirements for outsourced storage to be practical: (i) *storage savings* (i.e., the least possible storage resources

This work was supported in part by the National Key R&D Program of China (2022YFB4501200), Key Research Funds of Sichuan Province (24GJHZ0225), Fundamental Research Funds for Chinese Central Universities (ZYGX2021J018), and Innovation and Technology Commission of Hong Kong (GHX/076/20). Corresponding author: Jingwei Li.

Authors' addresses: Jia Zhao, jzhao@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Zuoru Yang, zryang@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Jingwei Li, jwli@uestc.edu.cn, University of Electronic Science and Technology of China, China; Patrick P. C. Lee, pclee@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China.

1553-3077/2024/1-ART1 $15.00
https://doi.org/0001.0001

are incurred to minimize outsourcing costs) and (ii) *data confidentiality* (i.e., the outsourced data is protected from unauthorized access [31]).

To achieve storage savings, a common approach is to apply data reduction techniques to remove the content redundancy at different granularities, so as to reduce the footprints in outsourced storage. There are three widely used data reduction techniques, namely *deduplication*, *delta compression*, and *local compression*, and they can be applied in sequence to maximize storage savings [51, 59, 60, 72, 77]. Specifically, a storage system first applies chunk-based deduplication to remove duplicate chunks with identical content. It then applies delta compression to remove the redundancy of non-duplicate but similar chunks that have large fractions of identical content. Finally, it applies local compression to encode the remaining non-duplicate chunks into smaller-size codes to remove the internal redundancy. Field studies show that deduplication can save the space of primary workloads by 2× [46] and that of backup workloads by even up to 50× [63]. For post-deduplicated data, local compression brings 3.1× savings, while delta compression and local compression together further bring 10.2× savings [59].

To realize data confidentiality, a client should first encrypt outsourced data before uploading them to the cloud, so as to protect the data from unauthorized access; the cloud further applies data reduction to the encrypted content originating from multiple clients (note that applying data reduction prior to encryption poses practical concerns; see §2.2). Traditional symmetric encryption, however, is incompatible with any data reduction technique since each client encrypts its data with its own distinct secret key, so any duplicate content shared among the data across clients will be destroyed after encryption. Thus, extensive studies in the literature explore the combination of data reduction and encryption for both storage savings and data confidentiality. One class of approaches is *encrypted deduplication* [13, 14, 23, 55, 71, 74], which combines encryption and deduplication by encrypting the original data chunks with a symmetric key derived from the content of each chunk, so that duplicate data chunks are deterministically encrypted to the duplicate encrypted chunks that can be removed by deduplication. Another class of approaches is *encrypted local compression* [18, 36, 52, 73], which performs local compression on data chunks (which can be first padded with dummy bytes to hide chunk lengths) and encrypts the compressed data chunks into the encrypted chunks of smaller lengths than without local compression.

We argue that existing approaches on combining data reduction and encryption remain limited in improving storage savings. First, existing approaches do not address delta compression, which is shown to provide significant space savings beyond deduplication and local compression [51, 59, 72]. Second, it is infeasible to extend existing approaches with additional data reduction techniques. For example, after encrypted deduplication, the remaining non-duplicate encrypted chunks cannot be further delta- or local-compressed, as encryption transforms the chunk data into a scrambled form that destroys the content redundancy within a chunk. To our knowledge, combining all three data reduction techniques (i.e., deduplication, delta compression, and local compression) and encryption remains unexplored in the literature.

We propose EDRStore, an outsourced storage system that supports deduplication, delta compression, and local compression for storage savings, and further combines them with encryption for data confidentiality. EDRStore carefully designs the data reduction workflow to make it applicable to encrypted data. It designs new key generation and encryption schemes that preserve the content similarity of data chunks after encryption for deduplication and delta compression. It also designs selective local compression, such that a client only locally compresses the non-similar data chunks for stable storage savings, while the remaining similar data chunks are encrypted and delta-compressed in the cloud for higher storage savings. Note that EDRStore makes a trade-off between storage savings and data confidentiality by allowing delta compression on encrypted similar chunks. We discuss its security guarantees and limitations (§5).

We evaluate EDRStore using six real-world datasets that resemble backup workloads. EDRStore has significant storage savings over encrypted deduplication, encrypted local compression, and a mix of both (by up to 6.8×, 19.7×, and 2.7×, respectively), while incurring moderate performance overhead (e.g., 22.7% of upload throughput drop) compared with data reduction on plaintext storage without encryption.

We open-source our EDRStore prototype at **https://github.com/adslabcuhk/edrstore**.

2 BACKGROUND AND PROBLEM

We present an overview of existing data reduction techniques (§2.1). We discuss how to possibly extend data reduction with encryption and pose the challenges (§2.2).

2.1 Overview of Data Reduction Techniques

In this work, we consider three data reduction techniques: deduplication, delta compression, and local compression.

Deduplication. Deduplication has been widely deployed (e.g., [21, 25, 41, 42, 65, 75]) to remove duplicate content from storage. We consider a deduplicated storage system that divides file data into non-overlapping variable-size chunks using content-defined chunking [54], so that deduplication remains effective even under content shifts. Each chunk is identified by a *fingerprint* computed from the cryptographic hash of the chunk content. The storage system keeps a key-value store, called the *fingerprint index*, to track the fingerprints of all currently stored chunks. It stores a chunk only if the chunk fingerprint is new to the fingerprint index (assuming that hash collisions of non-duplicate chunks are unlikely [15]), and hence there exists only one physical copy for duplicate chunks with the same fingerprint.

Delta compression. To achieve additional storage savings, prior studies [51, 59, 67, 68, 72] further apply delta compression to *similar* chunks after deduplication; by similar, we mean that the chunks are non-duplicate but have large fractions of the same content with changes in only a few chunk regions. Delta compression aims to remove the same byte-level content of similar chunks and record their differences.

The feature-based approach [16, 59, 72] is often used to identify similar chunks. It extracts one [16] or multiple [59, 72] *features* from each chunk to characterize the chunk content. For example, a feature can be the maximum Rabin fingerprint over all sliding windows of the chunk content, such that small changes to the chunk content are unlikely to perturb the feature value [59]. It also keeps a key-value store, called the *feature index*, to track the features of each first occurred chunk (called the *base chunk*) that contains the feature. For each new chunk to be stored, the feature-based approach queries the feature index to find the base chunk that has the most common features with the new chunk (and the number of common features exceeds some pre-specified threshold). If the base chunk is found (i.e., the new chunk is similar to the selected base chunk), it compresses the new chunk with the corresponding base chunk with *delta encoding* [43, 62] to form a *delta chunk*, whose size is typically smaller than the new chunk; otherwise, if such a base chunk is unavailable, the new chunk is stored as a base chunk.

Local compression. Chunk data often has redundant patterns with low *entropy* (a measure in information theory). Local compression (e.g., Zstandard [20] and DEFLATE [22]) can encode such chunk data into smaller-size codes without information loss. It is applied to both base chunks and delta chunks after deduplication and delta compression.

2.2 Extending Data Reduction with Encryption

Each client first encrypts the original *data chunks* (i.e., plaintexts) into *encrypted chunks* (i.e., ciphertexts) and stores only the encrypted chunks in the cloud. We review two classes of approaches that are well-studied in the literature, namely *encrypted deduplication* and *encrypted local compression*. Both classes of approaches extend data reduction with encryption to achieve both data confidentiality and storage efficiency in outsourced storage, yet we argue that they are limited in different aspects and cannot be readily extended to support all three data reduction techniques (i.e., deduplication, delta compression, and local compression).

Encrypted deduplication. We refer to the combination of encryption and deduplication as *encrypted deduplication* [13, 14, 23], which performs chunk-based encryption in a client for data confidentiality, followed by deduplication in the cloud for storage efficiency. *Message-locked encryption (MLE)* [14] is the most known cryptographic primitive for encrypted deduplication, in which each data chunk is encrypted with a key (called the *MLE key*) derived from the chunk content with some deterministic function. It encrypts duplicate data chunks (from the same or different clients) to duplicate encrypted chunks, which can be removed by deduplication in the cloud. One MLE instantiation is *convergent encryption* [23], which uses the chunk fingerprint (i.e., the cryptographic hash of the chunk content) as the MLE key.

MLE is vulnerable to offline brute-force attacks [13], in which an adversary enumerates all possible data chunks, derives the corresponding MLE keys via the deterministic function, and infers if a data chunk is mapped to any stored encrypted chunk. *Server-aided MLE* [13] defends against offline brute-force attacks by deriving an MLE key through a dedicated *key server*, which introduces a *global secret* that is known only to the key server in MLE key generation. If the global secret is secure, it is infeasible for the adversary to derive an MLE key of a data chunk. Furthermore, the key server can be extended with (i) key generation based on the oblivious pseudorandom function (OPRF) [49], which converts a data chunk into a blinded fingerprint, such that the key server is prevented from learning the client's input data chunks while still being able to return the same MLE key for duplicate data chunks; and (ii) rate-limiting of key generation, which prevents malicious clients from launching online brute-force attacks by issuing too many key generation requests. In this work, we use server-aided MLE as the encrypted deduplication component in EDRStore.

However, there will be limited storage savings by applying delta or local compression to MLE-encrypted chunks. First, similar (but non-duplicate) data chunks are assigned with different MLE keys, so encryption will destroy the content similarity among the encrypted chunks. Also, encryption forms high-entropy encrypted chunks, which prohibit local compression for further storage savings.

Note that a recent work [19] proposes to use the same MLE key to encrypt similar chunks based on bitwise-XOR operations, so as to support delta compression on encrypted chunks. We argue that such an approach has major security holes (§8).

Encrypted local compression. As encrypted chunks cannot be readily compressed, encryption should be done *after* local compression: a client first applies local compression to each data chunk, encrypts the compressed data chunk, and uploads the encrypted compressed chunk to the cloud [18, 73]. However, encrypting a compressed data chunk directly can cause length leakage, as the length of the encrypted compressed data chunk can disclose the compressibility of its original data chunk. An adversary can also launch side-channel attacks via length leakage [1, 18, 36, 52]. For example, if two encrypted chunks have similar sizes, they are likely to be originating from the data chunks with similar content [18].

To hide the length information, it is critical to pad random dummy bytes to a compressed data chunk before encryption, but at the expense of additional space. On the other hand, random padding

prohibits deduplication over compressed chunks, since duplicate data chunks could be padded with distinct dummy bytes to become unique data chunks, which cannot be removed by deduplication.

Data reduction before encryption. Recall that encrypted deduplication and encrypted local compression cannot be readily extended to support delta compression, as encryption destroys the content redundancy of a chunk (§1). One naïve solution is to apply data reduction before encryption: a client first applies deduplication, delta compression, and local compression in sequence to the data chunks, encrypts the reduced output, and uploads the encrypted output to the cloud. This solution has two drawbacks. First, it sacrifices the storage savings from cross-client deduplication, since each client cannot access the data chunks from other clients but can only perform deduplication locally. Second, the client needs to keep the fingerprint index for deduplication as well as the feature index and base chunks for delta compression, leading to significant management overhead (especially for large base chunks). Thus, deduplication and delta compression should be performed in the cloud, yet the cloud can only receive encrypted chunks due to confidentiality concerns. In summary, it is non-trivial to combine all three data reduction techniques with encryption to achieve both storage efficiency and data confidentiality for outsourced storage.

3 DESIGN OVERVIEW

We provide a high-level design overview for EDRStore. We state the design goals (§3.1), EDRStore's architecture (§3.2), and the threat model (§3.3).

3.1 Design Goals

EDRStore supports encrypted data reduction for multiple clients to securely outsource data storage to a cloud. It targets backup workloads, which have high content redundancy [59, 63], and each client uploads multiple versions of backups to the cloud. It supports the basic upload (i.e., storing a backup in the cloud) and download (i.e., restoring a backup from the cloud) operations. We do not specifically consider the overwrite and append operations, while both of them can be viewed as special cases of uploads: an overwrite operation means to upload a new backup that is modified from the previous backup by changing the contents of some chunks, while an append operation implies that EDRStore uploads a new backup with the new content added to the end of the previous backup. EDRStore applies data reduction to the new backup to achieve storage savings.

We design EDRStore with the following goals: (i) *storage savings*, which combine deduplication, delta compression, and local compression to remove content redundancy; (ii) *data confidentiality*, which protects outsourced data from unauthorized access (see §3.3 for details); (iii) *high performance*, which adds small performance overhead compared with data reduction on plaintext data chunks; (iv) *easy deployment*, which supports deployment in commodity servers and modern cloud storage providers without specialized hardware (e.g., SGX for deduplication [47, 55, 70] or GPU for delta compression [51]); and (v) *reliability*, which keeps client states in the cloud to resist client crashes.

3.2 Architecture

EDRStore extends server-aided MLE to support delta compression and local compression, as shown in Figure 1. There are three main entities in EDRStore: one or multiple clients, a key server, and a cloud. EDRStore applies local compression before encryption on the client side, while performing deduplication followed by delta compression in the cloud, so as to offload the management overhead for base chunks and index structures from clients (§2.2). Also, it maintains a dedicated key server as in server-aided MLE for key generation (§2.2).

The cloud stores the encrypted chunks of a backup originating from a client in its storage pool. Each backup is associated with two metadata files: (i) a *file recipe*, which lists the fingerprints of all

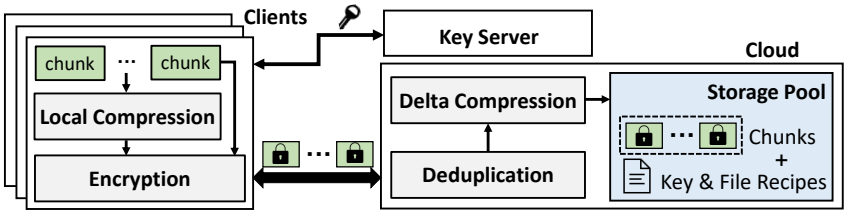


Fig. 1. EDRStore's architecture.

encrypted chunks in the backup for reconstruction, and (ii) a *key recipe*, which lists the secret keys of encrypted chunks in the backup for decrypting the encrypted chunks on the client side. Note that the key recipe is encrypted by the client's master key for protection. For a backup, the client uploads the key recipe along with the encrypted chunks to the cloud, while the cloud generates the file recipe based on the received encrypted chunks.

Deployment. EDRStore is applicable for an organization that plans to securely outsource the backup management for its clients to a remote cloud storage service. The organization can deploy a virtual machine or container instance in the cloud to perform data reduction on the outsourced data for storage efficiency. It can deploy the key server in a local server that is owned by the organization for ease of management, or through a semi-trusted third-party service that is independent of the cloud storage service [13].

3.3 Threat Model

We consider an honest-but-curious adversary that aims to infer the content of outsourced data without modifying the storage protocols. The adversary can compromise the cloud, one or multiple clients, and the key server in an attempt to infer the content of outsourced data. Specifically, the adversary can compromise the cloud and access the content and metadata (e.g., size) of each encrypted chunk stored in the cloud. Also, the adversary can compromise one or multiple clients and access the original data chunks and keys of the compromised clients. Furthermore, since EDRStore deploys a dedicated key server for key management as in server-aided MLE [13] (§3.2), the adversary can compromise the key server to infer chunk information during key generation; note that it has the same adversarial capability as described in the previous work [13]. In §5, we discuss how EDRStore protects the chunk content in such adversarial scenarios. Note that EDRStore is robust against chunk-inconsistency attacks (§4.3), so as to preserve data integrity.

One major goal of EDRStore is to enable delta compression on encrypted similar chunks (§4.1) to achieve the maximum possible storage savings, while mitigating the leakage of information (including the chunk content and the chunk length after local compression [1, 18, 36, 52]). Note that delta compression on encrypted similar chunks inevitably leaks information about the similarity of these chunks. In §5, we elaborate on the security limitations of EDRStore and argue that such limitations have limited practical impact.

Our threat model further makes the following assumptions. First, all communication channels in EDRStore are protected (e.g., by SSL/TLS) against eavesdropping and tampering. Second, EDRStore is robust against side-channel attacks by performing deduplication and delta compression in the cloud [29, 40, 48], so that a malicious client cannot feasibly infer the content of a victim client via the deduplication and delta compression patterns. Third, the key server can rate-limit key generation requests [13] to defend against online brute-force attacks and can also be extended with multiple key servers to address the single-point-of-attack [24, 71]. Fourth, due to the deterministic nature of encrypted deduplication (i.e., duplicate data chunks are always mapped to duplicate encrypted

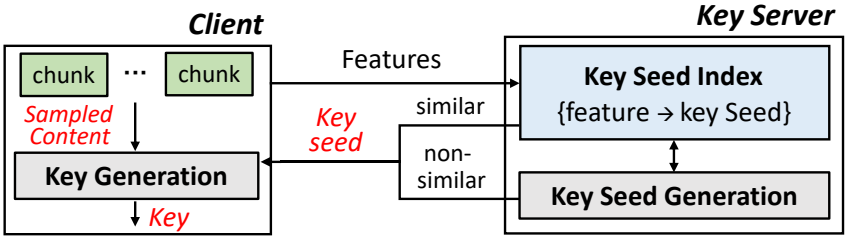


Fig. 2. Similarity-aware key management.

chunks), an adversary can launch frequency analysis and learn the occurrence frequencies of encrypted chunks [39]. We can mitigate such frequency leakage by encrypting duplicate data chunks with different keys subject to a storage blowup requirement [71]; we pose this extension as our future work. Finally, EDRStore can deploy existing defense mechanisms against traffic analysis [29] and data corruption [11, 33].

4 DETAILED DESIGN

We elaborate on three major design schemes of EDRStore (§4.1-§4.3) and describe its upload/download workflows (§4.4).

4.1 Similarity-aware Key Generation

Recall that server-aided MLE deterministically generates the same MLE key only for duplicate data chunks (§2.2). We extend server-aided MLE with *similarity-aware key generation*, which generates the same key for not only duplicate data chunks, but also similar (non-duplicate) data chunks. Figure 2 depicts the similarity-aware key generation approach. Our idea is to let the key server compare the features of data chunks to identify similar chunks and return the same *key seed* for each set of similar chunks. The client locally generates the key of each chunk based on the key seed, while the key server cannot learn the information of both the client's data chunk and the resulting keys (this is a major security requirement of server-aided MLE).

Specifically, for each data chunk M, a client derives a set of features $\{f_i\}$ (e.g., three features [72]) from the content of M and sends the features $\{f_i\}$ to the key server. The key server maintains a *key seed index*, a key-value store that tracks each feature and the corresponding key seed. It checks the key seed index based on $\{f_i\}$, and returns the key seed S that matches the most number of features in $\{f_i\}$ to the client. If none of $\{f_i\}$ is found in the key seed index (i.e., no existing chunk is similar to M, and M should be a base chunk (§2.1)), the key server generates a new key seed $S = \mathbf{H}(f_1||...||f_i||\theta)$ for M, where θ is the global secret of the key server [13] and $||$ is the concatenation operator. It adds the mapping of each f_i to S into the key seed index, and returns S.

The key server cannot learn the chunk content from a feature. First, a feature is a one-way hash that cannot be feasibly inverted to the chunk content. Second, a feature is a weak hash of the chunk content with many hash collisions, which make the key server difficult to identify a chunk from the many-to-one mapping of chunks to features [71].

One design challenge is how a client should generate the key from the returned key seed, while ensuring that the key server cannot learn the key from the corresponding key seed; note that if we directly derive a key based on the key seed, the key server can easily learn the key. We propose to generate the key of a chunk based on not only the corresponding key seed, but also the first P bytes of the original chunk content, where P is a configurable parameter. Our rationale is that similar chunks only have a small content difference that is unlikely to be sampled, and hence they

remain to have the same key with a high probability. Also, the key server cannot infer such keys without knowing the original chunk content. Specifically, upon receiving a key seed S, the client samples the first P bytes of the chunk M (denoted by $\overline{M}$), and computes the key $K = \mathbf{H}(S||\overline{M})$.

The value of P represents a trade-off between the degree of data confidentiality and the storage savings through the delta compression of similar chunks. A larger P means that more bytes are sampled from the original chunk content for key generation. It not only makes the key server more difficult to infer the key, but also makes similar chunks less likely to be encrypted by the same key (as any content difference in their first P bytes will lead to different keys). From the data confidentiality perspective, a leaked key only involves a smaller fraction of similar chunks in a larger P; in the extreme case, when P is the same as the chunk size, the encryption mode reduces to MLE. From the storage efficiency perspective, a larger P reduces the likelihood that two similar chunks are encrypted by the same key and their encrypted outputs cannot be delta-compressed. We empirically study the trade-off in §7.2. By default, we choose $P = 32$ bytes.

Note that the amount of network traffic between a client and the key server during key generation is limited. Based on our default configuration, each data chunk has an average size of 4 KiB. A client generates three features (8 bytes each) and uploads $3 \times 8 = 24$ bytes per data chunk to the key server, which returns a 32-byte key seed. The total network traffic of the one-round interaction takes only $(24 + 32)$ bytes / 4 KiB = 1.4% of logical data.

4.2 Two-phase Encryption

To support deduplication and delta compression (performed in the cloud) on encrypted chunks, EDRStore builds on the block cipher [37] to encrypt each data chunk independently as in MLE. However, the encryption approach in MLE will pose security risks to EDRStore. To this end, we propose *two-phase encryption* to address the security issue.

Encryption in MLE. We first explain how encryption works in MLE and why it is inappropriate for EDRStore. MLE encrypts each data chunk in counter (CTR) mode with a fixed initialization vector (IV) [14] and preserves the deduplication capability for duplicate data chunks after encryption. Specifically, each data chunk M (e.g., 4 KiB long) can have multiple *blocks*, the basic units in block cipher (e.g., 16 bytes long in AES encryption). MLE initializes both the IV and counter as zero, while the counter is incremented by one across each block. It generates a *mask* based on the key, IV, and the counter via the encryption function, such that the mask has the same length as M. Finally, it combines M and the mask via a bitwise XOR operation to generate the encrypted chunk. By deriving the same key for duplicate data chunks, MLE encrypts them into duplicate encrypted chunks to preserve the deduplication capability.

To support delta compression, we may naïvely ensure that the same key (and hence the same mask under the same IV and counter initializations) is used for encrypting similar data chunks via similarity-aware key generation (§4.1). As similar data chunks have large fractions of duplicate blocks, the resulting encrypted chunks also have large fractions of duplicate blocks that can be removed by delta compression. However, this naïve approach poses a security hole due to the reuse of the same key and IV for distinct data chunks. Specifically, an adversary can remove the mask with a bitwise XOR operation of the corresponding encrypted chunks and learn the XOR output of the distinct data chunks *even without knowing the key*.

For example, consider two similar data chunks $M = b_1||b_2$ and $\hat{M} = b_1||\hat{b_2}$, where b_1, b_2, and $\hat{b_2}$ are distinct data blocks and $||$ is the concatenation operator (i.e., M and $\hat{M}$ share the same first data block but differ in their second blocks). The naïve approach generates two masks e_1 and e_2 based on the key, IV, and counter, and encrypts M and $\hat{M}$ into the encrypted chunks $C = c_1||c_2$ and $\hat{C} = c_1||\hat{c_2}$, respectively, where $c_1 = b_1 \oplus e_1$, $c_2 = b_2 \oplus e_2$, and $\hat{c_2} = \hat{b_2} \oplus e_2$, where $\oplus$ is the bitwise

XOR operator. Suppose that an adversary has access to C and $\hat{C}$. It knows that both C and $\hat{C}$ have the same first data block, but does not know the content of the first data block; however, it not only knows that their second data blocks are distinct, but can also infer the relationships of data patterns via $c_2 \oplus \hat{c_2} = b_2 \oplus \hat{b_2}$. Note that the security hole does not exist in MLE, since the same key is only used for duplicate (rather than distinct) data chunks.

Our solution. Two-phase encryption augments the CTR mode with an additional encryption layer in electronic codebook (ECB) mode, in which each block is encrypted independently. Suppose that the encryption function is secure. Then two original blocks with any different bit can be encrypted into statistically different encrypted blocks (i.e., the block-level diffusion property [58]). Our security insight is that for the non-duplicate encrypted blocks obtained from the CTR mode, we further encrypt them in ECB mode into statistically distinct encrypted blocks, so that the bitwise XOR operation of the encrypted blocks can no longer leak the content patterns of the original data blocks.

The details of two-phase encryption are elaborated as follows. To encrypt a data chunk M, the client first partitions M into a sequence of data blocks $b_1, b_2, \ldots, b_m$, where m is the total number of blocks in M. In the first phase, for each data block b_i ($1 \leq i \leq m$), the client computes the mask $e_i = \mathbf{E}(K, (IV||i))$ and encrypts b_i in CTR mode into $c_i = e_i \oplus b_i$, where $\mathbf{E}(\cdot)$ is the encryption function with the key and block as inputs, K is the key of M generated in §4.1, $||$ is the concatenation operator, i is the counter in CTR mode, and $\oplus$ is the bit-wise XOR operator. In the second phase, the client encrypts c_i in ECB mode into $c_i' = \mathbf{E}(K, c_i)$, and forms the whole encrypted chunk $C = c_1'||c_2'|| \ldots ||c_m'$. To decrypt the encrypted chunk, the client computes each $c_i = \mathbf{D}(K, c_i')$ (where $\mathbf{D}$ is the decryption function with the key and block as inputs), generates the mask e_i as above, and recovers the data block $b_i = c_i \oplus e_i$, and hence M.

Two-phase encryption has several practical properties. First, although encryption in ECB mode is notoriously known to leak the occurrence frequency of a data block, two-phase encryption does not have the frequency leakage issue as each block is protected by the incremental counter in CTR mode [26]. Second, it preserves the effectiveness of delta compression, since the duplicate blocks at the same positions of similar data chunks are mapped into duplicate encrypted blocks. Finally, it is parallelizable at the block level, as both CTR and ECB modes encrypt blocks independently. Our evaluation shows that two-phase encryption is not a performance bottleneck of EDRStore (Exp#7 in §7.4). In §5, we analyze the security guarantees of two-phase encryption.

4.3 Selective Local Compression

Recall that EDRStore performs local compression before delta compression (§3.2). However, applying delta compression directly to locally compressed chunks brings limited storage savings, since local compression encodes similar chunks into distinct compressed chunks with high entropy (§2.1).

Selective local compression aims to maximize the possible storage savings via both delta compression and local compression under encryption. Its idea is that each client selectively performs local compression on a subset of data chunks that are unlikely to gain significant storage savings via delta compression, while leaving the remaining data chunks uncompressed so that they can be delta-compressed in the cloud. Specifically, the client uploads *encrypted uncompressed chunks* (i.e., the encrypted output of uncompressed data chunks) and *encrypted compressed chunks* (i.e., the encrypted output of compressed data chunks). The cloud performs delta compression on each set of encrypted chunks. For encrypted uncompressed chunks, delta compression should bring significant storage savings; for encrypted compressed chunks, delta compression can also bring slight storage savings. Note that content similarity across encrypted chunks is preserved due to similarity-aware key generation (§4.1) and two-phase encryption (§4.2).

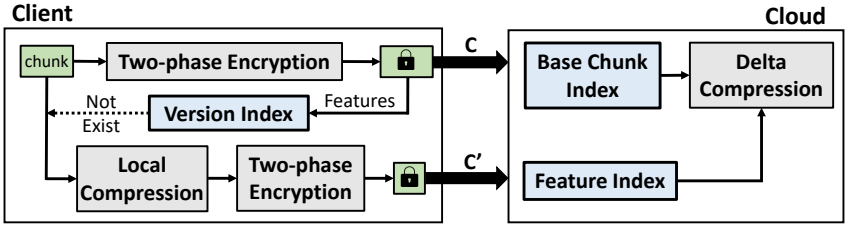


Fig. 3. Selective local compression.

Design requirements. Our selective local compression design should satisfy the following requirements. First, it should offload delta compression from a client to the cloud, so that a client does not need to keep a myriad of base chunks for delta compression and any client crash does not cause data loss (§3.1). Second, in some cases (see below), a client needs to upload a pair of encrypted compressed and uncompressed chunks, so our design should be secure against a malicious client that uploads inconsistent chunks (i.e., the pair of encrypted compressed and uncompressed chunks is derived from different data chunks) to erase other clients' data by abusing deduplication [13]. Finally, while the cloud keeps encrypted uncompressed chunks as base chunks for delta compression, our design should have policies to limit the storage of such base chunks in the cloud.

Design details. To address the above design requirements, our idea is that the cloud maintains the base chunks for the delta compression of encrypted uncompressed chunks on a per-client basis, based on the assumption that content differences are often derived from different versions of a backup series from the same client [66, 67, 77, 78]. Also, a client tracks the features of the encrypted uncompressed chunks. Such tracked features provide "hints" for the client to determine whether a data chunk needs to be locally compressed. Note that EDRStore still performs cross-client deduplication, even though it performs delta compression on uncompressed chunks for each client.

Figure 3 depicts the idea of selective local compression. Each client manages a *version index*, a key-value store that tracks the features of encrypted uncompressed chunks uploaded by the client, so as to decide whether an encrypted uncompressed chunk uploaded by the client can be delta-compressed with respect to any similar encrypted uncompressed chunks in the cloud, or the client should first perform local compression on the data chunks. Each feature is mapped to the version number of the most recent backup that contains the feature, so that the cloud can manage the base chunks (see version-aware chunk management below). Note that the version index is robust to client crashes, since it can be recovered from the features of the base chunks stored in the cloud. Also, the cloud manages a per-client *base chunk index*, a key-value store that tracks all features and their corresponding encrypted uncompressed chunks (i.e., base chunks) uploaded by a client, so as to perform delta compression on any subsequent chunks uploaded by the client. Finally, the cloud manages a *feature index*, a key-value store that tracks the features of encrypted compressed chunks and their fingerprints, so as to perform delta compression for encrypted compressed chunks.

Specifically, for each data chunk M (which is originally uncompressed), a client first encrypts it into an encrypted uncompressed chunk C via two-phase encryption (§4.2). It also extracts a set of features $\{g_i\}$ from C. If any feature g_i exists in the version index, the client sends (C, `tag`) to the cloud for deduplication and delta compression; the `tag` is used in dual-fingerprint deduplication (see below). Otherwise, if none of the features in $\{g_i\}$ exists, the client locally compresses M, generates the dummy padding bytes B based on the fingerprint of M and encrypts $M||B$ (where $||$ is the concatenation operator) into an encrypted compressed chunk C'; note that C' generally has a

smaller size than C. Padding is necessary to hide the lengths of compressed chunks (§2.2), while our padding approach ensures that duplicate chunks have the same padding data to preserve the deduplication capability. Note that even though we pad deterministic data to duplicate chunks, since the padding data is randomly generated based on the fingerprint of each data chunk via a hash function (§6), the security is not compromised as the cloud cannot infer the contents and length of each original compressed chunk.

The client then sends (C, C') to the cloud. It further adds all features $\{g_i\}$ into the version index, such that C may later serve as a base chunk for later uploaded similar encrypted uncompressed chunks, while C' is the chunk that is persistently stored in the cloud. We choose to upload both C and C' to trade small network bandwidth overhead for significant storage savings. Specifically, the uploads of both C and C' occur only when C does not have any similar chunk in previous backups. This is expected to be uncommon, as adjacent backups tend to have only small changes due to locality [65, 75]. On the other hand, C can serve as a base chunk to delta-compress many following chunks for significant storage savings. Also, when C is expired and evicted from the base chunk index (see below), C' serves as the persistent copy of the corresponding chunk and preserves storage efficiency (note that C' has a smaller size than C since it has been locally compressed). Our evaluation shows that the additional network traffic remains limited (Exp#5 in §7.3).

The cloud now receives either $(C, \texttt{tag})$ or (C, C') from a client. It first performs deduplication on C. If C is a duplicate chunk, the cloud stores a reference to its physical copy (and discards C' if any). Otherwise, if C is a non-duplicate chunk, there are two cases. In the first case, if the cloud receives $(C, \texttt{tag})$, it selects the most similar base chunk in the base chunk index based on the set of features $\{g_i\}$ of C and performs delta compression on C with respect to the selected base chunk. It stores the delta chunk of C in the storage pool. In the second case, if the cloud receives (C, C'), it adds C (which is treated as a base chunk) and its features to the base chunk index. Also, the cloud checks the feature index to see if C' can be delta-compressed with respect to any currently stored encrypted compressed chunk. It stores either C' or its delta chunk in the storage pool.

Defense against chunk-inconsistency attacks. A malicious client may launch *chunk-inconsistency attacks*, by generating inconsistent encrypted compressed and uncompressed chunks to compromise the uploads from other clients. Specifically, it uploads a valid encrypted uncompressed chunk C and a forged encrypted compressed chunk C'_f. Suppose that C is a non-duplicate chunk. The cloud refers the fingerprint of C to the forged C'_f (or its delta chunk). Later, a victim client uploads (C, C'), where C is now a duplicate chunk and C' is a valid encrypted compressed chunk corresponding to C. The cloud will discard C' by deduplication and wrongly refer C to the forged physical copy C'_f. When the victim client downloads C, it will receive the forged C'_f.

To defend against chunk-inconsistency attacks, we propose *dual-fingerprint deduplication*, which performs deduplication based on both the fingerprints of C and C'. Thus, the correct pair (C, C') uploaded by a benign client is considered to be different from the forged pair (C, C'_f). Note that dual-fingerprint deduplication does not incur extra storage overhead if the uploaded encrypted chunks are consistent.

We elaborate on dual-fingerprint deduplication as follows. Each client sets $\texttt{tag}$ as $\mathbf{H}(C')$ and uploads $(C, \mathbf{H}(C'))$ if there exists any feature in the client's version index. When the cloud receives $(C, \mathbf{H}(C'))$ or (C, C'), it derives the dual fingerprint, defined as $\mathbf{H}(\mathbf{H}(C)||\mathbf{H}(C'))$, and treats C as a duplicate chunk only when the dual fingerprint is new. It also manages the fingerprint index that tracks the dual fingerprints of the existing encrypted chunks.

Version-aware chunk management. As the cloud stores more backups, the base chunk index accumulates a large number of base chunks and grows significantly in size. While storing more base chunks in the base chunk index facilitates delta compression, it also incurs high storage overhead,

which may negate the overall storage savings (§7.3). Thus, our goal is to keep a sufficiently small base chunk index without compromising the effectiveness of delta compression.

EDRStore introduces *version-aware chunk management* to keep only the base chunks from the recent backup versions. Our rationale is that a backup version is often derived from the modifications from the recent backup versions, so adjacent backup versions tend to have more similar chunks [66, 67, 77, 78]. Specifically, EDRStore keeps only the base chunks of the recent N ($N = 1$ by default) backups in the base chunk index and evicts a base chunk if it has not appeared or been used in the recent N versions. Each client manages features in the version index. For each feature, if it exists in the version index, the client updates the version number of the feature to the current version; otherwise (i.e., the feature is new), the client inserts the feature into the version index with the current version number. The client regularly checks the whole version index (say, at the end of each backup upload) and deletes the entries whose version numbers differ from the current version by more than N. It sends the expired features in an eviction list to the cloud, which then evicts the base chunks with the expired features. Note that even if the base chunk has been evicted from the version index, the original chunk can still be restored since its encrypted compressed chunk is stored in the storage pool.

4.4 Putting It All Together

We describe EDRStore's upload and download workflows.

Upload. Suppose that a client uploads a backup. It divides the backup into data chunks and computes their fingerprints and features. For each data chunk, it obtains a key seed from the key server and generates the corresponding key via similarity-aware key generation (§4.1). It encrypts each data chunk, either in locally compressed form (which is further padded with dummy bytes before encryption) or in uncompressed form, via two-phase encryption (§4.2) and selective local compression (§4.3). It uploads encrypted chunks (derived from compressed or uncompressed data chunks) and tags to the cloud. Also, the client creates a key recipe for the backup, encrypts the key recipe with its master key, and uploads the key recipe to the cloud. Finally, the cloud performs deduplication and delta compression for the encrypted chunks (§4.3), and creates a file recipe on the encrypted chunks for the backup.

Download. Suppose that a client downloads a backup. The cloud first retrieves the encrypted chunks based on the file recipe; if an encrypted chunk, say C_d, has been delta compressed (i.e., its delta chunk is stored and its fingerprint index records the fingerprint of its base chunk), the cloud retrieves the corresponding base chunk to reconstruct C_d. There are three cases to consider for reconstructing C_d.

- **Case 1: C_d is delta-compressed with respect to an encrypted uncompressed chunk, say C, that remains in the base chunk index.** The cloud reads C from the base chunk index, performs delta decompression to reconstruct C_d, and sends C_d to the client.
- **Case 2: C_d is delta-compressed with respect to an encrypted uncompressed chunk, say C, that has been evicted from the base chunk index.** The cloud reads the encrypted compressed chunk, say C', that corresponds to the base chunk from the storage pool and sends C' to the client. The client first decrypts and local-decompresses C', and then re-encrypts the result into an encrypted uncompressed chunk C. It then performs delta decompression to reconstruct C_d.
- **Case 3: C_d is delta-compressed with respect to an encrypted compressed chunk C'.** The cloud reads C' from the storage pool, performs delta decompression to reconstruct C_d, and sends C_d to the client.

In short, the cloud returns the encrypted chunks and the encrypted key recipe to the client. The client first decrypts the key recipe based on its master key, reconstructs the encrypted chunks (i.e.,

Case 2 above) if necessary, and then decrypts each encrypted chunk based on the secret key in the key recipe. It removes the padded bytes and local-decompresses the compressed data chunks. It assembles all original uncompressed data chunks into the backup.

5 SECURITY IMPLICATIONS

5.1 Guarantees

We discuss the security guarantees achievable by EDRStore against an adversary based on our threat model (§3.3). We consider three adversarial scenarios.

- **Case 1: An adversary has access to only the cloud.** The adversary can access the encrypted key recipes and encrypted chunks, but it cannot learn any key from an encrypted key recipe (protected by the client's master key). Also, from each encrypted chunk, it cannot learn the original data due to two-phase encryption (§4.2) and the actual length of the compressed data chunk due to padding (§4.3). Furthermore, it cannot feasibly launch chunk-inconsistency attacks due to dual-fingerprint deduplication (§4.3).
- **Case 2: An adversary has access to both the cloud and some clients.** In addition to the cloud, the adversary also has access to some clients and further obtains their data chunks and keys. However, it cannot learn any non-similar data chunks that are encrypted by different keys.
- **Case 3: An adversary has access to the key server.** An adversary that has access to the key server additionally learns the global secret as well as the key seed for each set of similar chunks. However, it cannot learn *unpredictable* chunks whose content cannot be enumerated [13], since each key is generated based on a sampled fraction of the chunk content (§4.1). Also, it cannot readily identify the key generation for a data chunk, since each feature is derived based on a weak hash function (i.e., hash collisions are allowed for different data chunks) hash function (§4.1).

5.2 Limitations

We discuss the security limitations of EDRStore.

Leakage of duplicate blocks. Since EDRStore keeps the duplicate blocks at the same positions of similar data chunks under two-phase encryption (§4.2), the adversary can identify such duplicate blocks from similar data chunks. Such leakage is inevitable in order for delta compression to be viable for encrypted similar chunks. Prior studies [27, 28] also identify such leakage in disk encryption (e.g., encrypting the values stored in the same sector), yet the practical damage remains an open question.

Leakage of compression relationships. When a client uploads a pair of encrypted chunks derived from the compressed and uncompressed versions of the same data chunk (§4.3), the adversary can learn the underlying compression relationship between the encrypted chunks. We are unaware of any attack that exploits such relationships to learn additional information, especially when both chunks are encrypted.

Leakage of similar chunks under key compromise. EDRStore uses the same key for the encryption of similar data chunks and allows delta compression among such encrypted similar chunks (§4.1). It incurs the information leakage of similar data chunks if the key is compromised. Such information leakage is inevitable if we want to support delta compression on encrypted similar chunks. We can mitigate the leakage by trading the storage savings of delta compression for security. Recall that each client generates the key of each chunk M based on the first P bytes of the original chunk content (§4.1). We can configure P to balance the trade-off between data confidentiality and storage savings. We evaluate the impact of P in §7.2.

6 IMPLEMENTATION

We prototyped EDRStore in C++ on Linux. We also implemented cryptographic operations using OpenSSL-1.1.1 [50]. Our prototype contains about 8.7 K LoC. We highlight several implementation details of EDRStore as follows.

Index structures. EDRStore has several index structures: the key seed index in the key server (§4.1), the version index in each client (§4.3), and the fingerprint index, feature index, and base chunk index in the cloud (§4.3). All indexes except the base chunk index are implemented as in-memory hash-table using `std::unordered_map` in C++, while the base chunk index is implemented as a persistent key-value store using RocksDB [45].

Note that even though a system crash causes the loss of in-memory indexes (i.e., the key seed index, version index, fingerprint index, and feature index), it does not cause any data loss. Specifically, the version index can be recovered from the features of the base chunks in the cloud (§4.3), while the fingerprint index and feature index can be recovered from the encrypted chunks stored in the cloud. The loss of the key seed index also does not cause any data loss since a client can still recover the data chunks based on the keys in the key recipes (§3.2), yet the key seed index cannot be rebuilt since the features are derived from the original data chunks (§4.1) that are inaccessible by the key server. The consequence of not being able to rebuild the key seed index is that the same key seeds cannot be regenerated for similar data chunks, and hence we cannot allow the data reduction of encrypted chunks. We can extend EDRStore to make periodic snapshots of all in-memory indexes into persistent storage, so that the in-memory indexes can be readily rebuilt from the snapshots. We expect that such snapshot generation incurs limited performance and storage overhead since the indexes are generally small (§7.3).

Client. A client generates data chunks using FastCDC [69] for content-defined chunking, with the minimum, average, and maximum chunk sizes being 4 KiB, 8 KiB, and 16 KiB, respectively. For each data chunk, it computes the fingerprint using SHA-256, as well as three features (8 bytes each) using Finesse [72]. It implements two-phase encryption (§4.2) using AES-256-CTR and then AES-256-ECB. In selective local compression, the client uses Zstandard [20] for local compression. It also generates the padding data for each compressed chunk (§4.3) using the chunk fingerprint as the random seed of the pseudorandom number generator `std::default_random_engine` in C++, with a padding length of 0-255 bytes. To allow the removal of padding data during decompression, each compressed chunk has a 4-byte field that records the length of the original compressed chunk without padding, such that the length is encrypted along with the chunk data via two-phase encryption. After chunk decryption, EDRStore reads the length to remove the padding data, followed by local decompression to recover the original data chunk.

Key server. The key server maintains a key seed index to track the feature of each data chunk, and a 32-byte global secret θ to generate a key seed for each base chunk (§4.1). Given the three features $\{f_i\}_{i=1}^{3}$ of a base chunk, it computes the key seed as $S = \mathbf{H}(f_1||f_2||f_3||\theta)$, where $\mathbf{H}(\cdot)$ is SHA-256.

Cloud. The cloud performs delta compression using Xdelta [44]. It packs both delta chunks and encrypted compressed chunks in large storage units, called *containers* [41], for persistent storage; we now set the container size as 4 MiB.

Optimization. We improve the performance of our prototype using well-known optimization approaches. For example, each client parallelizes chunking, key generation, local compression, encryption, and uploads in multiple threads, while the key server and the cloud serve the requests from clients in different threads. The cloud maintains an in-memory least-recently-used cache (of size 2 GiB currently) to keep the containers that are recently accessed for delta compression (on encrypted compressed chunks) and downloads.

Dataset	Logical size	Dedup	Delta	Local	Plain
TF	23.0 GiB	1.9	4.2	2.9	22.5
DOCKER	37.8 GiB	4.0	2.2	2.0	17.9
GCC	40.0 GiB	1.9	2.9	3.0	17.0
CHROM	51.9 GiB	3.3	3.6	2.3	26.5
WEB	277.1 GiB	7.6	9.4	1.6	117.1
LINUX	335.1 GiB	2.3	8.9	2.3	47.0

Table 1. Logical sizes and data reduction ratios of plaintext datasets.

7 EVALUATION

We conduct trace-driven evaluation on EDRStore based on real-world datasets (§7.1). We evaluate EDRStore in three aspects: the space-confidentiality trade-off (§7.2), the storage usage (§7.3), and the upload and download performance in networked environments (§7.4). The results in §7.2 and §7.3 are based on trace analysis, while those in §7.4 are obtained from our testbeds.

7.1 Datasets

We present the results for six real-world datasets: (i) *TF*, with 159 versions of Tensorflow source code (v0.6.0 to v158.2.9.0) [6]; (ii) *DOCKER*, with 108 versions of Docker snapshots of Cassandra (v2 to v4.0.5) [61] downloaded from Docker Hub [3]; (iii) *GCC*, with 113 versions of GCC source code (v2.95.0 to v11.2.0) [4]; (iv) *CHROM*, with 100 versions of Chromium source code (v3.0 to v22.0) [2]; (v) *WEB*, with 103 versions of website backups of `news.sina.com`, captured from June to September in 2016 (this dataset is also used in previous work [72, 78]); and (vi) *LINUX*, with 489 versions of Linux kernel source code (v3.0 to v5.16) [5]. Our datasets cover various types of workloads, including source code (TF, GCC, CHROM, and LINUX), binary snapshots (DOCKER), and website backups (WEB). We treat each version of datasets as a backup. For each dataset, we decompress the versions and pack them into a *tar* package for evaluation.

We measure the *logical size* (i.e., the raw data size before data reduction) and the *physical size* (i.e., the actual data size in storage after data reduction). Table 1 shows the dataset statistics when the data chunks are stored in plaintext form without encryption based on the data reduction workflow in [59] (abbrv. *Plain*). For each dataset, we report the logical size and the *data reduction ratios* (measured by dividing the logical size by the physical size) of deduplication on the raw data, delta compression on the post-deduplicated data, and local compression on the post-deduplicated and delta-compressed data. We also report the total data reduction ratio of Plain by multiplying the respective data reduction ratios from each data reduction technique (note that the final ratio may slightly deviate due to rounding errors).

7.2 Space-Confidentiality Trade-off

Exp#1 (Impact of P on storage savings and data confidentiality). We first study the impact of P (§4.1) on the trade-off between storage savings and data confidentiality due to the encryption of similar chunks with the same key. We vary P from zero bytes to the chunk length (labeled as "max"); note that when P is the chunk length, EDRStore applies MLE to locally compressed chunks.

To measure data confidentiality, we consider an adversarial scenario where a number of keys are compromised and the corresponding chunks encrypted by the compromised keys are leaked. We define the *leak frequency* of a key as the number of logical chunks encrypted by the key. Suppose in the worst case that the top-k keys with the highest leak frequencies are compromised. We measure the *leak ratio* as the ratio between the total leak frequencies of the top-k compromised keys and the total number of logical chunks. To measure storage savings, we measure both the total reduction

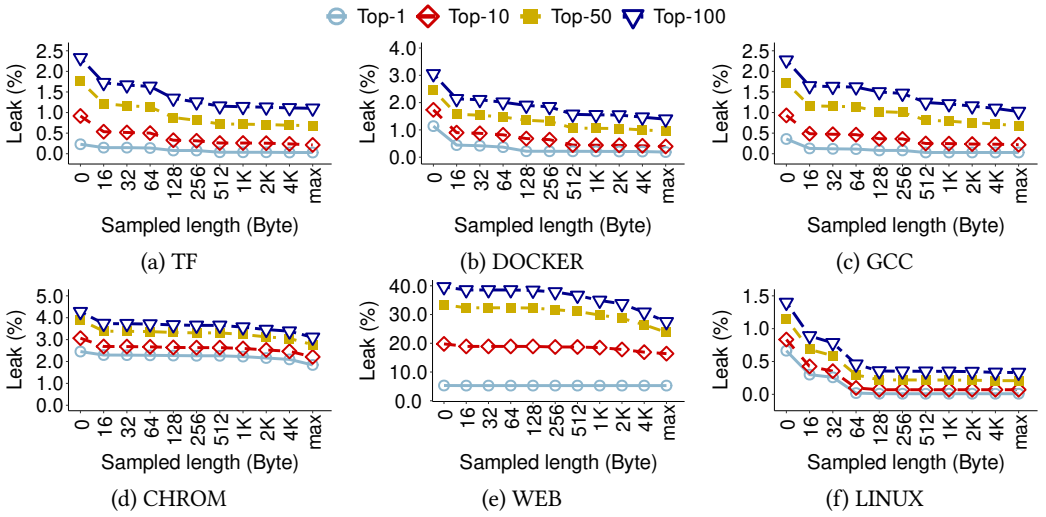


Fig. 4. (Exp#1) Impact of P on leak ratio.

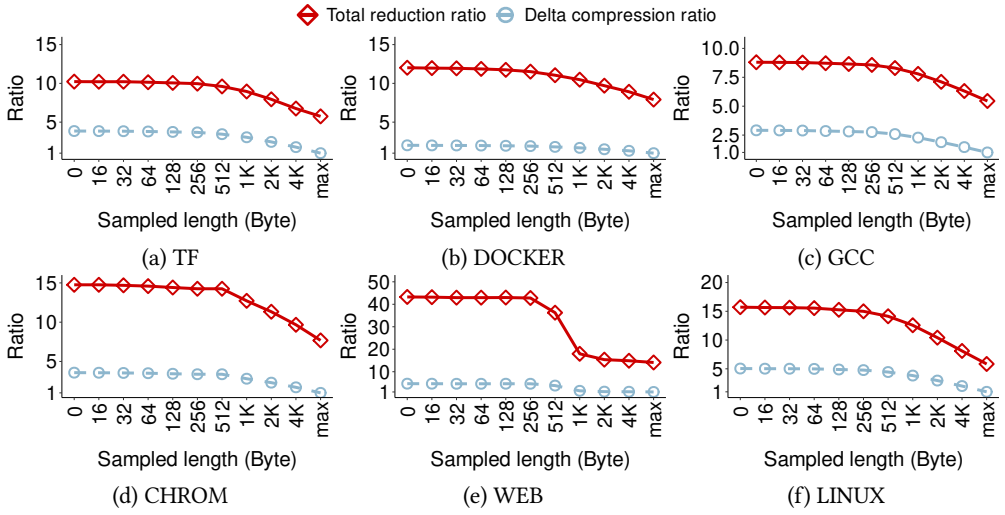
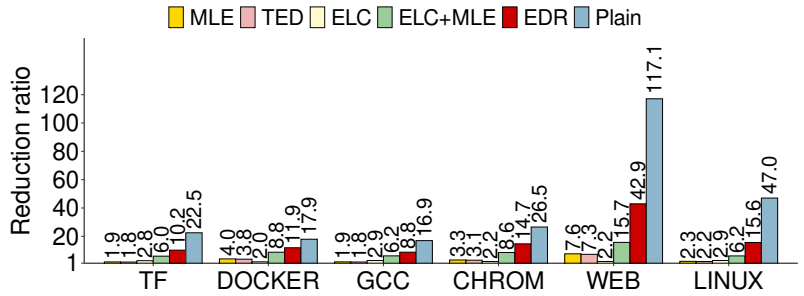


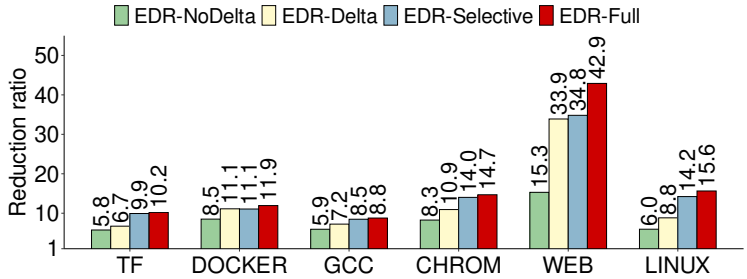
Fig. 5. (Exp#1) Impact of P on total data reduction ratio and delta compression ratio.

ratio (i.e., the data reduction ratio when all data reduction techniques are enabled) and the delta compression ratio (i.e., the data reduction ratio caused by delta compression only).

Figure 4 shows the leak ratio for different values of k versus P. The leak ratio decreases as P increases. It first decreases sharply from $P = 0$ to $P = 16$ bytes, and its decrease becomes less significant as P further increases. Figure 5 shows that the total reduction ratio and the delta compression ratio decrease as P increases (note that the delta compression ratio becomes one for "max"). For WEB, the leak ratios are high in general (even for "max", which reduces to MLE) as it has a large volume of duplicate chunks (Table 1). Such leakage for duplicate chunks can be mitigated by encrypting them with different keys [71] (§3.3). We do not claim that the "best" P exists, as P represents different trade-offs between storage savings and data confidentiality. In the following, we choose $P = 32$ bytes as our default.



(a) Overall data reduction ratio



(b) Breakdown of data reduction ratio

Fig. 6. (Exp#2) Analysis of data reduction ratio.

7.3 Data Reduction Analysis

Exp#2 (Analysis of data reduction ratio). We evaluate the data reduction ratio of EDRStore. Note that our analysis also includes the actual sizes of the data chunks and the base chunk index, both of which are stored in the storage pool.

We compare EDRStore with the following baselines based on §2.2: (i) *Plain*, which is reported in Table 1 and serves as an ideal (but insecure) baseline with the maximum possible data reduction. (ii) *MLE* [14], which performs encrypted deduplication; (iii) *TED* [71], which performs tunable encrypted deduplication to trade storage efficiency for data confidentiality; (iv) *ELC (encrypted local compression)*, which first performs local compression without padding on data chunks and encrypts the compressed data chunks; and (v) *ELC+MLE*, which applies MLE to locally compressed chunks and performs deduplication after ELC. For TED, we set the storage blowup factor as 1.05 (default in [71]), meaning that it stores 5% more data than MLE; for ELC+MLE, we disable padding to make ELC and MLE compatible (§2.2).

Figure 6(a) shows the results. EDRStore achieves the highest data reduction ratio among MLE, TED, ELC, and ELC+MLE in all datasets. For example, in LINUX, the data reduction ratio of EDRStore is 6.8×, 7.1×, 5.4×, and 2.5× compared with those of MLE, TED, ELC, and ELC+MLE, respectively. Overall, the data reduction ratio of EDRStore is up to 6.8×, 7.1×, 19.7×, and 2.7× compared with those of MLE, TED, ELC, and ELC+MLE, respectively, in all datasets. Note that EDRStore only has 33.2-66.5% data reduction ratio compared with Plain due to various reasons. First, EDRStore performs delta compression on encrypted chunks at coarse-grained block-level granularity (§4.2), while Plain can remove byte-level redundancy. Second, EDRStore includes the base chunk index, which is not needed in Plain. Finally, EDRStore only keeps the base chunks from recent versions, so a chunk cannot be delta-compressed with a similar base chunk from old versions.

We next break down how the design choices of EDRStore contribute to data reduction. We focus on selective local compression (on the client side) and delta compression (on the cloud side), and consider four variants of EDRStore: (i) *EDR-NoDelta*, which applies local compression to all data chunks (i.e., no selective local compression) and performs deduplication only (i.e., no delta compression); (ii) *EDR-Delta*, which applies local compression to all data chunks, and performs deduplication and delta compression on the encrypted compressed chunks; (iii) *EDR-Selective*, which performs selective local compression, deduplication, and delta compression on the encrypted uncompressed chunks, but disables delta compression on the encrypted compressed chunks; and (iv) *EDR-Full*, our complete EDRStore design that performs selective local compression, deduplication, and delta compression on all encrypted chunks. Note that both EDR-NoDelta and EDR-Delta do not need the base chunk index, as they do not perform selective local compression (EDR-Delta only keeps a feature index in the cloud for similarity detection).

Figure 6(b) shows the results. EDR-Delta increases the data reduction ratio of EDR-NoDelta by 17.0-121.4% due to delta compression over encrypted compressed chunks. EDR-Selective increases the data reduction ratio over EDR-Delta (by up to 61.2% in LINUX), with the only exception in DOCKER. The reason is that EDR-Selective keeps the base chunk index, but not in EDR-Delta. When a dataset has limited similarity (e.g., DOCKER has the lowest data reduction ratio from delta compression), the base chunk index can negate the storage savings from data reduction. EDR-Full increases the data reduction ratio by 2.9-23.4% over EDR-Selective, and has the highest data reduction ratio.

Exp#3 (Index overhead). We evaluate the overhead of index structures in EDRStore. Recall that EDRStore has five index structures (i.e., the version index in each client, the key seed index in the key server, as well as the fingerprint index, feature index, and base chunk index in the cloud). We report the fractions of the size of each index structure over the logical size and over the physical size.

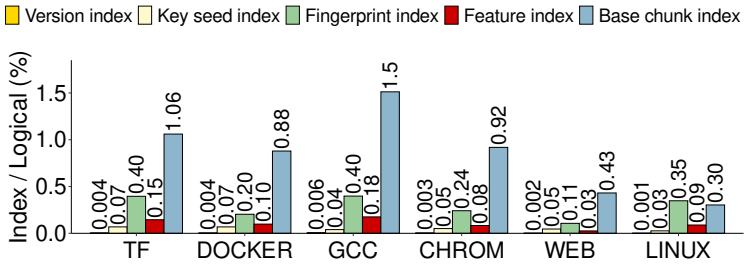
Figure 7 shows the results. For the in-memory indexes (i.e., the version index, key seed index, fingerprint index, and feature index), the major overhead mainly comes from the fingerprint index and the feature index, with up to 5.5% and 1.5% of physical size, respectively. We can employ existing memory-efficient indexing approaches, such as locality-preserved caching [75] for fingerprints and the stream-informed approach [59] for features, to manage small indexes in memory, while storing all fingerprints and features on disk.

The base chunk index has a much larger size (with up to 1.5% of logical size and up to 18.5% of physical size), as it stores the encrypted uncompressed chunks in the storage pool. Nevertheless, EDRStore still achieves higher data reduction than the encrypted storage baselines (see Exp#2).

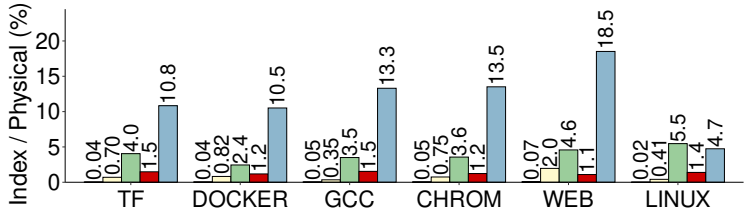
Exp#4 (Impact of N on version-aware chunk management). We evaluate the impact of N on version-aware chunk management, where N denotes the number of recent versions whose encrypted uncompressed chunks are stored in the base chunk index (§4.3). We compare the data reduction ratios with and without including the base chunk index in the physical size calculation, so as to understand the overhead of the base chunk index for different values of N.

Figure 8 shows the data reduction ratios versus N for different datasets. Without including the base chunk index in the physical size, the data reduction ratio increases with N, as it increases the likelihood of detecting similar chunks. However, with the base chunk index, the actual data reduction ratio may increase with N (e.g., in LINUX) or decrease for a large N (e.g., in GCC and CHROM); the latter implies that the storage overhead of the base chunk index negates the overall storage savings. How to find the best N for the maximum data reduction ratio is our future work.

Exp#5 (Network traffic). We evaluate the amount of network traffic being transferred when a client uploads backup snapshots to the cloud. Recall that EDRStore sends either the uncompressed



(a) Fraction of index size over logical size



(b) Fraction of index size over physical size

Fig. 7. (Exp#3) Index overhead.

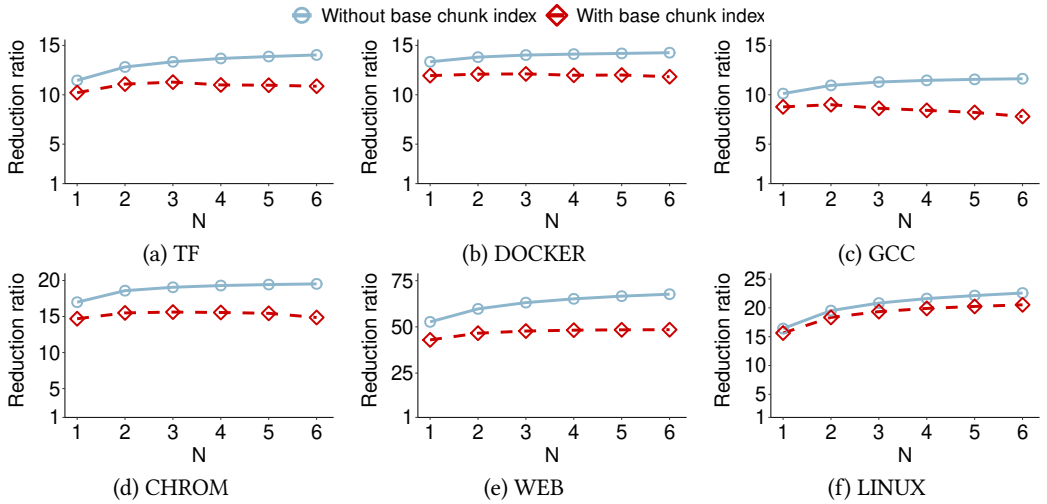


Fig. 8. (Exp#4) Impact of N on version-aware chunk management.

chunk with its fingerprint (C, tag), or both the uncompressed and locally compressed chunks (C, C') to the cloud (4.3). Here, we mainly compare EDRStore with Plain, which sends all data chunks in the uncompressed format to the cloud. The extra network traffic of EDRStore mainly comes from the fingerprints (tag) and the compressed data chunks (C'). Note that we do not compare EDRStore with MLE, ELC, and ELC+MLE, as they do not consider delta compression.

Figure 9 shows the results. EDRStore incurs limited extra network traffic compared with Plain. Specifically, EDRStore has up to 16.0% more network traffic than Plain (in DOCKER), and the additional traffic is within 10.0% for the other five datasets. Such additional network traffic has

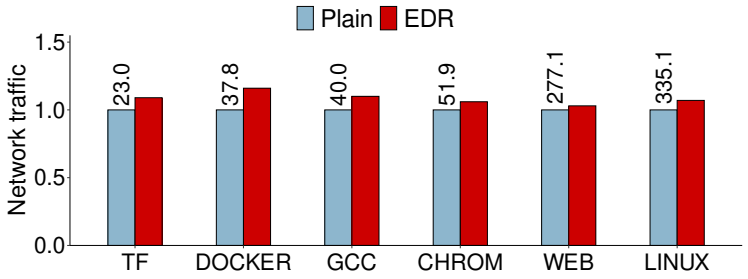


Fig. 9. (Exp#5) Network traffic. All results are normalized by Plain's network traffic in GiB (numbered atop the bars).

negligible impact on the upload performance in both the local cluster (where data transmission is not the bottleneck) and even the real-cloud testbeds (§7.4).

7.4 Performance Analysis

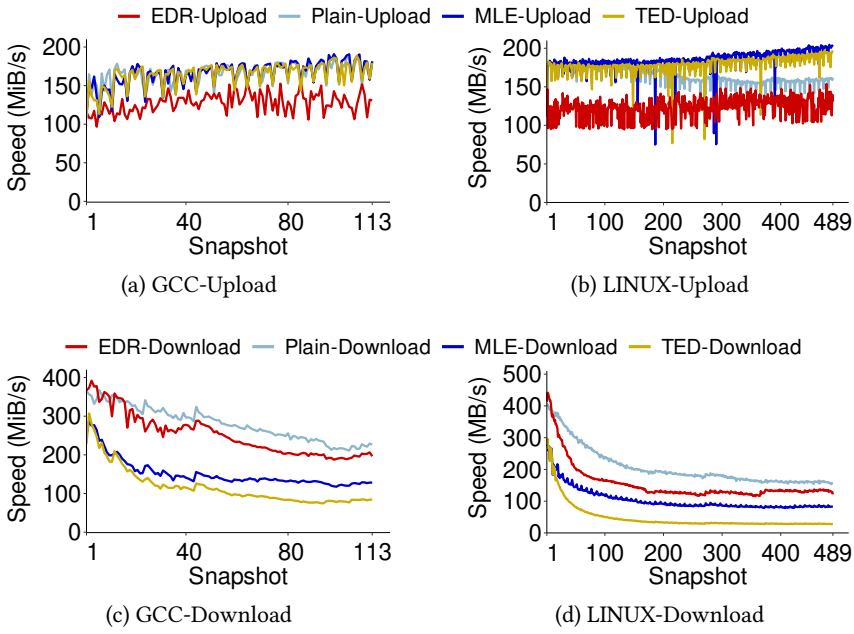
We evaluate the upload and download performance of EDRStore in both the local cluster and real-cloud testbeds.

Testbeds. For the local cluster, we deploy EDRStore over 12 machines with up to 10 clients, the key server, and the cloud. Each of the client and key server machines has a quad-core 3.4 GHz Intel Core i5-7500 CPU, 32 GiB RAM, and a TOSHIBA DT01ACA 1 TiB 7200 rpm SATA hard disk, while the cloud machine has a 16-core 2.50 GHz Intel(R) Xeon(R) Silver 4215 CPU, 96 GiB RAM, and a Western Digital Ultrastar DC SN640 3.84 TiB NVMe SSD. All machines run Ubuntu 20.04 and are connected via 10 GbE. We measure the upload and download throughput (Exp#6 and Exp#7) as well as the CPU utilization (Exp#9) of EDRStore.

For the real-cloud testbed, we use Amazon's Elastic Compute Cloud (EC2) [9] and Simple Storage Service (S3) [10] to set up a geo-distributed environment. We deploy the cloud and the key server on EC2 instances in Virginia (in East America), and deploy a client respectively in Virginia and California (in West America). We use EC2 spot instances [7], which leverage the unused EC2 capacity to reduce the VM running costs of regular instances (e.g., 60-90% less [8]) but may be interrupted occasionally. This is a cost-effective choice for backup workloads, where storage persistence is often more important than access performance. We configure each EC2 spot instance with type `t2.2xlarge`, which has eight vCPUs on a 2.30 GHz Intel(R) Xeon(R) CPU E5-2686 v4 CPU and 32 GiB RAM. Each instance is installed with Linux Ubuntu 20.04. We mount S3 via S3FS [56] in the cloud as the storage backend.

Methodology. We focus on two datasets: GCC (with a medium logical size) and LINUX (with the largest logical size). For the experiments in the local cluster (Exp#6 and Exp#7), we note that the disk I/Os of client machines may bottleneck the upload and download performance. To examine the maximum achievable performance without client-side disk I/Os, we load the snapshots into the client's memory before each upload experiment, and let the clients store the downloaded snapshots in memory. For the real-cloud experiment (Exp#8), we report the end-to-end performance results that include the disk I/Os of both the client and the cloud.

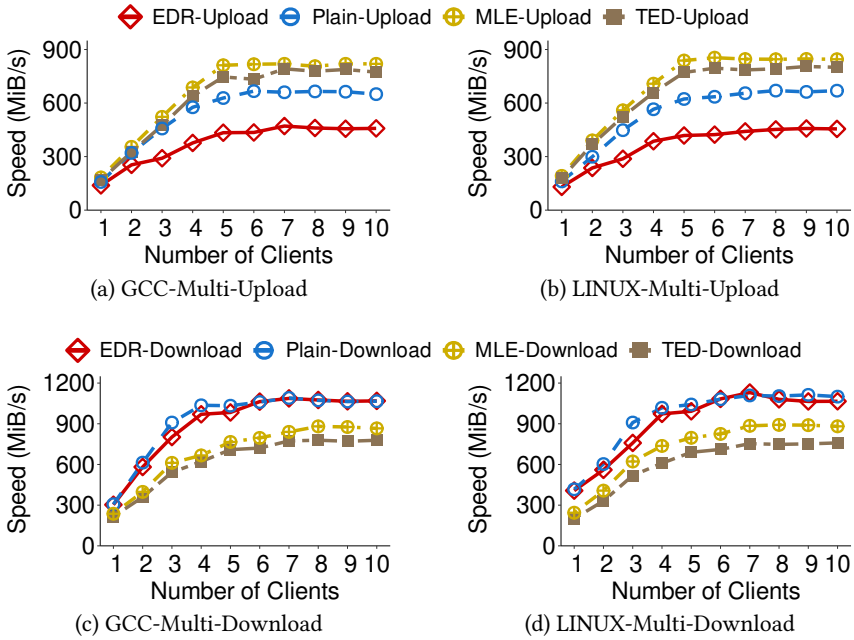
Exp#6 (Upload/download speeds). We compare the upload and download speeds of EDRStore, Plain, MLE, and TED. We first focus on a single client. The client uploads all snapshots until they are persisted to the local storage in the cloud machine, and then downloads the snapshots from the cloud machine. Figure 10 shows the single-client upload and download speeds of EDRStore, Plain, MLE, and TED.



(a) GCC-Upload (b) LINUX-Upload

(c) GCC-Download (d) LINUX-Download

Fig. 10. (Exp#6) Single-client Upload and download speeds.



(a) GCC-Multi-Upload (b) LINUX-Multi-Upload

(c) GCC-Multi-Download (d) LINUX-Multi-Download

Fig. 11. (Exp#6) Multiple-clients Upload and download speeds.

We first compare EDRStore and Plain and examine the performance overhead of EDRStore due to its secure data protection. For uploads, EDRStore is slightly slower than Plain, as it also performs key generation, encryption, and management of the base chunk index. For example, in

GCC, the average upload speeds of EDRStore and Plain are 127.6 MiB/s and 165.0 MiB/s, respectively (i.e., EDRStore is 22.6% slower than Plain); in LINUX, the average upload speed of EDRStore is 123.0 MiB/s, which is 24.7% slower than that of Plain (i.e., 163.3 MiB/s).

For downloads, EDRStore is slower than Plain, as it performs decryption and decompression in the client. For example, in GCC, the average download speed of EDRStore is 251.5 MiB/s, which is 10.3% slower than that of Plain (i.e., 280.3 MiB/s); in LINUX, the average download speeds of EDRStore and Plain are 154.6 MiB/s and 206.0 MiB/s, respectively (i.e., EDRStore is 25.0% slower than Plain). Note that EDRStore outperforms Plain in the first nine snapshots of LINUX downloads. The reason is that in such snapshots, EDRStore does not have as many similar chunks as Plain after encryption, and Plain needs to perform delta decoding on a large number of similar chunks in downloads. For both EDRStore and Plain, the download speeds gradually decrease across snapshots due to chunk fragmentation after deduplication and delta compression [17, 41, 78, 78]. We can extend EDRStore with efficient restore approaches [17, 41, 77, 78] to mitigate the fragmentation issue.

We then compare EDRStore with encrypted deduplication approaches (i.e., MLE and TED) to show the performance overhead of EDRStore by achieving higher storage savings via delta compression and local compression. For uploads, EDRStore is slower than MLE and TED. For example, in GCC, the average upload speed of EDRStore is 127.6 MiB/s, which is 23.2% and 22.3% slower than those of MLE (166.2 MiB/s) and TED (164.4 MiB/s); in LINUX, the average upload speed of EDRStore is 123.0 MiB/s, which is 33.4% and 31.2% slower than those of MLE (i.e., 184.6 MiB/s) and TED (i.e., 178.9 MiB/s), respectively.

For downloads, interestingly, EDRStore is faster than MLE and TED (e.g., in GCC, the average download speed of EDRStore is 251.5 MiB/s, which is 66.4% and 115.0% faster than those of MLE (i.e., 151.1 MiB/s) and TED (i.e., 117.0 MiB/s)). The reason is that EDRStore has a higher data reduction ratio than MLE and TED (Exp#2), meaning that less data is stored. During downloads, EDRStore retrieves less data from the cloud and hence incurs less I/O overhead. Even though EDRStore needs to perform extra local decompression and delta decoding, its overall download speed is still higher than those of MLE and TED.

We also evaluate the performance of EDRStore when multiple clients issue upload/download requests concurrently. We sample 10 different GCC snapshots, each of which has a size of around 500 MiB; and 10 different LINUX snapshots, each of which has a size of around 1 GiB. For each dataset, each client uploads one snapshot to the cloud and then downloads the snapshot from the cloud. We measure the *aggregate upload (download) speed* as the total uploaded (downloaded) data size divided by the total time when all clients complete the uploads (downloads). We compare the aggregate upload and download speeds of EDRStore, Plain, MLE, and TED.

Figure 11 shows the multi-client upload and download speeds versus the number of clients. Both GCC and LINUX show similar performance trends. The aggregate upload/download speeds for all the four approaches first increase and then slightly drop (or keep stable) due to the resource contention in the cloud machine (e.g., read/write contention and context switches across multiple clients).

For example, in GCC uploads, EDRStore reaches 471.1 MiB/s for 7 clients and then slightly drops to 458.0 MiB/s for 10 clients. In comparison, MLE reaches the highest aggregate upload speed of 819.7 MiB/s for 7 clients, while TED reaches 791.9 MiB/s, which are 74.0% and 68.1% faster than EDRStore. The reason is that MLE and TED do not perform delta compression and local compression, while EDRStore performs delta compression and local compression for additional storage savings. Plain can reach 668.7 MiB/s for 6 clients (i.e., 42.0% higher than EDRStore), as it does not perform any cryptographic operation. For downloads, EDRStore and Plain have similar speeds, which are

faster than those of MLE and TED. The reason is that MLE and TED have lower data reduction ratios and need to retrieve more data from the cloud during downloads than EDRStore and Plain.

Exp#7 (Performance breakdown). We focus on the upload and download performance of a single client in EDRStore using GCC and study the performance breakdown. We exclude the disk I/O overhead from the cloud and measure the computational time of each step of the upload and download operations. Specifically, we decompose the upload and download operations into the following steps.

- **Upload in the client:** (i) *Chunking*, in which the client partitions the input file into data chunks; (ii) *fingerprinting (D)*, in which the client generates fingerprints for data chunks; (iii) *feature generation (D)*, in which the client generates features for data chunks; (iv) *key generation*, in which the client sends the features of data chunks to the key server, and the key server returns the key seeds to the client; (v) *two-phase encryption*, in which the client encrypts data chunks via two-phase encryption; (vi) *feature generation (E)*, in which the client generates features for encrypted chunks; (vii) *version index management*, in which the client checks and updates the version index to decide if local compression is performed on data chunks; and (viii) *local compression*, in which the client performs local compression on the selected data chunks.
- **Upload in the cloud:** (i) *fingerprinting (E)*, in which the cloud generates fingerprints for encrypted chunks; (ii) *deduplication*, in which the cloud checks the fingerprint index to detect duplicate encrypted chunks; (iii) *delta compression (Uncomp)*, in which the cloud performs delta compression on the encrypted uncompressed chunks with respect to the base chunks in the base chunk index; (iv) *base chunk index management*, in which the cloud inserts or evicts chunks for the base chunk index; (v) *feature generation (Comp)*, in which the cloud generates features for the encrypted compressed chunks; and (vi) *delta compression (Comp)*, in which the cloud performs delta compression on the encrypted compressed chunks.
- **Download in the client:** (i) *decryption*, in which the client decrypts the received encrypted chunks; (ii) *local decompression*, in which the client performs local decompression on the compressed chunks; (iii) *re-encryption*, in which the client re-encrypts the decompressed base chunks; and (iv) *delta decompression (Case 2)*, in which the client performs delta decompression for reconstructing encrypted delta chunks. Note that Step (ii) corresponds to Cases 2 and 3 in downloads, while Steps (iii) and (iv) correspond to Case 2 in downloads (§4.4).
- **Download in the cloud:** (i) *delta decompression*, in which the cloud performs delta decompression on the encrypted delta chunk, corresponding to Cases 1 and 3 in downloads in §4.4.

Table 2 shows the breakdown, measured by the time for processing 1 MiB of data in each step of the upload and download operations. We first consider the upload operation. On the client side, both two-phase encryption and version index management incur low performance overhead. Feature generation is the most time-consuming step due to its expensive feature computation on all data and encrypted chunks. Note that the two rounds of feature generation in the client are necessary: the first (i.e., feature generation (D)) is on data chunks for similarity-aware key generation, and the second (i.e., feature generation (E)) is on encrypted chunks for selective local compression. On the cloud side, the most time-consuming step is delta compression, since it needs to fetch the base chunks and perform byte-level encoding to generate delta chunks. Note that the time for fingerprinting and feature generation on the client side is slightly higher than that in the cloud, since the cloud is deployed on a more powerful machine.

We next consider the download operation. On the client side, the most time-consuming step is the decompression as the client needs to decode every single byte based on its compression dictionary. The delta decompression step incurs much lower overhead than the local decompression step since it only needs to reconstruct the byte-level differences of a chunk, while the local decompression

Steps			Time (ms)
Client	Upload	Chunking	0.63 ± 0.013
		Fingerprinting (D)	3.41 ± 0.079
		Feature generation (D)	4.98 ± 0.058
		Key generation	1.22 ± 0.010
		Two-phase encryption	0.64 ± 0.008
		Feature generation (E)	4.98 ± 0.073
		Version index management	0.14 ± 0.002
		Local compression	4.26 ± 0.095
	Download	Decryption	0.36 ± 0.001
		Local decompression	1.00 ± 0.001
		Re-encryption	0.63 ± 0.012
		Delta decompression (Case 2)	0.35 ± 0.010
Cloud	Upload	Fingerprinting (E)	2.82 ± 0.006
		Deduplication	0.11 ± 0.001
		Delta compression (Uncomp)	5.33 ± 0.024
		Base chunk index management	1.34 ± 0.005
		Feature generation (Comp)	4.28 ± 0.007
		Delta compression (Comp)	5.23 ± 0.023
	Download	Delta decompression (Cases 1 and 3)	0.30 ± 0.002

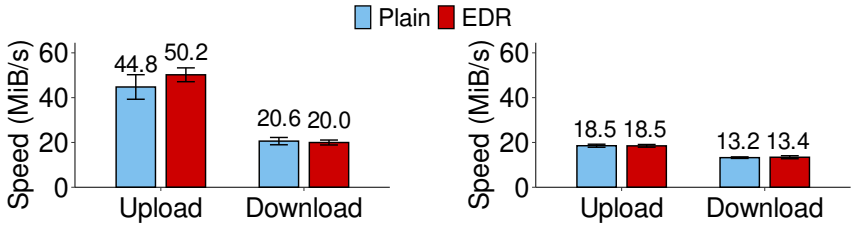
Table 2. (Exp#7) Breakdown of time for processing 1 MiB data in each step of the upload and download operations in EDRStore using GCC. The numbers are the average results over five runs with the 95% confidence interval based on the student's t-distribution.

step needs to reconstruct all bytes of a chunk. On the cloud side, its delta decompression time is less than that on the client side as the cloud machine is more powerful.

Exp#8 (Real-cloud uploads and downloads). We evaluate the real-cloud performance of EDRStore. We focus on GCC and a single client. We let the client upload and then download GCC snapshots. We compare EDRStore and Plain in the single-client upload and download speeds of EDRStore.

Figure 12 shows the results. As expected, the upload and download speeds depend on the region in which the client resides. When the client is in Virginia, where the cloud and the key server are deployed, the upload and download speeds are higher than when the client is in California. For example, the upload speed of EDRStore is 50.2 MiB/s in Virginia but drops to 18.5 MiB/s in California. Nevertheless, the performance of EDRStore is comparable to Plain, as network transmission and reads from S3 are the performance bottleneck.

Interestingly, EDRStore has a slightly higher upload speed than Plain when the client is in Virginia (by 12.1%). The reason is that Plain detects more similar chunks and hence fetches more base chunks from S3 than EDRStore to perform delta compression during uploads, and the access overhead to S3 is non-negligible. When the client is in California (Figure 12(b)), EDRStore and Plain have almost identical upload and download speeds. This result is affected by two factors. First, EDRStore introduces about 10% more upload traffic than Plain (Exp#5), thereby incurring extra overhead for EDRStore. Second, as stated above, Plain fetches more base chunks from S3 than EDRStore, thereby incurring extra overhead for Plain.



(a) Client deployed in Virginia (b) Client deployed in California

Fig. 12. (Exp#8) Real-cloud uploads and downloads. The client is deployed in Virginia or California, while both the key server and the cloud are deployed in Virginia. Each error bar represents the 95% confidence interval based on the student's t-distribution over five runs.

Dataset	Method	Client	Cloud	Key Server
GCC	Plain	(0.09 ± 0.013) %	(5.9 ± 0.06) %	-
	MLE	(35.4 ± 0.73) %	(5.3 ± 0.3) %	(0.3 ± 0.02)
	TED	(33.7 ± 0.51) %	(5.2 ± 0.04) %	(0.3 ± 0.01) %
	EDRStore	(49.1 ± 0.60) %	(12.7 ± 0.52) %	(0.7 ± 0.03) %
LINUX	Plain	(0.09 ± 0.015) %	(5.7 ± 0.04) %	-
	MLE	(34.0 ± 0.10) %	(4.8 ± 0.02) %	(0.3 ± 0.01)
	TED	(34.8 ± 0.15)	(4.9 ± 0.02)	(0.3 ± 0.01)
	EDRStore	(48.2 ± 0.52) %	(11.5 ± 0.08) %	(0.6 ± 0.02) %

Table 3. (Exp#9) CPU utilization. The numbers are the average results over five runs with the 95% confidence interval based on the student's t-distribution.

Exp#9 (CPU utilization). We evaluate the computational overhead of EDRStore by measuring the CPU utilization of the client, the cloud, and the key server in the single-client upload operation using GCC and LINUX. We compare EDRStore, Plain (which does not have a key server), MLE, and TED. We use Intel VTune Profiler [32] to measure the CPU utilization ratio, defined as the ratio between the average CPU time per CPU core (i.e., the time when a CPU core is actively running) and the elapsed time (i.e., the wall time of the overall operation).

Table 3 shows the results. The client-side CPU utilization ratio of EDRStore is much higher than that of Plain, since EDRStore introduces several computationally expensive operations to the client for secure data protection, such as feature generation, encryption, and fingerprint generation. The client-side CPU utilization ratio of EDRStore is also higher than those of MLE and TED, as EDRStore performs feature generation and local compression in the client for storage efficiency. Such CPU overhead can be reduced by deploying specialized hardware on the client side (e.g., using cryptographic accelerators with dedicated circuits [12]) to perform cryptographic operations. Also, we can use a lightweight resemblance detection algorithm [76] to mitigate the resource overhead of feature extraction. We pose the reduction of CPU overhead as a future work.

In the cloud, EDRStore incurs more CPU overhead than Plain due to dual-fingerprint deduplication and version-aware chunk management (§4.3). The cloud-side CPU utilization ratio of EDRStore is also higher than those of MLE and TED, as EDRStore performs delta compression in the cloud. The CPU overhead of the key server is limited for EDRStore, MLE, and TED (note that Plain has no key server).

8 RELATED WORK

Encrypted deduplication. In §2.2, we reviewed MLE [14] and server-aided MLE [13], the two representative primitives for encrypted deduplication. Follow-up studies enhance server-aided MLE with distributed key generation [24], frequency leakage mitigation [71], efficient metadata management [38], and performance optimization [47, 55], but they do not consider delta compression and local compression. DEBE [70] performs deduplication and local compression before encryption, but relies on hardware trusted execution.

Some approaches perform encryption based on data similarity. REED [53] generates a key for a segment of data chunks based on the minimum fingerprint of all data chunks in a segment, so that similar segments (which likely have the same minimum fingerprint) maintain large fractions of duplicate encrypted chunks for deduplication. Feature-based encryption [64] generates a key based on the feature derived from a file, so that duplicate data chunks from similar files (with the same key) can be mapped into duplicate encrypted chunks. However, they only focus on deduplication, but do not consider delta compression on similar chunks.

To enable delta compression on the encrypted data, a recent work, EDelta [19], preserves data similarity after encryption by applying the maximum Rabin hash [54] to data chunks as the key, so that similar chunks are encrypted by the same key (see the arguments in §4.1). However, there are two obvious security holes. First, it is vulnerable to offline brute-force attacks without having a dedicated key server [13]; in contrast, EDRStore introduces a dedicated key server and ensures that the key server cannot learn the chunk content and the resulting keys (§4.1). Second, a bitwise XOR operation of two encrypted similar chunks can remove the encryption key and reveal the XOR output of the corresponding plaintext data even without knowing the key (§4.2). Furthermore, EDelta [19] only considers the combination of encryption and delta compression, while EDRStore considers the combination of encryption and all three data reduction techniques (i.e., deduplication, delta compression, and local compression).

Encrypted local compression. Instead of applying encryption to locally compressed data chunks (§2.2), some studies address the combination of encryption and compression from a theoretical perspective by proposing new encryption primitives for supporting compression on encrypted chunks [34, 35, 37], but they are not implemented or empirically evaluated. Length-preserving compression [18] performs local compression on data chunks and pads zeroes on the compressed chunks before encryption, but it does not consider deduplication and delta compression.

9 CONCLUSION

EDRStore realizes encrypted data reduction by carefully combining deduplication, delta compression, and local compression with encryption to achieve both storage savings and data confidentiality. It includes new design schemes, including key generation, two-phase encryption, and selective local compression. We discuss the security guarantees and limitations of EDRStore. Our experiments show that EDRStore achieves high storage savings in real-world datasets and incurs moderate performance overhead.

REFERENCES

[1] 2012. BREACH: reviving the CRIME attack. Retrieved from http://breachattack.com/.
[2] 2023. Chromium: an open-source browser project. Retrieved from https://www.chromium.org/Home/.
[3] 2023. Docker Hub. https://hub.docker.com/.
[4] 2023. GCC: the GNU Compiler Collection. Retrieved from https://gcc.gnu.org/.
[5] 2023. The Linux Kernel Archives. Retrieved from https://www.kernel.org/.
[6] 2023. Tensorflow: an end-to-end open source machine learning platform. Retrieved from https://www.tensorflow.org/.
[7] Amazon Web Services, Inc. 2023. Amazon EC2 Spot Instance. Retrieved from https://aws.amazon.com/ec2/spot/.

[8] Amazon Web Services, Inc. 2023. Amazon EC2 Spot Instances pricing. Retrieved from https://aws.amazon.com/ec2/spot/pricing/.

[9] Amazon Web Services, Inc. 2023. Amazon Elastic Compute Cloud. Retrieved from https://aws.amazon.com/ec2/.

[10] Amazon Web Services, Inc. 2023. Amazon Simple Storage Service. Retrieved from https://aws.amazon.com/s3/.

[11] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS'07)*. 598–609.

[12] Michael Bartock, Murugiah Souppaya, Ryan Savino, Tim Knoll, Uttam Shetty, Mourad Cherfaoui, Raghu Yeluri, Akash Malhotra, Don Banks, Michael Jordan, Dimitrios Pendarakis, J. R. Rao, Peter Romness, and Karen Scarfone. 2022. *Hardware-Enabled Security:Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases*. Technical Report NIST IR 8320. National Institute of Standards and Technology.

[13] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. 2013. DupLESS: Server-aided encryption for deduplicated storage. In *Proceedings of the 22nd USENIX Security Symposium (Security'13)*. 179–194.

[14] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. 2013. Message-locked encryption and secure deduplication. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'13)*. 296–312.

[15] John Black. 2006. Compare-by-Hash: A Reasoned Analysis. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATC'06)*. 85–90.

[16] Andrei Z. Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings of the Conference on Compression and Complexity of Sequences (SEQUENCES'97)*. 21–29.

[17] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. 2018. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 309–324.

[18] Doron Chen, Michael Factor, Danny Harnik, Ronen Kat, and Eliad Tsfadia. 2021. Length preserving compression: marrying encryption with compression. In *Proceedings of the 2021 ACM International Conference on Systems and Storage (SYSTOR'21)*. 1–12.

[19] Yuchong Hu Chuang Gan, Leyan Zhao, Xin Zhao, Pengyu Gong, Wenhao ZWhang, Lin Wang, and Dan Feng. 2023. Enabling Encrypted Delta Compression forOutsourced Storage Systems via PreservingSimilarity. In *Proceedings of the 2023 IEEE International Conference on Computer Design (ICCD'23)*. 231–238.

[20] Yann Collet. 2023. Zstandard: Fast real-time compression algorithm. Retrieved from https://github.com/facebook/zstd.

[21] Biplob K Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*. 215–229.

[22] Peter Deutsch. 1996. *DEFLATE compressed data format specification version 1.3*. Technical Report.

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

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

[25] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. 2019. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 647–660.

[26] Morris J Dworkin. 2001. Recommendation for block cipher modes of operation: methods and techniques. (2001).

[27] Morris J Dworkin. 2010. Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices. (2010).

[28] Danny Harnik, Effi Ofer, Oded Naor, and Or Ozery. 2022. Rethinking block storage encryption with virtual disks. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. 9–14.

[29] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. 2010. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security & Privacy* 8, 6 (2010), 40–47.

[30] IDC. 2021. Global DataSphere and StorageSphere Forecasts. Retrieved from https://www.idc.com/getdoc.jsp?containerId=prUS47560321.

[31] IDC. 2021. State of Cloud Security 2021. Retrieved from https://www.vpngids.nl/wp-content/uploads/ermetic-idc-survey-report-state-of-cloud-security-2021.pdf.

[32] Intel Corporation. 2023. Intel VTune Profiler. Retrieved from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[33] Ari Juels and Burton S. Kaliski, Jr. 2007. PORs: Proofs of Retrievability for Large Files. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS'07)*. 584–597.

[34] Wei Kang and Nan Liu. 2016. Compressing encrypted data: Achieving optimality and strong secrecy via permutations. *IEEE Transactions on Information Theory* 62, 12 (2016), 7153–7163.
[35] James Kelley and Roberto Tamassia. 2014. Secure compression: Theory & practice. Retrieved from https://eprint.iacr.org/2014/113.pdf.
[36] John Kelsey. 2002. Compression and Information Leakage of Plaintext. In *Proceedings of the 9th Springer International Workshop on Fast Software Encryption (FSE'02)*. 263–276.
[37] Demijan Klinc, Carmit Hazay, Ashish Jagmohan, Hugo Krawczyk, and Tal Rabin. 2012. On compression of data encrypted with block ciphers. *IEEE transactions on information theory* 58, 11 (2012), 6989–7001.
[38] Jingwei Li, Suyu Huang, Yanjing Ren, Zuoru Yang, Patrick P. C. Lee, Xiao-song Zhang, and Yao Hao. 2021. Enabling Secure and Space-Efficient Metadata Management in Encrypted Deduplication. In *IEEE Transactions on Computers*, Vol. 71. 959–970.
[39] Jingwei Li, Patrick P. C. Lee, Chufeng Tan, Chuan Qin, and Xiaosong Zhang. 2020. Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. *ACM Transactions on Storage* 16, 1 (2020), 1–30.
[40] Mingqiang Li, Chuan Qin, and Patrick P. C. Lee. 2015. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. 111–124.
[41] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 183–197.
[42] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*. 111–123.
[43] Josh MacDonald. 2000. *File system support for delta compression.* Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, University of California at Berkeley.
[44] Joshua MacDonald. 2016. Xdelta: open-source binary diff, differential compression tools. Retrieved from http://xdelta.org/.
[45] Meta Platforms Inc. 2023. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. Retrieved from https://github.com/facebook/rocksdb.
[46] Dutch T Meyer and William J Bolosky. 2011. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*. 1–13.
[47] Mariana Miranda, Tânia Esteves, Bernardo Portela, and João Paulo. 2021. S2Dedup: SGX-enabled secure deduplication. In *Proceedings of the 2021 ACM International Conference on Systems and Storage (SYSTOR'21)*. 1–12.
[48] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. 2011. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proceedings of the 20th USENIX Conference on Security (Security'11)*. 65–75.
[49] Moni Naor and Omer Reingold. 2004. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM* 51, 2 (2004), 231–262.
[50] OpenSSL. 2023. Cryptography and SSL/TLS Toolkit. Retrieved from https://www.openssl.org/.
[51] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 2022. DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*. 247–264.
[52] Alfredo Pironti and Nikos Mavrogiannopoulos. 2013. Length Hiding Padding for the Transport Layer Security Protocol. Retrieved from https://datatracker.ietf.org/doc/html/draft-pironti-tls-length-hiding-02.
[53] Chuan Qin, Jingwei Li, and Patrick P. C. Lee. 2017. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Transactions on Storage* 13, 1 (2017), 1–30.
[54] Michael C. Rabin. 1981. *Fingerprint by Random Polynomials.* Technical Report. Center for Research in Computing Technology, Harvard University.
[55] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick P. C. Lee, and Xiaosong Zhang. 2021. Accelerating Encrypted Deduplication via SGX. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. 957–971.
[56] Randy Rizun. 2023. FUSE-based file system backed by Amazon S3. Retrieved from https://github.com/s3fs-fuse/s3fs-fuse.
[57] Seagate Technology LLC. 2020. Rethink Data: Put More of Your Business Data to Work - From Edge to Cloud. Retrieved from https://www.seagate.com/files/www-content/our-story/rethink-data/files/Rethink_Data_Report_2020.pdf.
[58] Claude Elwood Shannon. 1949. Communication theory of secrecy systems. *The Bell system technical journal* 28, 4 (1949), 656–715.
[59] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. 2012. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. 49–63.

[60] Haoliang Tan, Zhiyuan Zhang, Xiangyu Zou, Qing Liao, and Wen Xia. 2020. Exploring the Potential of Fast Delta Encoding: Marching to a Higher Compression Ratio. In *Proceedings of the 2020 IEEE International Conference on Cluster Computing (CLUSTER'20)*. 198–208.

[61] The Apache Software Foundation. 2023. Cassandra: Open source NoSQL database. Retrieved from https://cassandra.apache.org/.

[62] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. 2002. *Zdelta: An efficient delta compression tool.* Technical Report. Department of Computer and Information Science, Polytechnic University.

[63] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. 33–48.

[64] Suzhen Wu, Zhanhong Tu, Zuocheng Wang, Zhirong Shen, and Bo Mao. 2021. When Delta Sync Meets Message-Locked Encryption: a Feature-based Delta Sync Scheme for Encrypted Cloud Storage. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21)*. 337–347.

[65] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 286–298.

[66] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. 2014. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In *Proceedings of the 2014 IEEE Data Compression Conference (DCC'14)*. 203–212.

[67] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. 2015. DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Trans. Comput.* 65, 6 (2015), 1692–1705.

[68] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. 2014. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation* 79 (2014), 258–272.

[69] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. 101–114.

[70] Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. 2022. Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC'22)*. 37–52.

[71] Zuoru Yang, Jingwei Li, Yanjing Ren, and Patrick P. C. Lee. 2022. Tunable Encrypted Deduplication with Attack-Resilient Key Management. *ACM Transactions on Storage* 18, 4 (2022), 32:1–32:38.

[72] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 121–128.

[73] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. 2017. MiniCrypt: Reconciling encryption and compression for big data stores. In *Proceedings of the 2017 ACM European Conference on Computer Systems (EuroSys'17)*. 191–204.

[74] Yukun Zhou, Dan Feng, Wen Xia, Min Fu, Fangting Huang, Yucheng Zhang, and Chunguang Li. 2015. SecDep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In *Proceedings of the 31st IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*. 1–14.

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

[76] Xiangyu Zou, Cai Deng, Wen Xia, Philip Shilane, Haoliang Tan, Haijun Zhang, and Xuan Wang. 2021. Odess: Speeding up Resemblance Detection for Redundancy Elimination by Fast Content-Defined Sampling. In *Proceedings of the 37th IEEE International Conference on Data Engineering of (ICDE'21)*. 480–491.

[77] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. 2022. Building a High-performance Fine-grained Deduplication Framework for Backup Storage with High Deduplication Ratio. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC'22)*. 19–36.

[78] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. 2021. The Dilemma between Deduplication and Locality: Can Both be Achieved?. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. 171–185.