Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption

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

Abstract

Outsourced storage should fulfill confidentiality and storage efficiency for large-scale data management. Conventional approaches often combine encryption and deduplication based on deduplication-after-encryption (DaE), which first performs encryption followed by deduplication on encrypted data. We argue that DaE has fundamental limitations that lead to various drawbacks in performance, storage savings, and security in secure deduplication systems. In this paper, we study an unexplored paradigm called deduplication-before-encryption (DbE), which first performs deduplication and encrypts only non-duplicate data. DbE has the benefits of mitigating the performance and storage penalties caused by the management of duplicate data, but its deduplication process is no longer protected by encryption. To this end, we design DEBE, a shielded DbE-based deduplicated storage system that protects deduplication via Intel SGX. DEBE builds on frequency-based deduplication that first removes duplicates of frequent data in a space-constrained SGX enclave and then removes all remaining duplicates outside the enclave. Experiments show that DEBE outperforms state-of-the-art DaE approaches.

1 Introduction

Data outsourcing to public cloud storage provides a plausible solution for low-cost, large-scale data storage management in the face of explosive data growths [71]. To defend against data privacy leakage [57], clients require end-to-end encryption, such that their outsourced data be encrypted before being stored in (untrusted) public cloud storage. However, traditional symmetric encryption prohibits cross-user deduplication (i.e., removing duplicate data from multiple clients), since each client encrypts its own outsourced data with a distinct secret key, implying that the encrypted outputs from multiple clients are also distinct.

The literature has numerous studies (e.g., [3, 7, 8, 18, 23, 72, 74, 79]) on how to seamlessly combine encryption and deduplication for secure deduplicated storage in data outsourcing, which we collectively refer to as *deduplication-after-encryption (DaE)*. DaE first performs encryption on the outsourced data on the client side for confidentiality, followed by applying cross-user deduplication in the cloud to remove duplicate encrypted data for storage savings. To preserve the identical content after encryption, DaE encrypts data using a symmetric key derived from the content of each *chunk* (the basic unit of deduplication), such that duplicate original chunks (called *plaintext chunks*) are always encrypted by the same key into duplicate encrypted chunks (called *ciphertext chunks*) that are later removed by deduplication.

Despite its popularity, we argue that DaE has fundamental limitations including high key management overhead, incompatibility with compression, and security risks (see §2.1 for details). Since DaE always manages a key for each chunk for encryption before deduplication, it not only unnecessarily generates a huge number of keys for duplicate chunks that will later be removed by deduplication, but also incurs high storage overhead for managing a huge number of keys for all duplicate and non-duplicate chunks [47]. In addition, DaE stores non-duplicate encrypted chunks, whose contents look randomized and have limited room for further space reduction from compression. Furthermore, DaE necessitates deterministic encryption to preserve the deduplication capability on ciphertext chunks. Such a deterministic nature is vulnerable to information leakage through frequency analysis [48, 49].

The limitations of DaE motivate us to explore a simple but unexplored paradigm called *deduplication-before-encryption (DbE)*, which first performs deduplication on the plaintext chunks and then encrypts the remaining non-duplicate plaintext chunks with any key that is independent of the chunk content. A major distinction from DaE is that DbE does not need to manage per-chunk keys for encryption/decryption, and we argue that DbE addresses the limitations of DaE (§2.2). However, DbE remains unexplored in secure deduplicated storage, mainly because the chunks are no longer protected by encryption in deduplication processing, which is carried out in the cloud for cross-user deduplication.

Our insight is that the deduplication process in DbE can be protected with *shielded execution* [4, 37]. To this end, we present DEBE, a shielded DbE-based deduplicated storage system with performance, storage savings, and security in mind. DEBE builds on Intel Software Guard Extensions (SGX) [41], which provides a shielded execution environment, called an *enclave*, for secure deduplication processing. A key challenge of realizing DEBE in SGX is the limited enclave space (e.g., up to 128 MiB [36]). Thus, we propose *frequency-based deduplication*, a two-phase deduplication scheme that can realize secure and lightweight deduplication with the space-constrained enclave. Specifically, DEBE first performs deduplication on the most frequent chunks inside an enclave,

[*]Corresponding author: Jingwei Li (`jwli@uestc.edu.cn`)

motivated by our observation that the most frequent chunks often contribute to a large fraction of duplicates in real-world backup workloads (§4.1). It then performs deduplication on the remaining less frequent chunks outside the enclave. With frequency-based deduplication, DEBE has the key advantages of: (i) high performance, as it removes most duplicates in the first-phase deduplication and incurs limited performance overhead for the second-phase deduplication outside the enclave; (ii) high storage savings via both deduplication and compression; and (iii) security, as it protects the most frequent chunks (which are more vulnerable to frequency analysis attacks [48]) inside the enclave.

We evaluate our DEBE prototype in a LAN testbed. DEBE achieves significant speedups over state-of-the-art DaE approaches (e.g., 10.09× and 13.08× speedups over DupLESS [7] in uploading non-duplicate and duplicate data, respectively). In our technical report [81], we also show that DEBE achieves high storage savings (e.g., 93.8% of key metadata storage savings compared with DaE) and reduces information leakage without compromising storage savings (e.g., by 87.7% of the relative entropy over TED [49], while TED incurs a storage blowup). The source code of our DEBE prototype is at: `https://github.com/yzr95924/DEBE`.

2 Background and Motivation

2.1 Limitations of Deduplication-after-Encryption

Deduplication is a widely deployed data reduction technique in modern storage [26, 27, 59, 77, 85]. We focus on *chunk-based deduplication*, which removes duplicates at the granularity of a *chunk*. Specifically, a deduplicated storage system partitions input file data into chunks. It identifies each chunk by a cryptographic hash (e.g., SHA-256), called a *fingerprint*, of the chunk content (assuming that fingerprint collisions of distinct chunks are practically impossible [10]). It maintains a key-value store, called the *fingerprint index*, to track the fingerprints of all existing stored chunks, and stores only the non-duplicate chunks. It also stores a manifest file, called the *file recipe*, for each file to track all chunks of the file in storage for file reconstruction. In addition, it may further apply *compression* to remove byte-level duplicates within the non-duplicate chunks for more storage savings [27, 73, 85].

Deduplication-after-encryption (DaE) combines deduplication and encryption for both confidentiality and storage savings. In DaE, a client locally encrypts the plaintext chunks and uploads the ciphertext chunks to the cloud, which then performs deduplication on the ciphertext chunks. One popular cryptographic primitive for DaE is *message-locked encryption (MLE)* [8], which formalizes that the key for chunk encryption/decryption is derived from the content of each chunk, so that identical plaintext chunks are always encrypted into identical ciphertext chunks for deduplication. An instantiation of MLE is *convergent encryption (CE)* [3, 18, 23, 72, 74, 79], which derives each chunk's key based on its fingerprint.

CE is vulnerable to *offline brute-force attacks* [7], in which an adversary enumerates all possible plaintext chunks to derive their secret keys, attempts to decrypt a ciphertext chunk using each key, and deduces the plaintext chunk if the decryption succeeds. DupLESS [7] defends against offline brute-force attacks in CE via server-aided key management, by deploying a *key server* that generates the key of each chunk based on a global secret (securely owned by the key server) and the chunk fingerprint. Also, DupLESS implements key generation based on an oblivious pseudorandom function (OPRF) [63] to prevent the key server from learning the chunks or the keys during key generation, and rate-limits the key generation requests from clients to defend against *online brute-force attacks*, in which a malicious client aggressively issues key generation requests for different plaintext chunks to the key server.

Limitations. DaE is the state-of-the-art paradigm for building secure deduplicated storage systems. However, we argue that DaE suffers from three fundamental limitations.

- *L1 (High key management overhead).* DaE generates one key per chunk, leading to huge overheads for maintaining all chunk-based keys. Also, each client needs to encrypt its chunk-based keys via its own master secret key for protection. Thus, the key storage overhead increases proportionally with the numbers of chunks and clients, and is particularly significant for the workloads with high content redundancy (e.g., backups [77]) as they store only small amounts of non-duplicate data after deduplication. Also, DupLESS [7], which realizes server-aided key management, generates a key for the encryption of each chunk before the chunk is uploaded to the cloud, even though the chunk is a duplicate and is later removed by deduplication. As DupLESS employs OPRF and rate-limiting in key generation (see above), its key generation is shown to be expensive [70]. In short, DaE incurs high key management overhead, both in terms of key storage and key generation.
- *L2 (Incompatibility with compression).* In DaE, the cloud cannot further save additional storage space of non-duplicate encrypted chunks via compression, as encrypted chunks have high-entropy (almost random) contents. While a client may apply compression to the plaintext chunks before encryption and upload the encrypted compressed chunks, this leaks the compressed chunk lengths and introduces security risks [13].
- *L3 (Security risks).* Server-aided key management in DupLESS [7] makes the key server a single point-of-attack. If an adversary compromises the key server and has access to the global secret, it can infer the secret keys of chunks via offline brute-force attacks as in CE. Also, DaE is deterministic by nature and realizes one-to-one mappings between plaintext chunks and ciphertext chunks. An adversary can launch frequency analysis to infer the original plaintext chunks from the frequency distribution of ciphertext chunks in deduplicated storage [48].

2.2 Moving to Deduplication-before-Encryption

Given the limitations of DaE (§2.1), we study an unexplored paradigm, namely *deduplication-before-encryption (DbE)*, for secure deduplicated storage. Its idea is to first perform deduplication on the plaintext chunks to remove duplicates, followed by encrypting the non-duplicate plaintext chunks into ciphertext chunks for storage.

DbE naturally offers several benefits over DaE. First, since deduplication is applied first, DbE can encrypt each non-duplicate plaintext chunk with a content-independent key as in traditional symmetric encryption (§1) without compromising deduplication. This avoids generating and storing per-chunk content-derived keys and reduces the key management overhead (i.e., L1 addressed). Second, DbE can apply compression to the non-duplicate plaintext chunks after deduplication for further storage savings, followed by encrypting the compressed non-duplicate plaintext chunks (i.e., L2 addressed). Finally, since DbE can perform encryption with a content-independent key, it no longer needs a key server for per-chunk key generation as in DupLESS. This removes the single point-of-attack in the key server (i.e., L3 addressed).

The major challenge of DbE, however, is to decide whether clients or the cloud should perform deduplication, which is no longer protected by encryption. We consider three scenarios:

- Each client maintains a local fingerprint index for its own plaintext chunks. It encrypts the non-duplicate plaintext chunks and uploads the ciphertext chunks to the cloud. However, this approach prohibits cross-user deduplication.
- The cloud maintains a global fingerprint index to track the stored chunks of all clients. Each client first submits the fingerprints of its own plaintext chunks to the cloud to query if they can be deduplicated. It encrypts the non-duplicate plaintext chunks identified by the cloud, and uploads the ciphertext chunks to the cloud. This approach, also referred to as *source-based deduplication* [35], is vulnerable to *side-channel attacks* [35, 62] since any malicious client can infer if some target chunk has already been stored by querying if the target chunk can be deduplicated.
- Each client uploads all chunks to the cloud. The cloud performs deduplication based on its global fingerprint index that tracks the stored chunks of all clients, followed by encrypting the non-duplicate chunks. This approach, also referred to as *target-based deduplication* [35], hides the deduplication pattern from the clients and is secure against side-channel attacks. However, each client inevitably exposes its plaintext chunks to the cloud.

Thus, DbE remains unexplored in the literature, while existing studies mostly focus on DaE for secure deduplicated storage.

2.3 Intel SGX

In this work, we realize DbE with target-based deduplication and show how we protect DbE via *shielded execution*. We implement shielded execution using Intel SGX [41]. As our major requirement is to provide a secure memory region for data processing in the untrusted cloud, we conjecture that our design can be supported with other shielded execution technologies (e.g., ARM TrustZone [67] and AMD SEV [2]).

SGX basics. SGX is a set of extended instructions for Intel CPUs to realize a shielded execution environment, called an *enclave*, in an encrypted and integrity-protected memory region called the *enclave page cache (EPC)*. It ensures confidentiality and integrity for in-enclave contents with hardware protection. It provides two interfaces to interact with untrusted applications outside the enclave: (i) *enclave calls (ECalls)*, which permit applications to safely access in-enclave contents, and (ii) *outside calls (OCalls)*, which allow in-enclave code to issue function calls in applications.

Challenges. Realizing DbE in SGX is non-trivial due to the resource constraints of an enclave. First, the EPC size is limited (e.g., up to 128 MiB [36]). When an enclave has memory usage exceeding the EPC size, it encrypts and evicts the unused memory pages to the unprotected main memory, and decrypts and verifies the integrity of the evicted pages when loading them back to the EPC. This incurs expensive EPC paging overhead [5, 21]. Although recent SGX designs support a large EPC size of up to 1 TiB [44], they provide weaker security guarantees due to the loss of integrity tree protection [28]. Second, both ECalls and OCalls involve expensive hardware operations (e.g., flushing TLB entries [5]) that lead to significant context switching overhead (e.g., around 8,000 CPU cycles per call [66, 78]).

3 Design Overview

3.1 DEBE Architecture

We make a case for DbE by designing DEBE, a shielded DbE-based deduplicated storage system based on Intel SGX [41]. Figure 1 presents the architecture of DEBE; note that DEBE does not maintain a key server as in DupLESS [7] (§2.1). We consider a *multi-tenant* scenario, in which the clients from different organizations store outsourced data to a cloud storage service (or the cloud in short). DEBE performs target-based deduplication [35] (§2.2) to remove the duplicate data of multiple clients in the cloud. Currently, each DEBE client uploads all its data to the cloud for deduplication. Although a client may apply deduplication to its own data to save upload bandwidth without introducing side-channel attacks [50], our design does not make this assumption.

To prevent the cloud from accessing any plaintext chunks during deduplication processing, DEBE hosts an enclave in the cloud and performs deduplication inside the enclave. To support the multi-tenant scenario, we assume that a trusted third party (e.g., a certificate authority in the public key infrastructure (PKI) [56]) is responsible for the enclave setup. Specifically, the trusted third party compiles the enclave code into a shared object (as a `.so` file). It distributes the shared object to the cloud, along with its signature for integrity ver-

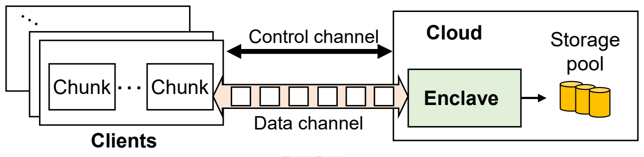


Figure 1: DEBE architecture.

ification. The cloud loads the shared object to bootstrap the enclave. The trusted third party can initiate *remote attestation* [41] to ensure that the correct code is loaded into the enclave, and it can go offline after the enclave is bootstrapped.

After the enclave is bootstrapped, each client sets up two secure communication channels: (i) the *control channel* with the cloud for transmitting the commands of storage operations and (ii) the *data channel* with the enclave for transmitting the plaintext chunks originated by the client. Currently, DEBE sets up the control channel between a client and the cloud using traditional SSL/TLS authentication. To set up the data channel between a client and the enclave, since the enclave cannot directly access the network socket of the cloud [41], DEBE implements the Diffie-Hellman key exchange to agree on a session key between a client and the enclave (§4.2), and the session key is used to protect the data channel. Note that other key exchange algorithms can be used for session key establishment.

To upload a file to the cloud, a client divides the file data into fixed-size or variable-size plaintext chunks as in traditional chunk-based deduplication (§2.1). It issues an upload request to the cloud through the control channel, and sends all plaintext chunks to the enclave through the data channel. The enclave deduplicates and compresses the received plaintext chunks on a per-batch basis (§4.1), encrypts the remaining non-duplicate compressed chunks into ciphertext chunks, and emits the ciphertext chunks and the file recipe to the storage pool.

To download a file, the client issues a download request to the cloud through the control channel. The enclave then retrieves the file's recipe and the corresponding ciphertext chunks. Finally, it decrypts the ciphertext chunks, and decompresses and returns the plaintext chunks to the client through the data channel.

Practical relevance of DEBE. DEBE focuses on multi-tenant deduplication, which is widely deployed in practice (e.g., Dropbox [24], Druva [25], Cohesity [14], and Memopal [58]) and is shown to achieve higher storage savings than single-tenant deduplication by removing the duplicate data from multiple clients [50,59,83]. Existing DaE approaches are also designed for multi-tenant deduplication, while DEBE addresses the limitations of DaE (§2.1). Although DEBE incurs costs due to shielded execution (e.g., the enclave verification fee from a trusted third party), its improvements over DaE approaches (in terms of performance, storage savings, and robustness; see §3.3) provide incentives for a cloud storage provider to use DEBE to provide secure and cost-effective cloud storage services for customers.

3.2 Threat Model

We consider an honest-but-curious adversary that does not modify the system protocol but aims to compromise data confidentiality by identifying the original content of the outsourced data stored in the cloud. The adversary can tap into the cloud and gain access to any data stored in the unprotected main memory of the cloud as well as the ciphertext chunks in the storage pool. It can also eavesdrop on the content of OCalls issued to the unprotected main memory (e.g., the parameters and untrusted functions used by OCalls).

Our threat model assumes that the enclave is trusted and reliable; its authenticity is verified by remote attestation [41] when it is created (§3.1). Any denial-of-service or side-channel attack against SGX is protected by existing solutions [64, 76]. Also, if the adversary has access to a compromised client, then it can access all the plaintext chunks of the client. However, since DEBE performs target-based deduplication (§3.1), the adversary cannot access or infer the plaintext chunks of other non-compromised clients.

3.3 Design Goals

DEBE is designed for clients from multiple tenants (§3.1) to securely outsource their storage management to public cloud storage services. It targets storage workloads with high content redundancy (e.g., backups [77] and file system snapshots [59]) that can be effectively removed by deduplication and compression. DEBE has the following design goals:

- *High performance.* DEBE has significantly lower key management overhead than DaE approaches. It also incurs limited overhead in SGX.
- *High storage savings.* DEBE supports *exact* deduplication (§4.4), i.e., all duplicates from multiple clients can be removed. It also applies compression to the non-duplicate chunks after deduplication for extra storage savings.
- *Confidentiality.* DEBE preserves the security of DaE by enforcing end-to-end encryption for the plaintext chunks between each client and the enclave, and the plaintext chunks are inaccessible by the cloud provided that the enclave is trusted and reliable (§3.2). DEBE remains secure against offline brute-force attacks in CE (§2.1), without the need of server-aided key management as in DupLESS [7].
- *Robustness over DaE.* DEBE mitigates the single point-of-attack of DaE by eliminating the key server. It also mitigates the information leakage caused by frequency analysis against DaE [48, 49].

4 Detailed Design

4.1 Main Idea

DEBE's core idea is to perform deduplication inside the enclave (hosted in the cloud), so as to provide confidentiality guarantees for the plaintext chunks during the deduplication process. Keeping a full fingerprint index (or the *full index* in short) inside the enclave can track the fingerprints of all

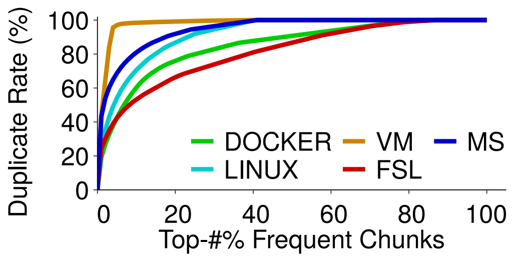


Figure 2: Duplicate rate versus top-percentage of frequent chunks in five real-world traces.

non-duplicate chunks being stored, but incurs significant EPC paging overhead due to the limited EPC size (§2.3). Alternatively, managing the full index outside the enclave saves the EPC usage, but incurs expensive context switching due to excessive OCalls for querying the full index (§2.3).

We propose *frequency-based deduplication*, which performs secure deduplication subject to the resource constraints of the enclave. Our insight is that the frequencies (i.e., numbers of duplicates) of chunks are highly skewed in practical backup workloads, such that *a small fraction of chunks can contribute to a large fraction of duplicates*. To justify, we conduct trace analysis on five real-world backup traces (see §6.1 for the trace details). We measure the *duplicate rate* for a subset of input chunks, defined as the ratio between the total size of duplicate chunks derived from the subset of chunks and the total size of duplicate chunks in the whole trace (note that a chunk is said to be a duplicate chunk if its identical copy has already been stored and it can be removed by deduplication). Figure 2 shows the duplicate rate versus the top-percentage of frequent chunks (ranked by their frequencies in descending order). For example, in the VM trace, the top-5% of frequent chunks contribute to a duplicate rate of around 97%. This implies that if we maintain a small fingerprint index to track the top-5% of frequent chunks, we can remove around 97% of duplicate data and achieve high storage savings.

The idea of frequency-based deduplication is to separate the deduplication process based on chunk frequencies. It manages a small fingerprint index inside the enclave to remove the duplicates from the most frequent chunks. It also maintains the full index outside the enclave to remove the remaining duplicates for the less frequent chunks. Frequency-based deduplication addresses both performance and security concerns. For performance, it only manages a small fingerprint index for the most frequent chunks inside the enclave to remove a large fraction of duplicate chunks. Thus, it mitigates the EPC paging overhead. It also reduces the context switching overhead as it only queries the full index outside the enclave via OCalls for a limited fraction of less frequent chunks. For security, since the most frequent chunks are more vulnerable to frequency analysis [48], we remove the duplicates of the most frequent chunks with in-enclave processing only. Thus, an adversary in the cloud cannot readily learn the frequencies of the most frequent chunks, and hence the information

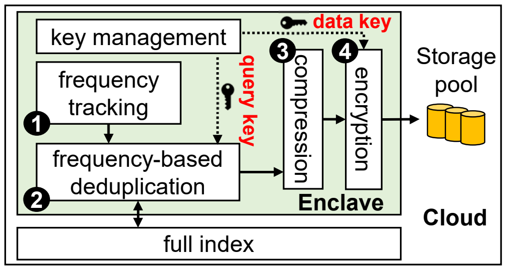


Figure 3: Architecture of the enclave.

leakage caused by frequency analysis is limited.

Enclave architecture and design roadmap. Figure 3 depicts the architecture of the enclave in DEBE. Initially, the enclave is bootstrapped with a set of keys and establishes secure data channels with each client (§4.2). Then the enclave tracks the frequency of each plaintext chunk received from the data channel of a client (§4.3). Based on the chunk frequencies, frequency-based deduplication removes the duplicates of the most frequent plaintext chunks and interacts with the full index outside the enclave to remove the duplicates of the remaining less frequent plaintext chunks (§4.4). The enclave performs compression on the non-duplicate plaintext chunks and encrypts the compressed plaintext chunks. Finally, the enclave stores the ciphertext chunks in the storage pool (§4.5).

4.2 Key Management

The enclave maintains a set of keys for the secure storage of chunks after deduplication and compression as well as for secure communication with clients.

Data key and query key. The enclave maintains two long-term keys, which remain valid throughout the lifetime of the enclave (i.e., the whole duration when DEBE is running): (i) the *data key* for encrypting and decrypting the compressed non-duplicate plaintext chunks in secure storage, and (ii) the *query key* for protecting the information of plaintext chunks when querying the full index outside the enclave (§4.4). When the enclave is bootstrapped, it initializes both the data key and the query key via the on-chip hardware random number generator (i.e., `sgx_read_rand` [42]). Both keys can be periodically renewed via existing approaches (e.g., key regression [30]), without compromising deduplication as DEBE performs deduplication before encryption.

Session key. Recall that each client maintains a data channel with the enclave for secure data communication (while maintaining a control channel with the cloud for securely issuing storage operations) (§3.1). Each data channel protects its communication using a short-term *session key*, which remains valid for a single communication session. It establishes a session key for the data channel using Diffie-Hellman key exchange through the control channel. The session key is kept in the enclave during the communication session of the client, and will be freed after the session is completed (both the control and data channels will be released as well).

Per-client master key. The enclave requires each client to submit a *master key* through the data channel for each storage request. It uses the master key to protect the file recipes for the client's files and enforces the client's ownership of the files. Similar to the session keys, the enclave only keeps the master key of the client for a single communication session and will destroy the master key at the end of the session, so the storage overhead for the master keys is also limited.

4.3 Frequency Tracking

The enclave needs to track the frequencies of plaintext chunks to identify the most frequent and less frequent chunks for frequency-based deduplication. To mitigate the EPC usage (§2.3), the enclave uses a *Count-Min Sketch (CM-Sketch)* [16] to track the *approximate* frequency of each chunk with fixed-size space and small errors.

The CM-Sketch is a two-dimensional array with r rows of w counters each. One key design here is to limit the computational overhead of mapping the plaintext chunks to the counters. To do so, our insight is that the chunk fingerprint is computed as a cryptographic hash (e.g., SHA-256 in our case), so we can treat the chunk fingerprint as a random input value and map it directly to a counter without compromising the accuracy of the CM-Sketch. Specifically, for each plaintext chunk M, the enclave partitions the fingerprint of M into r pieces. It takes the i-th piece modulo w to find one of the w counters, indexed from 0 to $w-1$, in row i ($1 \leq i \leq r$) and increments each of the mapped counters by one; this is in contrast to the traditional CM-Sketch, which maps the input to the counters of different rows using pairwise independent hash functions [16] and hence has extra computational overhead. To estimate the frequency of a chunk, the enclave uses the minimum value of the r mapped counters of the chunk. By default, we configure $r = 4$, $w = 256\,\text{K}$, and 4-byte counters, so the overall EPC usage of the CM-Sketch is only 4 MiB.

4.4 Frequency-based Deduplication

We present the design of frequency-based deduplication, which removes all duplicate plaintext chunks in two phases based on their estimated frequencies (§4.3).

First-phase deduplication. The enclave maintains a small fingerprint index, called the *top-k index*, to deduplicate the k most frequent plaintext chunks. We implement the top-k index as a combination of a min-heap and a hash table, as shown in Figure 4. The min-heap differentiates the top-k-frequent and less frequent plaintext chunks. It tracks top-k estimated frequencies of the plaintext chunks, such that the root heap entry corresponds to the plaintext chunk with the minimum frequency in the current top-k estimated frequencies. Each heap entry in the min-heap stores a pointer to a hash entry in the hash table. On the other hand, the hash table is used for duplicate detection, as in conventional deduplication. Each hash entry stores a mapping from the chunk fingerprint to a tuple of elements: (i) the pointer to the heap entry (i.e., both the heap entry and the hash entry reference each other), (ii) the estimated frequency of the chunk, (iii) the chunk address (including the container ID and the internal offset within the container; see §4.5), and (iv) the compressed chunk size (i.e., the size of the chunk after compression).

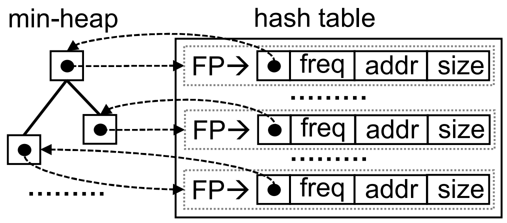


Figure 4: Overview of the top-k index.

Given a plaintext chunk, to perform the first-phase deduplication, the enclave takes the estimated frequency of the plaintext chunk obtained from the CM-Sketch (§4.3) and the chunk fingerprint as inputs. It first checks against the root heap entry of the min-heap. If the input frequency is smaller than the minimum frequency of the min-heap (i.e., the chunk is a less frequent chunk), the enclave skips querying the hash table for the chunk and proceeds to the second-phase deduplication (see below); otherwise (i.e., the chunk is a top-k-frequent chunk), the enclave uses the input fingerprint to look up the hash table. We have the following two cases:

- If the fingerprint is found in the hash table (i.e., the chunk is a duplicate), the enclave updates the frequency in the hash table and adds both the chunk address and the compressed chunk size to the file recipe (§4.5). Since the frequency is updated, it also adjusts the min-heap based on the pointer to the heap entry in the min-heap.
- If the fingerprint is not found in the hash table (i.e., the chunk is a new top-k-frequent chunk), the enclave creates a new hash entry in the hash table and inserts a new heap entry containing the pointer to the new hash entry into the min-heap. If the min-heap has already stored k heap entries, the enclave deletes the current root heap entry of the min-heap (with the minimum frequency) and also deletes the corresponding hash entry in the hash table via the pointer stored in the root heap entry. Since the chunk may have already been stored (but not tracked by the top-k index as its frequency is low), the enclave also runs the second-phase deduplication on the chunk and updates the chunk address and the compressed chunk size according to the result of the second-phase deduplication.

We show that the top-k index has low space usage. Suppose that the chunk fingerprint has 32 bytes (a SHA-256 hash), the chunk address has 12 bytes (an 8-byte container ID and a 4-byte internal offset; see §4.5), and the compressed chunk size has 4 bytes. For each top-k-frequent chunk, the hash entry additionally stores a 4-byte frequency and a pointer to a heap entry. Since we implement the min-heap as an array, the pointer to a heap entry can be represented as a 4-byte integer array index. Also, the heap entry keeps an 8-byte pointer to a hash entry. Overall, each top-k-frequent chunk uses 64 bytes in the top-k index (excluding the internal pointers of the hash table, which we now implement as an `unordered_map` of the C++ standard library). For example, to track 512 K most frequent chunks, the EPC usage of the top-k index is 32 MiB.

We further show that the top-k index has low time complexity. For each plaintext chunk, the top-k index can return the minimum frequency (from the root heap entry) in the current min-heap in constant time. For a top-k-frequent chunk, the top-k index needs to further check the hash table (in constant time) and update the min-heap. Since we store the pointer to the heap entry in the hash entry, we can directly update the corresponding heap entry when the frequency is changed, without searching the whole min-heap for its location. Thus, the time complexity of updating the min-heap is $\mathcal{O}(\log k)$.

Second-phase deduplication. The second-phase deduplication is performed on the plaintext chunks that are not removed by the first-phase deduplication, including the less frequent chunks and the fresh new top-k-frequent chunks whose fingerprints are new to the top-k index. DEBE manages a full index outside the enclave as the EPC size is limited (§2.3). We implement the full index as a hash table, in which each entry stores the mapping from the *encrypted fingerprint* of a plaintext chunk to the *encrypted chunk information* (i.e., the chunk address and the compressed chunk size, both of which are encrypted by the query key) of the corresponding ciphertext chunk. Our rationale of encrypting both the fingerprint and the chunk information is to prevent any adversary in the cloud from inferring the plaintext chunks, since the full index is not protected by the enclave.

Given a plaintext chunk, to perform the second-phase deduplication, the enclave *deterministically* encrypts the fingerprint of the plaintext chunk (not removed by the first-phase deduplication) with the query key (§4.2), so that duplicate plaintext chunks from different clients are always mapped to duplicate encrypted fingerprints for cross-user deduplication. It then queries the full index based on the encrypted fingerprint via an OCall. If the encrypted fingerprint is found in the full index, the OCall returns the encrypted chunk information that will be decrypted by the query key inside the enclave. Then the enclave will update the address and the compressed chunk size into the file recipe (§4.5). Otherwise, if the encrypted fingerprint is new to the full index, the enclave identifies this chunk as a non-duplicate chunk, assigns the chunk an address, compresses the chunk to obtain its compressed chunk size, and encrypts both the address and the compressed chunk size with the query key. It then updates the encrypted fingerprint and the corresponding encrypted chunk information in the full index. Note that the context switching overhead due to OCalls is limited, as a large fraction of duplicates are expected to have been removed by the first-phase deduplication.

Remarks. Traditional efficient indexing techniques for deduplication, such as similarity-based [9] and locality-based deduplication [52] approaches, can also address the limited EPC size by loading only a portion of the full index into the enclave. However, they only achieve *near-exact* deduplication (i.e., some duplicates cannot be removed), while DEBE can achieve exact deduplication (§3.3).

Note that the CM-Sketch may overestimate the chunk frequencies as multiple chunks can be mapped to the same counters (§4.3). Thus, the enclave may track some less frequent chunks in the top-k index. Nevertheless, it does not affect the storage savings from deduplication, as the full index still tracks all currently stored non-duplicate chunks.

4.5 Storage Management

Container storage. DEBE manages physical chunks in fixed-size *containers* to mitigate disk I/O costs [51]. The enclave performs compression on the non-duplicate plaintext chunks after deduplication, and encrypts the compressed non-duplicate plaintext chunks into ciphertext chunks with the data key. It writes each ciphertext chunk, together with an initialization vector (IV) (§5), into an in-memory container inside the enclave. When the in-memory container is full, the enclave emits it to persistent storage in the cloud. Note that DEBE only stores an IV (of size 16 bytes in AES-256) for each non-duplicate ciphertext chunk *after* deduplication, while DaE approaches store an encrypted key (of size 32 bytes in AES-256) for each ciphertext chunk *before* deduplication and incurs substantial key storage overhead when there exist many duplicate chunks (§2.1).

Also, the enclave creates and manages the file recipe for each uploaded file. Each entry in the file recipe keeps the chunk address and the compressed chunk size of each ciphertext chunk of the file. Note that when the enclave adds entries to the file recipe, it does not need to perform compression for the duplicate chunk to obtain its compressed chunk size, since the compressed chunk size has been stored in both the top-k index and the full index. To guarantee the ownership of the file, the enclave encrypts the file recipe by the client's master key and stores the encrypted file recipe as a regular file. Since the enclave treats containers (each of which contains multiple ciphertext chunks) as the basic I/O units and the chunk size is stored in the file recipe (protected by the per-user master key), DEBE preserves the security of compression as it avoids leaking the lengths of compressed chunks [13].

Downloads. To download a file, the client issues a download request and its master key to the enclave through the secure data channel. The enclave retrieves the file recipe and decrypts the file recipe with the given master key. It then parses the decrypted file recipe to obtain the chunk addresses and compressed chunk sizes. To restore all chunks, the enclave exposes the container IDs of the requested chunks to the cloud to perform I/Os via OCalls. Once the cloud fetches the corresponding containers into the unprotected main memory, the enclave accesses the ciphertext chunks based on their internal offsets and decrypts the ciphertext chunks by the data key. Finally, it decompresses and sends the plaintext chunks to the client through the data channel.

4.6 Security Discussion

We discuss the security of DEBE in response to our threat model (§3.2). We focus on two cases.

Case 1: A snapshot adversary gains one-time access to contents in unprotected memory and storage pool. DEBE enforces end-to-end encryption for the plaintext chunks between each client and the enclave, and provides *semantic security* [33] for the ciphertext chunks stored in the cloud. Specifically, it sets up a secure data channel that encrypts all plaintext chunks exchanged between a client and the enclave by a session key. It performs deduplication inside the enclave (that is oblivious to the adversary), and encrypts the non-duplicate plaintext chunks into ciphertext chunks by the data key before the ciphertext chunks are stored. Note that both data-in-transit and data-at-rest encryption operations are based on traditional symmetric encryption, and semantic security is achieved.

Case 2: A persistent adversary eavesdrops on OCalls in deduplication. DEBE encrypts both chunk fingerprints and chunk information by the query key inside the enclave before it includes them as the inputs of OCalls for accessing the full index outside the enclave. Thus, even though an adversary can eavesdrop on the OCalls, it cannot infer the original inputs from the OCalls.

One potential information leakage is that a persistent adversary (that stays in the cloud for a long time) can learn the chunk frequencies in the deduplication process, as the enclave maps duplicate plaintext chunks into duplicate encrypted fingerprints when querying the full index. Specifically, the adversary can track the frequency distribution of encrypted fingerprints by monitoring the OCalls, and launch frequency analysis to infer the plaintext chunks. However, DEBE limits such information leakage to the less frequent chunks, which are relatively robust against frequency analysis [48] (for comparisons, DaE leaks the frequencies of *all* chunks since it is deterministic by nature; see §2.1). Our evaluation shows that DEBE mitigates information leakage more effectively than TED [49], a state-of-the-art approach that trades storage savings for security (see our technical report [81]).

Remarks. A powerful adversary may launch frequency-based side-channel attacks by simultaneously compromising a client and the cloud. If it proactively lets the compromised client upload artificial chunks to the cloud and monitors OCalls in the cloud, the adversary could infer chunk frequencies and even identify the most frequent chunks among the clients. While the practical damage caused by such side-channel attacks remains an open question, we can obfuscate the chunk frequency information by perturbing the OCalls patterns (e.g., adding dummy OCalls), at the expense of incurring extra performance overhead.

5 Implementation

We have implemented a prototype of DEBE in C++ on Linux based on Intel SGX SDK Linux 2.7 [42]. It uses OpenSSL-1.1.1 [65] and Intel SGX SSL [43] for cryptographic operations. Our current prototype contains 17.5 K LoC.

Each client implements FastCDC [80] to realize content-defined chunking, where the minimum, average, and maximum chunk sizes are configured at 4 KiB, 8 KiB, and 16 KiB, respectively. The container size is 4 MiB. We implement Diffie-Hellman key exchange based on NIST P-256 elliptic curve for session key management of the data channel between the client and the enclave. The enclave computes the fingerprints of plaintext chunks via SHA-256 and encrypts each unique plaintext chunk via AES-256 in GCM mode with a random 16-byte IV. Also, it encrypts the fingerprint of each plaintext chunk via AES-256 in CMC mode with a 16-byte zero IV to support queries over encrypted fingerprints (as in CryptDB [68]). Both SHA-256 and AES-256 are configured to use (via OpenSSL EVP API) the Intel New Instructions Set for hardware-accelerated operations [39,40]. We also implement LZ4 [15] for lossless stream-based compression in the enclave for chunk compression after deduplication.

To mitigate context switching overhead of the enclave, DEBE transmits and processes chunks on a per-batch basis (the default batch size is 128 chunks). Also, to speed up downloads, the cloud keeps an in-memory least-recently-used cache (256 MiB by default) to hold the recently accessed containers. For each container access request issued by the enclave (§4.5), the cloud first checks the cache and retrieves the containers from local storage only if they are not in cache.

Limitations. DEBE currently does not address crash consistency. We now discuss how to extend DEBE with crash consistency, and pose the implementation as future work.

When a system crash occurs, DEBE would lose its in-enclave contents (i.e., the query key, the data key, the CM-Sketch, the top-*k* index, and the in-memory container pending to be persisted into the storage pool). We can extend DEBE to recover the query and data keys, the CM-Sketch, and the top-*k* index via *sealing*, an SGX feature that encrypts in-enclave content for secure out-enclave storage on disk [41]. When the enclave is bootstrapped (§3.1), DEBE stores a persistent copy of both the query key and data key by sealing. Also, it makes periodic snapshots of the CM-Sketch and top-*k* index by sealing. To restore the enclave states from a system crash, the cloud re-initializes the enclave by unsealing the keys and snapshots back to the enclave.

To realize crash consistency, we can augment DEBE with write-ahead logging [61] to record updates in on-disk logs before updating the in-memory CM-Sketch and top-*k* index. To recover from the data loss of the in-memory container, the enclave can log the IDs of the persisted containers for the currently uploaded file in on-disk logs. If a system crash occurs during the current upload, DEBE can roll back to the state before the upload starts based on the logs. It finally notifies the client to re-upload the file. Note that logging the changes into on-disk logs would incur extra OCalls. To mitigate the context switching overhead of logging, we can batch multiple logging operations in a single OCall.

We can initialize a new CM-Sketch and a new top-k index after enclave recovery. This would not affect the storage savings from deduplication, provided that the full index is crash-consistent (e.g., via its implementation in a persistent key-value store) and tracks all currently stored non-duplicate chunks. However, DEBE incurs extra performance overhead, as it cannot learn frequent chunks and hence incurs more OCalls to build the top-k index from scratch.

6 Evaluation

We deploy DEBE in a local cluster of 11 machines connected with 10 GbE. Each machine has a quad-core 3.4 GHz Intel Core i5-7500 CPU and 32 GiB RAM, and is installed with Ubuntu 16.04. We deploy one or multiple clients, a key server (for DaE only), or a cloud storage backend on distinct machines. The machine for the cloud storage backend is attached with a TOSHIBA DT01ACA 1 TiB 7200 rpm SATA hard disk. By default, DEBE sets $k = 512$ K for the top-k index and configures the CM-Sketch with $r = 4$ and $w = 256$ K, so as to keep the overall EPC usage within 64 MiB to limit the paging overhead in SGX. Note that we can tune the parameters based on the available EPC size.

We evaluate DEBE using both synthetic and real-world datasets. We summarize our evaluation results as follows.

- DEBE accelerates the uploads of non-duplicate and duplicate data of state-of-the-art DaE approaches by up to 10.09× and 13.08×, respectively (Exp#1). Its frequency-based deduplication only takes 5.8-18.4% of the overall upload time (Exp#2). It preserves high performance for multi-client uploads/downloads (Exp#3) and various synthetic workloads (Exp#4).
- For real-world workloads, DEBE achieves 1.17-2.76× speedups over state-of-the-art deduplication alternatives (Exp#5), and preserves high performance in long-term uploads and downloads (Exp#6).

In our technical report [81], we present additional evaluation results and show that DEBE achieves high storage savings and preserves security against frequency analysis.

6.1 Datasets

Synthetic datasets. We consider two synthetic datasets for our evaluation. The first dataset, namely *SYN-Unique*, includes non-duplicate and individually compressible chunks. Specifically, we generate SYN-Unique as a set of 2 GiB compressible files via the LZ data generator [38], which generates synthetic data based on SDGen [34]. The LZ data generator takes two parameters as inputs: (i) a compression ratio, which specifies the compressibility of the generated data, and (ii) a random seed for data generation. We configure the compression ratio as 2 to resemble real-world backup workloads [77], and vary the random seeds to generate distinct synthetic files. We perform chunking on each synthetic file to ensure that its chunks are globally unique over all files. We use the dataset for stress-testing different schemes with non-duplicate chunks.

The second dataset, namely *SYN-Freq*, includes the original chunks (before deduplication) following a target frequency distribution. We build a synthetic file generator that generates files whose chunk frequencies follow a Zipf distribution as shown by prior work [83, 84]. Our generator takes three parameters as inputs: (i) the number of original chunks, (ii) the deduplication ratio (i.e., the ratio between the original data size and the non-duplicate data size), and (iii) the Zipfian constant (a larger constant implies higher skewness). To generate a synthetic file, we prepare a set of non-duplicate 48-bit fingerprints based on the expected number of non-duplicate chunks (i.e., the number of original chunks divided by the deduplication ratio). We assign each fingerprint with a compression ratio based on the normal distribution with a mean of 2 and a variance of 0.25 [46, 77]. To generate each original chunk, we sample its fingerprint from the fingerprint set based on the target Zipf distribution, and construct its content using the LZ data generator [38] with the compression ratio and fingerprint (as the random seed) as inputs. Finally, we generate the SYN-Freq dataset as a set of synthetic files, each of which contains 13,107,200 8-KiB original chunks (i.e., 100 GiB) and a deduplication ratio of 5×. The number of non-duplicate chunks is large enough that only a subset of non-duplicate chunks can be tracked by the top-k index.

Real-world datasets. We consider five real-world datasets of backup workloads, which are also used in previous studies for trace-driven evaluation [49, 50, 69, 70, 80, 86]:

- *DOCKER*: docker snapshots (from v4.1.0 to v7.0.0) of Couchbase [17] from Docker Hub [22];
- *LINUX*: snapshots (from stable versions between v2.6.13 and v5.9) of Linux source code [54], in which each snapshot is stored in the *mtar* format [53];
- *FSL*: home directory snapshots [29], among which we select 42 snapshots from nine users in 2013;
- *MS*: Windows file system snapshots [59], among which we select 30 snapshots of size around 100 GiB each; and
- *VM*: virtual machine snapshots [50] collected by ourself.

Table 1 shows the statistics of the five real-world datasets. Previous studies have shown that multi-tenant deduplication can achieve higher storage savings than single-tenant deduplication in FSL, MS, and VM [50, 59, 75]. Given the limited available disk space in our testbed, we sample a subset of snapshots from the original datasets [29, 59] for FSL and MS as in [49]. As FSL, MS, and VM only contain fingerprints, we generate compressible chunk contents as in SYN-Freq.

6.2 Evaluation on Synthetic Data

To examine the maximum achievable performance without disk I/O overhead, we load the synthetic files into each client's memory before each test and let the cloud store all post-

Dataset	Raw size	Snapshots	Dedup Ratio	Compress Ratio
DOCKER	70.2 GiB	94	4.2	1.7
LINUX	44.6 GiB	82	2.8	2.3
VM	3.0 TiB	660	33.4	2.0
FSL	3.0 TiB	42	8.2	2.0
MS	3.9 TiB	30	4.1	2.0

Table 1: Characteristics of real-world datasets.

deduplicated data in memory (we include the disk I/O overhead in the evaluation in §6.3). We report the average results over five runs and include the 95% confidence intervals based on student's t-distribution (except for line graphs).

Exp#1 (Overall performance). We evaluate the upload (download) performance of overall systems. We consider a single client that successively uploads the same 2 GiB file from SYN-Unique *twice*. The client then downloads the same file. We measure the upload (download) speed of each operation. Our goal is to examine the maximum achievable performance of all schemes for storing all non-duplicate data and all duplicate data. Note that the file size is small here, such that all fingerprints can be tracked in the top-*k* index in DEBE (we consider large-scale datasets in §6.3).

We compare DEBE with three DaE approaches: (i) *DupLESS* [7], which implements server-aided key management based on OPRF (§2.1); (ii) *TED* [49], which generates the key of each chunk based on lightweight hash computations in the key server; and (iii) *CE* [23], the convergent encryption scheme (§2.1). To study the security overhead of DEBE, we include plain deduplication (*Plain*), in which the client uploads the plaintext chunks to the cloud for deduplication and compression through a communication channel protected by SSL/TLS. Unlike DEBE and Plain, the DaE schemes (i.e., DupLESS, TED, and CE) do not realize compression due to incompatibility (§2.1). For fair comparisons, we re-implement all baselines based on their original papers under the same implementation setting (§5) in C++.

Figure 5(a) shows the upload speeds. DEBE outperforms all DaE schemes. When uploading non-duplicate data, DEBE achieves 10.09×, 1.42×, and 1.25× speedups over DupLESS, TED, and CE, respectively, by avoiding the generation of chunk-based keys (note that DupLESS has very low upload speeds due to the expensive OPRF operations). Even though DEBE applies compression, its compression overhead is masked by the performance gain over the key generation overhead of DaE. When uploading duplicate data, DEBE becomes more efficient. Its speedups increase to 13.08×, 1.88×, and 1.65× over DupLESS, TED, and CE, respectively, since it avoids performing encryption and compression on the duplicate chunks. Compared with plain deduplication, DEBE only incurs 21.6% and 7.4% performance overhead for the uploads of non-duplicate and duplicate data, respectively.

Figure 5(b) shows the download speeds. All DaE schemes follow the same download paradigm, in which the client retrieves both ciphertext chunks and encrypted chunk-based keys from the cloud, decrypts each key and the corresponding

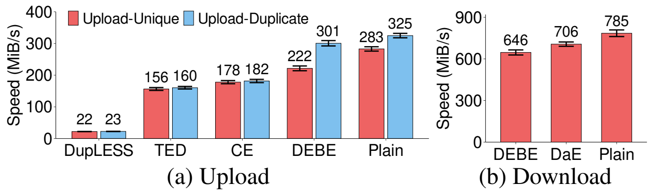


Figure 5: (Exp#1) Overall performance.

Steps	1st upload	2nd upload
Chunking	0.61 ± 0.01 ms	
Transmission Protection	0.37 ± 0.01 ms	
Fingerprinting	2.27 ± 0.04 ms	
Frequency tracking	0.06 ± 0.01 ms	
First-phase dedup	0.10 ± 0.01 ms	0.14 ± 0.01 ms
Second-phase dedup	0.80 ± 0.02 ms	-
Compression	0.67 ± 0.01 ms	-
Encryption	0.33 ± 0.01 ms	-

Table 2: (Exp#2) Breakdown of computational time per processing 1 MiB data in two successive uploads.

chunk, and reconstructs the original file. Compared with DaE, DEBE incurs 8.5% download speed drop due to the OCalls for moving chunks into the enclave for decryption and decompression (§4.5). Also, DEBE and DaE have 17.7% and 10.1% download speed drops compared with Plain, respectively, since they perform decryption on retrieved chunks.

Exp#2 (Upload breakdown). We study the breakdown of the upload performance. We consider the same scenario as Exp#1 (i.e., a client successively uploads the same 2 GiB file from SYN-Unique twice) and measure the computational time of the client and the enclave in different steps in uploads: (i) *chunking*, in which the client partitions the input file into plaintext chunks; (ii) *transmission protection*, in which the enclave exchanges a session key with the client and decrypts received ciphertext chunks; (iii) *fingerprinting*, in which the enclave computes the fingerprint of each plaintext chunk; (iv) *frequency tracking*, in which the enclave estimates the frequency of each plaintext chunk via the CM-Sketch; (v) *first-phase deduplication*, in which the enclave removes duplicate plaintext chunks via the top-*k* index; (vi) *second-phase deduplication*, in which the enclave queries the full index via OCalls to remove remaining duplicates; (vii) *compression*, in which the enclave compresses the non-duplicate chunks; and (viii) *encryption*, in which the enclave encrypts the compressed chunks with the data key.

Table 2 shows the results (measured by the computational time per 1 MiB of uploads). In the first upload (i.e., uploading non-duplicate data), fingerprinting is the most time-consuming step since it performs expensive computations on all chunks. On the other hand, frequency-based deduplication (including frequency tracking plus first-phase and second-phase deduplication) takes only 18.4% of the overall time. Note that since the storage is empty before the upload, each non-duplicate chunk is treated as a frequent chunk and examined by both the first-phase and second-phase deduplication.

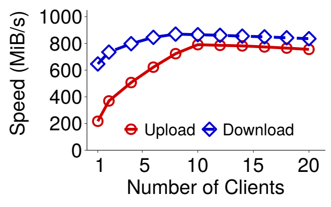


Figure 6: (Exp#3) Multi-client uploads and downloads.

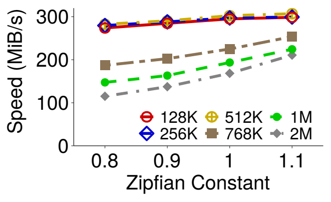


Figure 7: (Exp#4) Impact of frequency distribution.

In the second upload (i.e., uploading duplicate data), all duplicate chunks are removed by the first-phase deduplication and hence the second-phase deduplication is skipped. In this case, frequency-based deduplication takes only 5.8% of the overall upload time.

Exp#3 (Multi-client uploads and downloads). We evaluate DEBE when multiple clients issue upload/download requests concurrently. In addition to the cloud, we deploy 10 machines, with two client instances each, so as to simulate the concurrent uploads/downloads by up to 20 clients. Each client uploads a 2 GiB synthetic file from SYN-Unique to the cloud, and then downloads the same 2 GiB file. We measure the *aggregate* upload (download) speed as the ratio of the total uploaded (downloaded) data size to the total time all clients complete the uploads (downloads).

Figure 6 shows the results versus the number of clients. The aggregate upload speed first increases with the number of clients and reaches 791.1 MiB/s for 10 clients, followed by dropping to 755.8 MiB/s for 20 clients due to the resource contention in the enclave. The aggregate download speed has a similar trend, and first increases to 870.0 MiB/s and finally drops to 835.7 MiB/s.

Exp#4 (Impact of frequency distribution). We evaluate DEBE on processing the chunks from different frequency distributions. We configure a single client to upload each original chunk of SYN-Freq without chunking, and measure the *computational speed* of the enclave (i.e., including the steps of Table 2 except chunking).

Figure 7 shows the results for different k in the top-k index versus the Zipfian constant. A larger k implies lower performance for all Zipfian constants, since SGX incurs significant paging overhead when the size of enclave contents is greater than 64 MiB [45]. For example, when the Zipfian constant is 0.8, the computational speeds for $k = 512$ K and $k = 1$ M are 282.5 MiB/s and 147.3 MiB/s, respectively. In addition, the computational speed of the enclave increases in more skewed distribution (i.e., a larger Zipfian constant), since the most frequent chunks contribute more duplicates. This mitigates the OCall overhead of querying the full index.

6.3 Evaluation on Real-world Traces

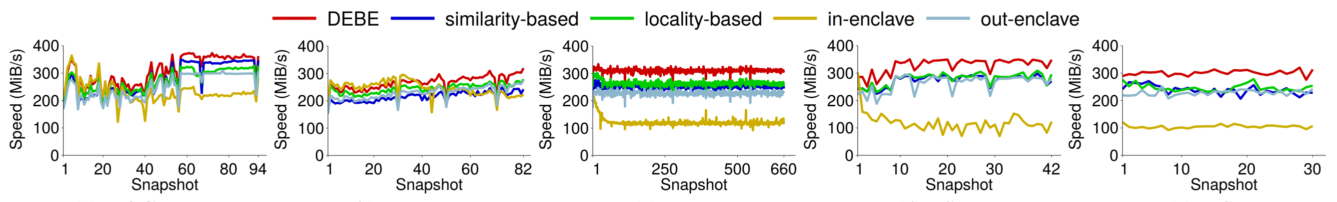
Exp#5 (Performance of deduplication approaches). DEBE's key design is frequency-based deduplication, and we compare it with other design alternatives. We consider two state-of-the-art memory-efficient deduplication approaches, namely *similarity-based deduplication* [9] and *locality-based deduplication* [52]. Both approaches store a small in-enclave fingerprint index based on the *feature* of each segment of chunks and perform deduplication by loading a portion of the full index (outside the enclave) into the enclave based on the matched feature. Similarity-based deduplication derives the feature based on the minimum chunk fingerprint of each segment of chunks, while locality-based deduplication generates it by sampling a few fingerprints. As in [9, 52], we choose the segment size as 10 MiB, and the sampling rate of locality-based deduplication as 1/64. While both approaches aim to mitigate disk I/O in plain deduplication, our idea is that they can also be applied to reduce EPC usage, but can only support near-exact deduplication (§4.4).

In addition to the above near-exact deduplication approaches, we include the naïve but exact deduplication baselines for comparisons. Specifically, *in-enclave deduplication* attempts to manage the full index in the enclave; when the full index increases in size and cannot fit into the EPC, it triggers page swapping to evict unused EPC pages to memory. *Out-enclave deduplication* manages the full index in memory, and detects duplicates by issuing OCalls to the full index. For fair comparisons, we include compression over non-duplicate chunks into all baseline approaches. We upload the snapshots of each real-world backup dataset (§6.1) in the order of their creation times. We measure the computational speed of the enclave as in Exp#4.

Figure 8 shows the results. DEBE generally outperforms all approaches. For example, in FSL, it achieves 1.17×, 1.20×, 1.25×, and 2.76× average speedups over the similarity-based, locality-based, out-enclave, and in-enclave approaches, respectively. The reason is that DEBE avoids the extra computational overhead of compressing and encrypting some duplicate chunks in both similarity-based and locality-based approaches (which perform near-exact deduplication). Also, it performs the first-phase deduplication and filters out many queries to the full index, thereby mitigating the OCall overhead of the out-enclave deduplication. Although in-enclave deduplication outperforms DEBE when the workload size is small (e.g., the first few snapshots in DOCKER and LINUX), its performance drops dramatically in subsequent snapshots due to expensive paging overhead. DEBE manages lightweight data structures (a CM-Sketch and the small top-k index) in the enclave and mitigates the paging overhead.

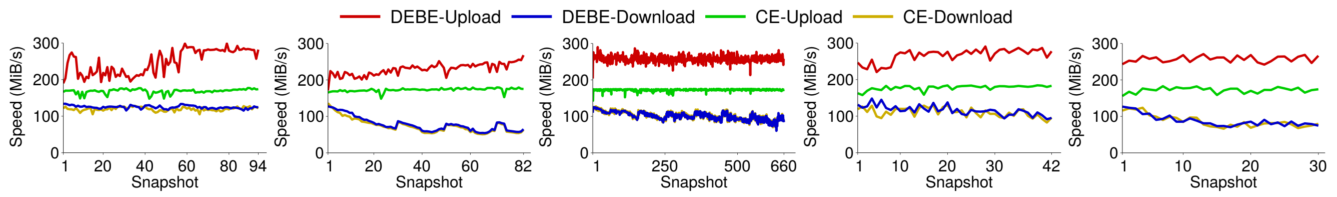
Exp#6 (Trace-driven upload and download). Unlike in Exp#1, we evaluate the upload and download performance of DEBE based on real-world data. We enable cloud-side disk I/O, upload all snapshots of each dataset, and finally let the client download them on disk. Here, we only compare DEBE with CE, which is the fastest DaE approach. Since FSL, VM, and MS only include the compressible chunks (§6.1), we let the client machine directly upload chunks without chunking.

Figure 9 shows the speeds for uploading and downloading each snapshot in DEBE and CE. The upload speed of



(a) DOCKER (b) LINUX (c) VM (d) FSL (e) MS

Figure 8: (Exp#5) Performance comparison of different deduplication approaches.



(a) DOCKER (b) LINUX (c) VM (d) FSL (e) MS

Figure 9: (Exp#6) Trace-driven upload and download performance.

DEBE gradually increases in subsequent snapshots, which include more duplicate plaintext chunks, so DEBE does not need to perform compression and encryption on the duplicate plaintext chunks (removed by deduplication). In contrast, CE is consistently slower than DEBE in uploads, as it performs key generation and encryption for all duplicate plaintext chunks. For example, in FSL, the upload speed of DEBE is 246.5 MiB/s for the first snapshot, and reaches 277.5 MiB/s for the last snapshot. In contrast, the upload speed of CE is 163.5-179.1 MiB/s across all snapshots.

The download speeds of both DEBE and CE are almost the same, since they are throttled by disk I/O. Also, their download speeds decrease across snapshots due to chunk fragmentation [51] (i.e., the chunks of subsequent snapshots become scattered after deduplication), which increases I/O overhead. For example, the download speed of DEBE in FSL is 131.4 MiB/s for the first snapshot, and drops to 95.1 MiB/s for the last snapshot (the download speed of CE is almost the same). Chunk fragmentation can be mitigated via existing approaches [11, 12, 31, 51, 86] and we pose the integration of such approaches into DEBE as future work.

7 Related Work

DaE approaches. Several approaches realize secure deduplication via DaE. In addition to those described in §2.1, some approaches are designed from the security perspectives. Random MLE [1] and iMLE [6] apply non-deterministic encryption to prevent frequency leakage, but they use expensive primitives (e.g., non-interactive zero-knowledge proofs [1], fully homomorphic encryption [6]) that are not ready to be implemented. Liu *et al.* [55] propose to share keys via a decentralized agreement protocol without relying on a dedicated key server, but it introduces expensive performance overhead of interactions among different clients. TED [49] mitigates frequency leakage with a configurable storage blowup. In contrast, DEBE realizes DbE to address both key management overhead and security issues simultaneously.

SGX meets secure deduplication. SGX has been used in secure deduplication. Dang *et al.* [20] employ SGX as a trusted proxy to save network bandwidth for secure deduplication. SPEED [19] performs deduplication for computations inside the enclave to improve resource utilization. You *et al.* [82] leverage SGX to verify the ownership of deduplicated data for secure deduplication. SeGShare [32] builds on a server-side enclave for file-based deduplication, but does not consider fingerprint indexing for chunk-based deduplication. S2Dedup [60] uses a server-side enclave to eliminate a trusted key server for key generation, and it performs deduplication outside the enclave via re-encrypting chunks; in contrast, DEBE directly performs deduplication inside the enclave to protect plaintext chunks. SGXDedup [70] leverages SGX to improve the performance of client-side secure deduplication under DaE. Note that the above SGX-based deduplication approaches are still based on DaE.

8 Conclusion

DEBE realizes an unexplored paradigm, deduplication-before-encryption (DbE), for secure deduplicated storage. It builds on SGX and applies frequency-based deduplication to manage a small fingerprint index for most frequent chunks in the enclave. We show that DEBE outperforms conventional deduplication-after-encryption (DaE) approaches in performance, storage savings, and security.

Acknowledgments

We thank our reviewers and shepherd for their comments. This work was supported in part by the National Natural Science Foundation of China (61972073), the Key Research Funds of Sichuan Province (2021YFG0167, 2020YFG0298), the Sichuan Science and Technology Program (2020JDTD0007), the Fundamental Research Funds for Chinese Central Universities (ZYGX2020ZB027, ZYGX2021J018), CUHK Direct Grant (4055148), and the Research Matching Grant Scheme.

References

[1] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In *Proc. of CRYPTO*, 2013.

[2] Advanced Micro Devices Inc. AMD Secure Encrypted Virtualization (SEV). `https://developer.amd.com/sev/`.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of USENIX OSDI*, 2002.

[4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proc. of ACM HASP*, 2013.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. SCONE: Secure Linux containers with Intel SGX. In *Proc. of USENIX OSDI*, 2016.

[6] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Proc. of PKC*, 2015.

[7] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.

[8] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EuroCrypto*, 2013.

[9] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. of IEEE MASCOTS*, 2009.

[10] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX ATC*, 2006.

[11] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proc. of USENIX FAST*, 2019.

[12] Z. Cao, H. Wen, F. Wu, and D. H. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proc. of USENIX FAST*, 2018.

[13] D. Chen, M. Factor, D. Harnik, R. Kat, and E. Tsfadia. Length preserving compression: Marrying encryption with compression. In *Proc. of ACM SYSTOR*, 2021.

[14] Cohesity Inc. Cohesity. `https://www.cohesity.com/`.

[15] Y. Collet. LZ4: Extremely fast compression algorithm. `https://lz4.github.io/lz4/`.

[16] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[17] Couchbase Inc. Couchbase: The modern database for enterprise applications. `https://www.couchbase.com`.

[18] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of USENIX OSDI*, 2002.

[19] H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou. SPEED: Accelerating enclave applications via secure deduplication. In *Proc. of IEEE ICDCS*, 2019.

[20] H. Dang and E.-C. Chang. Privacy-preserving data deduplication on trusted processors. In *Proc. of IEEE CLOUD*, 2017.

[21] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont. Everything you should know about Intel SGX performance on virtualized systems. In *Proc. of ACM SIGMETRICS*, 2019.

[22] Docker Inc. Docker Hub. `https://hub.docker.com/`.

[23] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.

[24] Dropbox Inc. Dropbox. `https://www.dropbox.com/`.

[25] Druva Inc. Druva. `https://www.druva.com/`.

[26] A. Duggal, F. Jenkins, P. Shilane, R. Chinthekindi, R. Shah, and M. Kamat. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *Proc. of USENIX ATC*, 2019.

[27] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplication-large scale study and system design. In *Proc. of USENIX ATC*, 2012.

[28] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen. Scalable memory protection in the PENGLAI enclave. In *Proc. of USENIX OSDI*, 2021.

[29] File System and Storage Lab at Stony Brook University. Traces and snapshots public archive. `http://tracer.filesystems.org`.

[30] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of NDSS*, 2006.

[31] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proc. of USENIX ATC*, 2014.

[32] B. Fuhry, L. Hirschoff, S. Koesnadi, and F. Kerschbaum. SeGShare: Secure group file sharing in the cloud using enclaves. In *Proc. of IEEE/IFIP DSN*, 2020.

[33] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 1984.

[34] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck. SDGen: Mimicking datasets for content generation in storage benchmarks. In *Proc. of USENIX FAST*, 2015.

[35] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[36] D. Harnik, E. Tsfadia, D. Chen, and R. Kat. Securing the storage data path with SGX enclaves. `https://arxiv.org/abs/1806.10883`, 2018.

[37] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. of ACM HASP*, 2013.

[38] J. Ibsen. LZ data generator. `https://github.com/jibsen/lzdatagen`.

[39] Intel Corporation. Intel(R) Advanced Encryption Standard Instructions (AES-NI). `https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html`.

[40] Intel Corporation. Intel(R) SHA Extensions. `https://software.intel.com/content/www/us/en/develop/articles/intel-sha-extensions.html`.

[41] Intel Corporation. Intel(R) Software Guard Extensions. `https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top.html`.

[42] Intel Corporation. Intel(R) Software Guard Extensions SDK for Linux. `https://01.org/intel-softwareguard-extensions`.

[43] Intel Corporation. Intel(R) Software Guard Extensions SSL. `https://github.com/intel/intel-sgx-ssl`.

[44] Intel Corporation. Supporting Intel SGX on multi-socket platforms. `https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html`.

[45] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proc. of ACM EuroSys*, 2019.

[46] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proc. of USENIX ATC*, 2014.

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

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

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

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

[51] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, 2013.

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

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

[54] Linux Foundation. The Linux kernel archives. `https://www.kernel.org/`.

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

[56] U. Maurer. Modelling a public-key infrastructure. In *Proc. of ESORICS*, 1996.

[57] M. Meehan. Data privacy will be the most important issue in the next decade. `https://www.forbes.com/sites/marymeehan/2019/11/26/data-privacy-will-be-the-most-important-issue-in-the-next-decade/`.

[58] Memopal. Memopal. `https://www.memopal.com/`.

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

[60] M. Miranda, T. Esteves, B. Portela, and J. Paulo. S2Dedup: SGX-enabled secure deduplication. In *Proc. of ACM SYSTOR*, 2021.

[61] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1):94–162, 1992.

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

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

[64] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*, 2018.

[65] OpenSSL. Cryptography and SSL/TLS toolkit. `https://www.openssl.org/`.

[66] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless OS services for SGX enclaves. In *Proc. of ACM EuroSys*, 2017.

[67] S. Pinto and N. Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51(6):1–36, 2019.

[68] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of SOSP*, 2011.

[69] C. Qin, J. Li, and P. P. C. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Trans. on Storage*, 13(1):1–30, 2017.

[70] Y. Ren, J. Li, Z. Yang, P. P. C. Lee, and X. Zhang. Accelerating encrypted deduplication via SGX. In *Proc. of USENIX ATC*, 2021.

[71] Seagate Technology LLC. Data Age 2025. `https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf`.

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

[73] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. of USENIX FAST*, 2012.

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

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

[76] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. of USENIX Security*, 2018.

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

[78] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *Proc. of ACM ISCA*, 2017.

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

[80] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proc. of USENIX ATC*, 2016.

[81] Z. Yang, J. Li, and P. P. C. Lee. Secure and lightweight deduplicated storage via shielded deduplication-before-encryption. Technical report, The Chinese University of Hong Kong, 2022. `http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_debe.pdf`.

[82] W. You and B. Chen. Proofs of ownership on encrypted cloud data via Intel SGX. In *Proc. of ACNS*, 2020.

[83] W. Zhang, D. Agun, T. Yang, R. Wolski, and H. Tang. VM-centric snapshot deduplication for cloud data backup. In *Proc. of IEEE MSST*, 2015.

[84] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng. Multi-level selective deduplication for VM snapshots in cloud storage. In *Proc. of IEEE CLOUD*, 2012.

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

[86] X. Zou, J. Yuan, P. Shilane, W. Xia, H. Zhang, and X. Wang. The dilemma between deduplication and locality: Can both be achieved? In *Proc. of USENIX FAST*, 2021.

A Artifact Appendix

Abstract

Our artifact consists of the prototypes of DEBE and all baseline approaches, a trace analysis tool for frequency leakage measurement, and the scripts to run all our experiments in §6. The DEBE prototype is a shielded DbE-based deduplicated storage system that supports secure deduplication via Intel SGX. It supports upload/download operations to allow multiple clients to securely outsource their data storage to the cloud. It applies frequency-based deduplication and implements the designs described in §4.

Scope

Our artifact can be used to validate our main claim that DEBE outperforms conventional DaE approaches in performance, storage efficiency, and security. Specifically, our artifact can be used to validate the results shown in the figures and tables in §6 to support our main claim. In addition, our artifact can be used to run the workloads independent of our evaluation in §6.

Contents

The artifact comprises the following sub-directories:

- `./Prototype`, which includes the implementation of the DEBE prototype.
- `./Baseline`, which includes the implementation of all baseline approaches (e.g., DupLESS, TED, CE, and Plain) used in Exp#1 and Exp#6.
- `./Sim`, which includes a trace analysis tool to measure frequency leakage of CE, TED, and DEBE (see Exp#9 in our technical report [81]).

Also, each sub-directory has a separate README file to introduce the build instructions and usage.

Hosting

Our artifact is available on GitHub. Users can obtain the artifact from the repository `https://github.com/yzr95924/DEBE`. The version we provided for the artifact evaluation is marked with the `atc22ae` tag. We encourage the users to use the latest version of the repository, since it may include bug fixes. We also release the traces used in §6. The README file (`https://github.com/yzr95924/DEBE/blob/master/README.md`) describes the detailed instructions to collect the traces.

Requirements

We developed and evaluated our artifact on a local cluster of 11 machines connected with 10 GbE. Each machine has a quad-core 3.4 GHz Intel Core i5-7500 CPU and 32 GiB RAM running Ubuntu 16.04. We implement DEBE based on Intel SGX SDK Linux 2.7 [42], OpenSSL 1.1.1 [65], and Intel SGX SSL 1.1.1 [43]. The DEBE prototype is written in C++ and compiled by Clang 3.8.0. To validate the basic upload/download operations of DEBE, users need to prepare at least two machines, one of which needs to support Intel SGX to run as the cloud. We recommend users to check the SGX-supported device in `https://github.com/ayeks/SGX-hardware`.

Note that if the user's OS version is higher than Ubuntu 16.04 LTS (e.g., Ubuntu 20.04 LTS), it might not be able to install the packages with the same versions as in our paper. Nevertheless, we expect that the impact of using the packages with newer versions would be limited and our prototype can still run correctly.

Workflow

To reproduce the experiments in §6, users can refer to `./Prototype/ae_instruction.md` for the detailed instructions.