Deduplication vs Privacy Tradeoffs in Cloud Storage

(extended abstract of the MSc dissertation)

Rodrigo de Magalhães Marques dos Santos Silva
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professor Luís Rodrigues and Professor Miguel Correia

Abstract

To ensure the privacy of their data when stored in the cloud, users can choose to encrypt files before exporting them. Unfortunately, without additional mechanisms, encrypted data storage makes it impossible to implement server-side deduplication techniques, as two identical files will have different encrypted versions. In this thesis, we address the problem of reconciling the need to encrypt data with the advantages of deduplication. In particular, we study techniques that achieve this objective while avoiding frequency analysis attacks, i.e., attacks that infer the content of an encrypted file based on how frequently the file is stored and/or accessed. We propose a new protocol for assigning encryption keys to files that leverages the use of trusted execution environments to hide the frequency of chunks from the adversary.

1 Introduction

The use of *cloud storage* systems is now ubiquitous and has numerous advantages, but on the other hand, it can compromise data privacy of customers in the face of curious operators. One way to get around this limitation consists of encrypting the data before exporting it to the cloud. Unfortunately, this may prevent the Storage Service (SS) from applying mechanisms that significantly reduce the resources required to provide the service, such as deduplication mechanisms.

Deduplication is a technique that allows a storage provider to identify duplicated files (or *chunks* of these files) and therefore, avoid storing too many copies of the same content, by deleting some redundant copies. Such a technique can obtain significant savings in storage cost, since it was experimentally verified that in systems that store large amounts of data from several users, such as Dropbox [1] and Google Drive [2], it is possible to find large amounts of repeated data. Unfortunately, it is difficult to apply deduplication to encrypted data. If users encrypt their data before storing it and use different keys to perform the encryption, the same chunk will produce different cryptograms. This makes the storage provider unable to identify which data is redundant, preventing deduplication.

Encrypted deduplication is the name given to techniques that combine file encryption and data deduplication. This combination usually requires some form of direct or indirect coordination between the different clients. The challenge is to carry out this coordination in an efficient manner that allows for the preservation of the privacy of the information stored by each user. A potential vulnerability of encrypted deduplication techniques is exposure to *frequency analysis attacks*. A frequency analysis attack allows storage providers to infer which content is encrypted by observing how often it is accessed.

Recent work takes advantage of Trusted Execution Environment (TEE) to increase the performance and security of encrypted deduplication systems [3, 4], in this document, we follow the same path.

We present FH-Dedup, a tunable encrypted deduplication system that leverages TEE to perform sensitive cryptographic operations and keep track of the frequency of all chunks. This data is stored in a frequency table (cache) internal to the TEE and in a larger encrypted table stored in an untrusted environment. We are the first to offer full privacy protection while achieving exact deduplication. We name our system FH-Dedup since we provide secure deduplication through *Frequency Hiding*. We propose techniques to protect the vulnerable external table stored outside the TEE. Additionally, our proposed protocol allows clients to read their files without requiring interaction with the TEE, offering an increased performance on reading operations, more than 5x faster.

The rest of the document is organized as follows: Section 2 introduces the main concepts relevant to our work and Section 3 describes the main techniques to implement encrypted deduplication. Section 4 describes the system architecture and implementation, and Section 5 shows our evaluation of the system. Finally, Section 6 concludes the document and highlights directions for future work.

2 Background

This section introduces concepts relevant to our work. We start by introducing deduplication (2.1) and encrypted deduplication (2.2), then we present the main attack against encrypted deduplication (2.3), we also present present secure mediation (2.4), and finally, we introduce TEEs, and their security mechanisms and properties (2.5).

2.1 Deduplication

Deduplication is a technique that allows the storage space needed to store large sets of data to be reduced by identifying duplicate content. We consider deduplication of equal-sized chunks of files, not complete files. When a client wants to store a file, the system divides the file into chunks. Then, the storage service checks whether each of these chunks have already been stored (by the same client or another). If so, storing a new replica set of that chunk is avoided. Studies show that deduplication can generate significant storage savings in a production environment, for example, savings of 50% on primary storage [5] and up to 98% on backup storage [6].

2.2 Encrypted Deduplication

When customers encrypt data before storing it, opportunities to perform deduplication can be drastically reduced. The reason is that if two clients store the same file/chunk, the ciphertext will be different. Encrypted deduplication is a class of techniques that aims to reconcile file encryption with deduplication. Most of these techniques are based on mechanisms that allow clients to coordinate, implicitly or explicitly, to choose the same keys to encrypt the same files, and in this way guarantee that chunks with the same content result in the same ciphertext. One of the possible techniques to achieve this end is Message-Locked Encryption (MLE) [7] that deterministically derives the encryption key from the data content, normally through a cryptographic function of *hash* (e.g., SHA-2) applied to the chunk content. However, MLE has several limitations. In particular, if a malicious storage provider has access to a given content, it can generate the corresponding ciphertext and then identify the clients that have this content, breaking the client's privacy.

2.3 Frequency Analysis

Unfortunately, encrypting the contents of files may not be enough to ensure the privacy of the contents stored by a given user. Another vulnerability can be exploited if an attacker knows the frequency in which certain content appears in a dataset, as it can correlate this value with the frequency with which a ciphertext is submitted for storage and infer relationships between the ciphertexts and their original content. Some encrypted deduplication techniques, such as MLE, suffer from this vulnerability: as a given content is always encrypted with the same key, the ciphertext frequency is exactly the same as the original content. That is, MLE preserves opportunities to apply deduplication, but does not offer any protection against frequency analysis attacks.

2.4 Secure Mediation

A way to achieve encrypted deduplication and prevent frequency analysis consists of resorting to a trusted entity, the *secure mediator*, that coordinates the allocation of content-encryption keys. Whenever clients need to encrypt a file, clients contact a secure intermediary, who is responsible for indicating which key must be used to encrypt each chunk. Given multiple copies of the same content, the choice of keys by the mediator allows separating these copies into different sets, within each set, the copies are encrypted with a given key (enabling deduplication), and each set uses a different key (avoiding frequency analysis). Trust in the mediator can be achieved using cryptographic techniques and/or secure hardware.

2.5 Trusted Execution Environments

A TEE is a secure mode of the CPU that allows one to run code and store data isolated from the operating system and user-level processes. Enclaves are trusted execution environments that are available on many common Intel CPUs; they are provided by an architecture called Intel Software Guard Extensions (SGX) [8], which uses hardware mechanisms such as *hardware secrets*, *remote attestation*, *sealed storage* and *memory cypher*. The use of enclaves is one of the possible techniques to realize secure mediators in encrypted deduplication systems.

The enclave relies on a hardware-protected memory region called Enclave Page Cache (EPC) to host protected code and data. An EPC comprises 4 KB pages, and any application in the enclave can use up to 128 MB. If an enclave is larger than EPC, it encrypts unused pages and transfers them to unprotected memory, suffering a performance penalty [9]. SGX provides two interfaces: ECALLs, used by an application to invoke enclave functionality, and OCALLs, used by enclave code to access an external application.

3 Related Work

In the literature, it is possible to find several proposals for secure mediators to facilitate encrypted deduplication.

Duplicateless Encryption for Simple Storage (DupLESS) [10] uses a dedicated server to assign encryption keys to chunks. Its operation is inspired by MLE since the encryption key depends on the *hash* of the chunk. However, unlike basic MLE, the encryption key also depends on a secret known only to the mediator. This prevents the storage server from knowing which ciphertext is associated with a given content. Trust in the mediator is achieved through cryptographic protocols, based on oblivious pseudo-random functions [11] and blind RSA signatures [12–14], which ensures that the server cannot extract the *hash* of the chunk and that the client can only extract the cipher key.

Tunable Encrypted Deduplication (TED) [15] uses a similar architecture, but adds mechanisms to control the number of times a given content is encrypted with the same key. For this purpose, the mediator keeps a record of how many times the same key has already been assigned to a given chunk, changing the key when necessary.

The use of trusted execution environments to run the mediator avoids the use of complex cryptographic algorithms to ensure trust in the mediator, which is assured directly by the *hardware*. In particular, the attestation mechanisms allow clients to confirm which code is being executed by the mediator, and the rest of the enclave's security mechanisms ensure that the mediator's execution is protected.

SGXDedup [4] is a variant of DupLESS that uses SGX to avoid the heavier cryptographic protocols of DupLESS. Despite introducing several improvements over DupLESS, SGXDedup suffers from the same vulnerabilities as the system on which it was based, in particular with regard to frequency analysis attacks.

SGX Enabled Secure Deduplication (S2Dedup) [3] is also based on TEEs, but keeps a table with the frequencies of chunks inside the enclave, in order to ensure that a key is not reused more than a predetermined number of times for the same content. A limitation of S2Dedup is that, due to the limited memory of the enclave, it cannot keep a complete history of the access frequency of all objects. In this way, it is forced to reset its frequency table, changing keys more often than strictly necessary, limiting the efficiency of deduplication.

4 FH-Dedup

This section describes the architecture and implementation of FH-Dedup. Our system is inspired by previous work and is based on the realization of a secure mediator, running in a server enclave.

4.1 Threat Model

We make the following assumptions:

1. The integrity and confidentiality of the data sent through the communication channel between the client and server are protected by the use of cryptographic operations;
2. Communication between the client and the enclave is secure;
3. There is no collusion between the client and the storage server;
4. The recipe files (explained in more detail ahead) are stored securely, as they are protected with the client's key;
5. The enclave guarantees confidentiality and integrity of the code and data held therein.

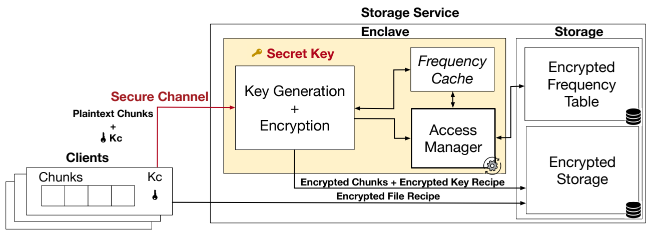
We consider an honest but curious attacker, that is, one who does not change the system protocol but who has access to a set of auxiliary data and observes the access frequency distribution of the hashes of the chunks.

This attacker aims to identify the original content of encrypted chunks on the server by observing the accesses in the server (comparing the frequency of accesses of a given entry to the frequency of a given chunk hash), the accesses to the data itself, and the accesses to the metadata of the system, specifically the entry point for the encrypted table stored in the untrusted area on the server (presented in the following paragraphs).

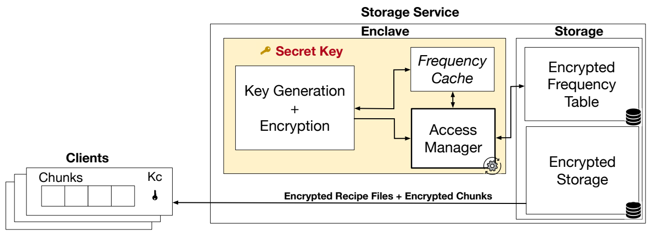
4.2 Components and Interactions

The architecture of FH-Dedup can be seen in Figure 1. It has 3 main components: *Clients*, which hold the chunks (in plaintext) that will be later sent and stored in the server. The *Server* (or SS), which contain the encrypted data of the clients, and the *Enclave* that runs in the server.

Similarly to the related work, we consider that clients can write and read their data as they wish. This is done by establishing a secure connection with the enclave present on the



(a) Write/Send data operation.



(b) Read data operation.

Figure 1. FH-Dedup server access protocol and components

SS, which is responsible for encrypting the data and placing it in encrypted storage. The enclave will choose the key to encrypt the data, taking into account the frequency of each of the chunks. Next, we describe the operations of sending and reading a file on FH-Dedup.

The data *write/send* operation is represented in Figure 1a. A client who wants to send a file *F*, first divides the file into several chunks, and builds a *recipe file* based on the order of the chunks in the file. The size of the chunks is a system-wide constant, e.g., 4KB. The client then encrypts the recipe file with the client key *Kc* (unique per client) and sends it to the storage server. The client then sends all the chunks and the client key, through the secure communication channel with the enclave. The enclave generates a key for each chunk based on its frequency, a parameter *t* that indicates the maximum number of equal copies that can be encrypted with the same key, and a secret known only to the enclave, and encrypts each chunk with its respective key. After generating all cryptograms, the enclave builds a key recipe, which contains the keys for all the chunks that were encrypted. The enclave encrypts this recipe with the client key *Kc*, and stores the encrypted chunks and the encrypted recipe file in the SS.

To *read* a file (see Figure 1b), a client retrieves the encrypted file recipe and the encrypted key recipe from the storage system, as well as the encrypted chunks. Then it decrypts the recipe files with the *Kc* client key. Finally, the client decrypts each encrypted chunk using the respective key and reconstructs the file based on the file recipe.

To prevent frequency attacks, it is necessary to store a cache in the enclave with the frequency of each chunk, that is, the number of times it has already been written. Due to the memory limit of the enclaves (128 MB) [16], it is not possible to store the frequency of all chunks within the enclave cache, e.g., using SHA-256 it is possible to obtain 2^{256} different hashes.

With 100MB available for the enclave cache, it is only possible to store approximately 2M entries in the enclave, as each entry occupies 42 bytes (including the hash, a frequency counter, and a reference to the respective entry in the frequency table). In real storage systems, these 2M entries can represent 5% of the number of existing chunks.

This type of cache was already present in S2Dedup [3], but, in that system, when the enclave memory limit is reached, the table is reset, preventing the use of deduplication for the chunks from which information was lost. To overcome this problem, our system maintains the frequency of all chunks even when the memory limit of the enclave is reached. The server contains a second encrypted frequency table, outside the enclave, whose contents can only be deciphered by the enclave. The enclave will consult this table (as seen in Figure 1a) when the maximum memory is reached and it is not possible to store the frequency of new chunks. A problem that can arise when using a table in an untrusted zone is that the server can observe the access of the enclave to the entries of this table, and thus also infer the contents through the frequency of accesses [17]. To mitigate this problem, our solution includes a new component within the enclave, called *Access Manager*. This component implements access dispersion policies and cache eviction policies to mask the frequency of accesses to each entry.

4.3 Fault Tolerance

Since our system was designed for cloud storage, failures in data centers may affect the functioning of our system. In our design, some information is directly dependent on the enclave and can be lost if the enclave fails. In such a situation, clients would continue to be able to read their data, as reads do not depend on interactions with the the enclave. However, our frequency cache and the secret key used to encrypt the external table would become unavailable. A possible solution is to synchronize multiple enclaves to maintain a copy of this secret key and also replicate the frequency table. The synchronization of this table may not be trivial, since the frequencies may diverge during some time period resulting in different choices for when to perform deduplication inside the enclave.

4.4 Privacy Guarantees

To quantify the level of privacy that our *Access Manager* component is able to offer to our table in the untrusted zone, we decided to follow a variant of the privacy preservation criterion used in PraDa [18], called *α-privacy*. This criterion ensures that there are always at least $\frac{1}{\alpha}$ chunks with the same frequency. In our work, we consider a generalization of this criterion, called *(α, δ)-privacy*, which ensures that, given a chunk accessed with a frequency *f*, there are always at least $\frac{1}{\alpha}$ chunks that are accessed with a frequency in the interval $[f - \delta, f + \delta]$. Note that when $\delta = 0$ the *(α, δ)-privacy* is equivalent to *α-privacy*.

Figure 2 shows an example of the frequency distribution of a data set; this data set contains 5 chunks, some of the chunks share the same frequencies, while others do not.

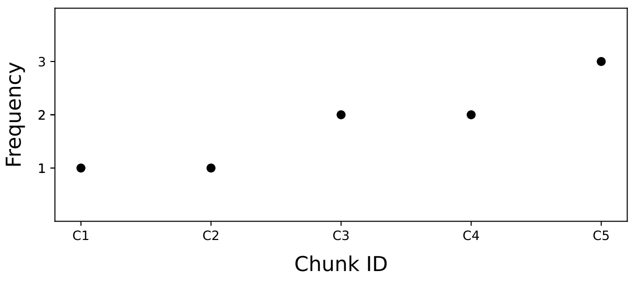


Figure 2. Example dataset frequency distribution.

When applying our criterion with $\delta = 0$, the value is $\frac{1}{\alpha}$ equals 1, since for each chunk, there is at most 1 chunk with a difference of 0 in frequency. When $\delta = 1$, $\frac{1}{\alpha}$ is equal to 2, since there are at most 2 chunks with a max difference of 1 in frequency. And with $\delta = 2$, $\frac{1}{\alpha}$ is equal to 5, since all chunks have a maximum difference of 2 in frequency.

This generalization allows for capturing scenarios where the frequency of a given chunk cannot be accurately estimated.

4.5 Implementation

In this section, we present the implementation of FH-Dedup. We start by discussing the technologies used to develop our prototype, then we move to the implementation details of the write and read operations, and finally we discuss the implementation of the *Access Manager* component.

4.5.1 Technologies. Our prototype is based on the S2Dedup code base. This simplified the development and let us focus on the most important aspects of our solution. The prototype is implemented in C and takes advantage of both Intel SGX and the user space Software Performance Development Kit (SPDK) [19, 20], a framework that provides a set of tools and libraries for writing high-performance, scalable, user-mode storage applications.

Our deduplication engine is also implemented as an SPDK virtual block device. It intercepts incoming block I/O requests, performs secure deduplication with user-specified fixed chunk size, and then forwards the request to an NVMe block device or another virtual processing layer depending on the target SPDK deployment. Using NVMe drivers in SPDK is valuable since it provides zero-copy, highly parallel access directly to an SSD from a user-space application.

These requests eventually reach the NVMe driver and storage device, unless intermediate processing removes such a need (for example, repeated writes). In addition, SPDK provides a set of storage protocols that can be stacked on top of the block device abstraction layer. Among them, our work is also implemented using Internet Small Computer System Interface (iSCSI) targets, allowing clients to remotely access storage servers.

4.5.2 Write Operations.

When a client wishes to write data to the remote storage server, it begins the process by encrypting each chunk with a key established between the client and the enclave, using a symmetric encryption scheme, in this case, the standard AES scheme with 256-bit keys in XTS mode. XTS was designed as a more robust alternative to other available block cipher modes such as CBC. We choose it because it is a length-preserving scheme (i.e., the ciphertext has the same length as the plaintext), and does not apply chaining, thus supporting random access to encrypted data [21]. We encrypt the data to protect it against attackers eavesdropping on the communication channels.

Next, the encrypted data is sent over the network through the iSCSI protocol. The server sends the data to the enclave, which will first decipher the data with the shared key with the client. It will then generate the chunk encryption key using TED's probabilistic scheme, using both the enclave secret and the chunk frequency. The frequency of a chunk can be obtained from the enclave cache or by using the *Access Manager* component, with the policies that we describe below. After generating the key and encrypting the data, the enclave writes the data directly to storage. The enclave also encrypts the chunk encryption key with the shared client key; in order to decrypt the chunk later, this encrypted key is stored inside the server also.

4.5.3 Read Operations.

When the client retrieves data from the remote server, the server will fetch the encrypted chunks and send them back to the client. For each encrypted chunk, the client will contact the server to retrieve the key (in encrypted form) to decipher the chunk. The client decrypts the key with its shared client key and with the resulting key decrypts the data. This way the enclave is not present in the read operations and no re-encryption of data is done as in S2Dedup. The read performance comparison between our solution and S2Dedup is also present in Section 5.

4.5.4 Access Dispersion Policies.

As mentioned in the previous section, our strategy to prevent frequency analysis attacks lies in the *Access Manager* mediating the interaction between the frequency cache residing in the enclave and the encrypted frequency table stored outside the enclave. Its operation is based on the combination of an access dispersion mechanism that is applied whenever the enclave accesses the encrypted frequency table that is in the untrusted zone, and a cache eviction mechanism, which is used to replace entries in the cache when it reaches the maximum size. Next, we describe some of the policies that can be implemented by these mechanisms.

The information exchange mechanism between the enclave cache and the encrypted table on the server can be performed by following several approaches. We consider three access dispersion policies:

- *Direct Access*: The *Access Manager* directly accesses the entry present in the table. This approach has good computational and storage performance, as only the desired server entry is accessed;
- *K-Anonimity Access*: The *Access Manager* queries K entries (the desired entry and $K-1$ random), in random order. This allows for changing the frequency of accesses, causing infrequent chunks to be consulted more often;
- *Bucket Access*: It consists of storing server entries in buckets. To retrieve the desired entry from the server, the *Access Manager* indicates its bucket, not its hash. In this way, the server will send all the entries present in that bucket and will not be able to understand which entry was requested. The bucket of a given entry is given by:

$$Hashcode(h) \mod N \tag{1}$$

  where N is the number of buckets, and *Hashcode* is a function that returns an integer given an hash h (e.g., $h[0]*31^{n-1}+h[1]*31^{n-2}+...+h[n-1]$, where *h[0]* is the first character of h, and n is the number of characters). The result identifies the bucket in which the new entry will be placed. This scheme assumes a uniform distribution of entries by buckets and allows chunks with less frequent accesses to share the same bucket with chunks with frequent accesses, thus increasing the access frequency of the former.

4.5.5 Cache Eviction Policies.

When it is necessary to store the frequency of a chunk but the enclave cache has already reached its maximum capacity, it is necessary to evict one of the entries, to store the new one inside the enclave. For this purpose, we consider 4 eviction policies:

- *Random*: Evict a random element present in the enclave cache;
- *Least Recently Used* (LRU): Evict the element that was accessed the longest in the enclave cache;
- *Less Frequent*: Evict the element with the lowest frequency present in the enclave cache;
- *Least Accessed Externally*: Evict the element with the least access to the encrypted table on the server that is present in the enclave cache.

4.5.6 Policies Combination.

The different access dispersion policies can be combined with the different cache eviction policies, allowing 12 different modes of operation of the *Access Manager* component. The 12 possible combinations were evaluated, and the results obtained are presented in Section 5.

5 Evaluation

In this section, we evaluate the storage savings provided by our solution, analyze the privacy guarantees in the access to the encrypted table on the server, and evaluate the performance of read and write operations.

5.1 Storage Saving and Privacy Guarantees

In order to evaluate the frequency distribution of access to the server's encrypted table, we created a dataset susceptible to frequency analysis, i.e., a dataset that contains a number of

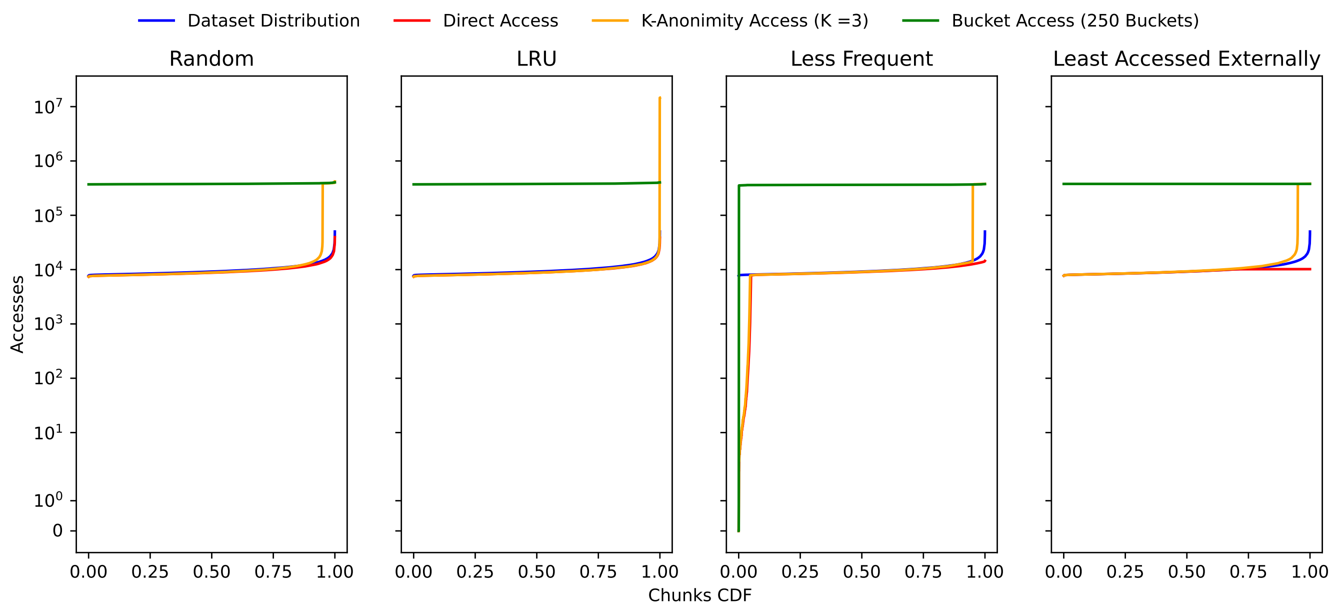


Figure 3. Distribution of accesses to the server's encrypted table with different cache policies for each approach.

chunks with a high frequency. This dataset was obtained using a data generator that follows a Zipfian distribution (see Figure 4) with 10K distinct chunks and 100M accesses. The cache present in the enclave can store up to 500 entries, meaning it can only store 5% of the frequencies of the chunks in the enclave (green area).

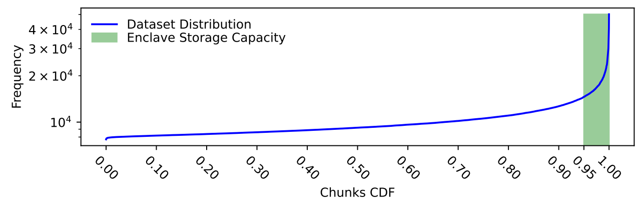


Figure 4. Frequency distribution of the chunks in the data set.

5.1.1 Storage Savings Analysis. The purpose of having an external frequency table is to store the frequencies of all chunks, maximizing the deduplication gain. To verify the deduplication gain, we simulated the storage cost when the S2Dedup solution is used (with $t = 350$, when the frequency of a chunk reaches t a new key is used to encrypt it [15]), and our solution (with $t = 350$). The results are shown in Table 1. The analyzed dataset has approximately 381.5 GB (4KB size chunks). These simulations were done by varying the cache size, meaning, the enclave storage capacity.

As S2Dedup needs to restart the frequency table within the enclave whenever it reaches its maximum size, S2Dedup is limited by the size of the enclave table to offer storage savings through deduplication. On the other hand, FH-Dedup takes advantage of a table in the untrusted zone, overcoming this limitation, and thus being able to take full advantage of chunk deduplication. As can be seen in Table 1 our solution always offers maximum storage savings, while S2Dedup has a hard time applying deduplication the smaller the enclave memory. It can only save 2.7% of storage when the enclave can keep only 5% of the entries.

Table 1. Storage savings in S2Dedup and in our solution.

Percentage of entries that can be stored in the enclave		5%	50%	95%	100%
Storage Savings	S2Dedup	2.7%	29.3%	70.1%	99.74%
	FH-Dedup	99.74%	99.74%	99.74%	99.74%

5.1.2 Accesses to Untrusted Storage Analysis. To evaluate the information that a malicious server can obtain by observing the accesses to its encrypted table, the execution of each of the combined policies for the dataset presented above was simulated. In addition to the type of access policy used to query the server table, we wanted to understand whether cache eviction policies somehow influence the number of accesses. Figure 3 shows the results obtained.

The frequency distribution for the dataset analyzed is shown in blue (the same as shown in Figure 4), to represent the effect of policies on access frequency. The distribution of accesses to the encrypted table on the server, using the strategies *Direct Access*, *K-Anonimity Access* and *Bucket Access*, is represented in red, orange, and green, respectively.

When the *Random* cache eviction policy is used, the strategies *Direct Access* and *K-Anonimity Access* show the same type of accesses as the original distribution, however, the accesses are more accented using *K-Anonimity Access*, as the number of hits becomes greater than the original. The access

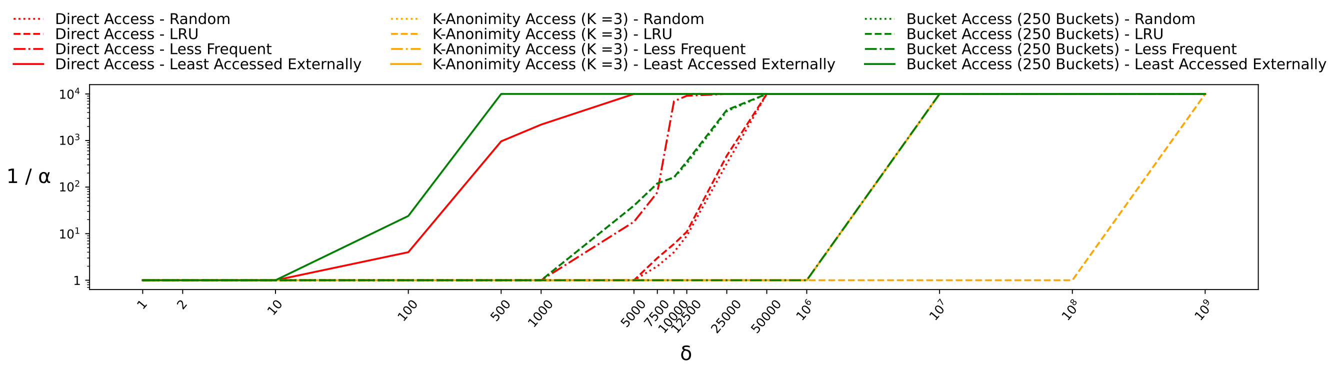


Figure 5. Privacy guarantees offered by each combination.

policy *Bucket Access* manages to change the frequency of accesses of chunks but increases the number of accesses.

For strategies *Direct Access* and *K-Anonimity Access*, the LRU policy does not properly manage the elements that are evicted, as it only takes into account the order in which the chunks are accessed. The access policy *Bucket Access* maintains the previous behavior.

The eviction policy *Less Frequent* ensures that the most frequent elements are kept within the enclave; however, the less frequent elements will always be evicted, so when they are needed, it will be necessary to retrieve their counter from the frequency table outside the enclave. With access policies *Direct Access* and *K-Anonimity Access*, this reduces the number of accesses to some elements. The access policy *Bucket Access* shows a reduction in the number of accesses, but this number continues to be much higher than the other approaches.

Finally, the eviction policy *Least Accessed Externally* is the one that best normalizes the frequency of access to each element in the server table. This can be observed especially with strategies *Direct Access* and *Bucket Access*, which are closer to a straight line. This effect is not easy to achieve with the *K-Anonimity Access* approach because, despite dumping the least accessed elements in the untrusted table, their accesses are random, causing some elements to continue to be accessed many more times.

5.1.3 Privacy Guarantees Analysis. In order to evaluate the (α, δ)-*privacy* offered by each combination, the criterion presented in Section 4.4 was applied to the dataset for different values of δ. The results are shown in Figure 5.

The calculation of α was performed using the function *f(X,δ)*, which calculates α for a given δ in a dataset X. Note that the value of α falls between $1/N$ and 1, where N is the number of distinct chunks. The higher the value of $1/\alpha$ and the lower the value of δ, the greater the privacy offered by the combination.

The combination of the *Bucket Access* access policy with the *Least Accessed Externally* eviction policy has the best (α, δ)-*privacy*. With δ equal to 100, we have $\alpha = 1/24$. Given that $\frac{1}{\alpha} = 24$, this means that for each chunk we have 24 other chunks with a maximum difference of 100 in frequency. With δ equal to 500, all 10K chunks have a maximum frequency difference of 500. The second best combination is *Direct Access* with the *Least Accessed Externally* cache eviction policy: with δ equal to 100, $\frac{1}{\alpha}$ equals 4; with δ equal to 500, $\frac{1}{\alpha}$ equals 960; with δ equal to 1K, $\frac{1}{\alpha}$ equals 2197; and with δ equal to 5K, $\frac{1}{\alpha}$ is 10K.

The other combinations are able to achieve $\frac{1}{\alpha}$ equal to 10K but for much larger values of δ, making it easier to distinguish the chunks between them, making the use of these combinations less secure.

The other combinations are able to achieve 10k(α, δ)-*privacy* but for much larger values of δ, making it easier to distinguish the chunks between them, making the use of these combinations less secure.

5.2 Performance Evaluation

To evaluate the performance of FH-Dedup, we carried out experiments to understand the impact of the use of the *Access Manager* on the performance of write operations and the intervention of the enclave in read operations. Our testbed consists of an Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz with 16GB of RAM and SGX.

5.2.1 Write Operations. Our solution allows us to maintain deduplication gains while providing privacy guarantees: however, to maintain this effect, the *Access Manager* has to interact with the encrypted frequency table of the server to store information about chunks that it cannot store in the enclave cache.

To evaluate the impact of the encrypted table in an untrusted environment, we performed multiple write operations with 4KB chunks, more specifically we only compare the cost of enforcing each policy. By this, we mean that we measure operations that include hash calculations, OCALLs, and table accesses since this is where our policies and S2Dedup differ. For these write operations, we exclude operations that require chunk encryption or memory copies, since such operations are equal in any policy. We made this choice to better understand the performance overhead introduced by each combination of policies.

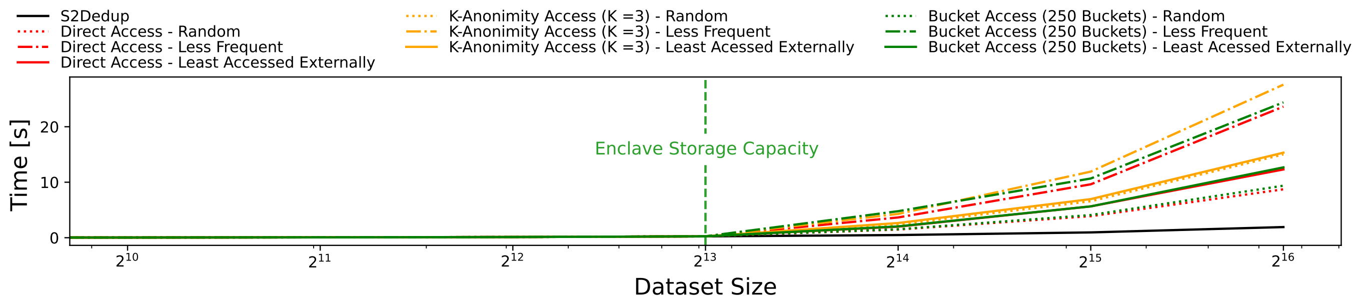


Figure 6. Cost to enforce each policy for a new chunk.

We compared each of our access combinations (except LRU, not implemented) with S2Dedup; the results are shown in Figure 6.

In this experiment, we use a cache inside the enclave that supports 8196 entries and gradually increases the dataset size, composed only of write operations, which are those that make the enclave interact with the external frequency table. Another important aspect is that we tested the worst case where all writes consist of distinct chunks, which causes FH-Dedup to interact with the server's encrypted frequency table every time the cache is full and a new chunk is written.

While the number of writes is up to 8196 (2^{13} in the figure), our solution behaves exactly like S2Dedup, i.e., information about chunks is written directly to the enclave cache. When the number of writes is greater than 8196, communication with the server is required since the cache no longer allows for the storage of more information.

There are two factors in FH-Dedup that increase overhead. The first is the access policy, which can contribute to a larger number of OCALLs and memory allocation; the second one is the cache eviction policy, which can cause evictions from *Access Manager* to take a longer period of time.

The *Direct Access* policy requires OCALLs to retrieve the desired entry from the server frequency table, and an OCALL to write the new (or updated) information about the evicted entry.

The *K-Anonimity Access* policy requires the same number of OCALLs as the *Direct Access* policy multiplied by *K*, we will consult *K* entries in the server, and evict one entry.

Finally, the *Bucket Access* policy retrieves all the entries in the desired bucket, since we ask for information from the server by sending a bucket identifier and not a hash through an OCALL. It scans all elements until it finds the desired entry and decrypts its counter. Later, it will evict one entry from the enclave cache.

For all the policies, we also need to allocate memory for the information to be returned from the server, and we need to decrypt the desired entries counters.

We can see in the figure that the random cache eviction strategy is the least expensive. This happens because enclave entries do not need to be sorted, we just generate a random value in the range $[0, 8196]$ and select the entry in the generated index.

Other cache eviction strategies are more expensive because they require entries to be sorted by frequency or external hit counts in the server frequency table.

The least external access strategy outperforms the least frequency strategy because the sorting process takes less time. This happens because external accesses to server frequency table entries do not change while they are in the enclave cache, as opposed to an entry frequency, which gets updated for elements inside the cache.

K-Anonimity Access is the slowest access strategy due to our implementation. Every time a new entry needs to be added, it needs to pick *K*-1 items from the enclave cache in a random fashion, so it must first create an array with all the elements of the cache, and then select these random elements. Later, when evicting an entry, it must pick another random element to evict, so it must bring all cache elements again into an array and then randomly select one element. The other access strategies only require this last step.

The implementation of the *K-Anonimity Access* policy could be improved using other data structures, however, this improvement could not be implemented in time.

5.2.2 Read Operations.

To observe the performance difference of read operations in S2Dedup and FH-Dedup, we performed several read operation experiments, we varied the number of 4KB chunks read and measured the amount of time required to read and decrypt these chunks. We also use the same type of symmetric cipher as S2Dedup, AES-256-XTS. The results are shown in Figure 7.

From the results obtained, we conclude that FH-Dedup, on average, performs read operations 5.38 times faster than S2Dedup.

To cross-check our results are trustworthy, we tested the speed of this cipher suite using OpenSSL's *speed* command. It takes 3 seconds to encrypt/decrypt 3,147,723 4KB chunks. This means that it can encrypt/decrypt an average of 1,049,241 4KB chunks per second. FH-Dedup takes approximately 2.25 seconds to read and decrypt 1,048,576 4KB chunks, making the result credible.

In S2Dedup, every time a client requests an encrypted chunk, this encrypted chunk must be deciphered in the enclave using the enclave key, and ciphered with the client key, later, the client will decipher the encrypted chunk with its key, finally retrieving the original plaintext chunk. This means

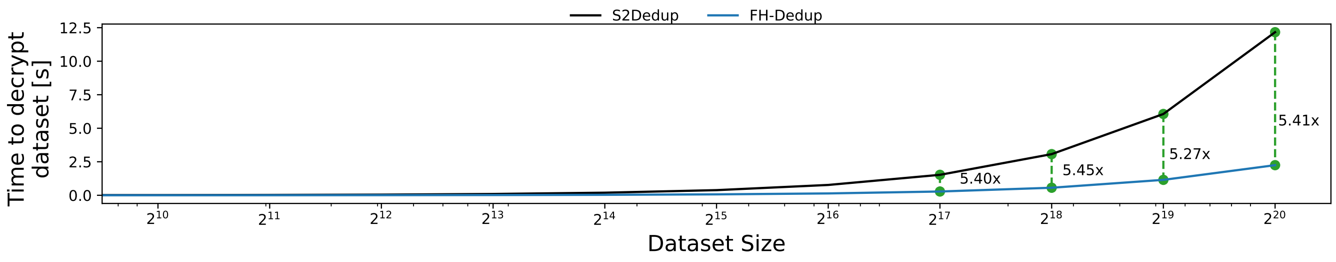


Figure 7. Time cost to read and decrypt different datasets.

that every time the client requires the same chunk, it will be necessary to perform two decryption operations and one encryption operation on chunks.

In FH-Dedup there is no need to communicate with the enclave; the client retrieves the encrypted key recipe from the server and deciphers it with its key, and from the key recipe, the client can retrieve the key needed to decipher the chunk. Due to this approach, there are only two decryption operations, one necessary to obtain the key for decryption and one to decrypt the chunk.

This makes FH-Dedup reads more performant and scalable than S2Dedup, removing bottlenecks in the enclave during reads. Typically, reads have a 70/30 ratio compared to writes, making them the most common operations in memory systems [22].

5.3 Discussion

It is observed that the access policy *Bucket Access* is the one that manages to keep the access frequencies of the chunks close to each other, regardless of the cache eviction policy used, making it more difficult to distinguish the chunks. This happens because less frequent chunks can be placed in the same bucket as more frequent chunks, changing their access frequency, and making it more difficult to perform frequency analysis attacks. However, this access policy is computationally more expensive than the *Direct Access* policy, and also increases the number of accesses by more than an order of magnitude. With the *Least Accessed Externally* policy, the *Direct Access* access policy is also able to keep the access frequencies of the chunks close to each other, but with a lower number of accesses. However, using this eviction policy implies higher memory usage within the enclave, which leads to a smaller number of entries that can be stored in the enclave. We can also observe that FH-Dedup creates overhead when it must retrieve information from the encrypted table during write operations; nevertheless, in the most common operations, the read operations, we achieve faster reads than systems that require the enclave interaction, specifically, we, are on average 5.38 times faster than S2Dedup.

6 Conclusions

With the rapid growth of data, cloud storage systems have become popular. To allow the storage of duplicate data without compromising storage efficiency, new techniques were introduced.

Data deduplication is a technique that reduces the amount of redundant data held in a cloud storage service. If the data is already stored, a pointer to that copy of the data is created, rather than storing additional copies of the data.

To address users' privacy, encrypted deduplication was created, combining deduplication with privacy, allowing users to store data encrypted while allowing storage systems to apply deduplication.

Several attacks can occur in these types of systems, one of them, being frequency analysis, which aims to find out which data is being stored encrypted by users.

In this work we presented an Intel SGX-based solution that is capable of supporting deduplication while maintaining strong levels of privacy, preventing frequency analysis attacks. In this context, we propose and compare different combinations of cache strategies and policies to prevent frequency analysis attacks.

Our evaluation demonstrates that the combination of different caching strategies and policies influences the privacy guarantees and the number of accesses to the server (encrypted) frequency table. Our performance evaluation also let us understand the overhead added by our solution in write operations when new elements cannot be stored inside the enclave cache, nonetheless, our read operations are faster than the state-of-the-art systems, in particular, on average 5.38 times faster than S2Dedup.

Furthermore, the storage savings in our system are constant, regardless of the size of the cache present in the enclave, having greater storage savings than the most recent state-of-the-art solution, which also uses chunk frequency to apply encrypted deduplication inside an enclave. In this way, we are enabling storage efficiency, with a commitment to privacy.

6.1 System Limitations and Future Work

Our work shows that it is possible to use TEEs as secure mediators in encrypted deduplication systems while maintaining deduplication gains even with memory limitations, our solution uses an external encrypted frequency table to keep track of chunks frequencies which it can no longer store inside the enclave cache. However, our solution with the use of OCALLs and different access and cache eviction policies has a performance penalty during write operations, we consider that better access strategies and cache eviction policies should be discussed and implemented.

Our current architecture allows for one server and multiple clients, in a more realistic environment, there should be multiple servers that a client can contact to store/retrieve data. For this reason, we consider that an improvement is having multiple servers that communicate with each other the frequency of their chunks through a lazy replication communication protocol such as gossip, although this design seems simple it is not trivial, remember the chunk frequency in the server is encrypted, so the chunk frequencies must first be decrypted in the enclaves and later be updated.

Acknowledgments

This work has been developed in the context of the project NGSTORAGE, that aims to test distributed systems. In the development of the work I benefited from the discussion and collaboration with other members of the NGSTORAGE team, in particular from working with Cláudio Correia, that is addressing the security aspects of the project.

This work was supported by FCT – Fundação para a Ciência e a Tecnologia, through the grant 2020.05270.BD, the NGSTORAGE project (funded by the OE with the ref. PTDC/CCI-INF/32038/2017), and the project UIDB/50021/2020.

References

[1] Dropbox, Available at https://dropbox.com/, Accessed: 2022-10-31.

[2] Google Drive, Available at https://www.google.com/drive/, Accessed: 2022-10-31.

[3] M. Miranda, T. Esteves, B. Portela, and J. a. Paulo, "S2Dedup: SGX-enabled secure deduplication," in *International Conference on Systems and Storage*, Haifa, Israel, Jun. 2021.

[4] Y. Ren, J. Li, Z. Yang, P. P. Lee, and X. Zhang, "Accelerating encrypted deduplication via SGX," in *USENIX Annual Technical Conference*, Remotely, Jul. 2021.

[5] D. Meyer and W. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage*, vol. 7, pp. 1–20, 2012.

[6] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *International conference on File and Storage Technologies*, San Jose (CA), USA, Feb. 2012.

[7] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-locked encryption and secure deduplication," in *International Conference on the Theory and Applications of Cryptographic Techniques*, Athens, Greece, May 2013.

[8] SGX101, "Overview - SGX 101," Available at https://sgx101.gitbook.io/sgx101/sgx-bootstrap/overview, Accessed: 2022-10-31.

[9] C. Correia, M. Correia, and L. Rodrigues, "Omega: a secure event ordering service for the edge," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, Valencia, Spain, Jun. 2020.

[10] S. Keelveedhi, M. Bellare, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage," in *Security Symposium USENIX Security*, Washington (D.C.), USA, Aug. 2013.

[11] M. Naor and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions," *Journal of the ACM (JACM)*, vol. 51, pp. 231–262, 2004.

[12] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko, "The one-more-rsa-inversion problems and the security of chaum's blind signature scheme." *Journal of Cryptology*, vol. 16, p. 185–215, 2003.

[13] J. Camenisch, G. Neven, and A. Shelat, "Simulatable adaptive oblivious transfer," in *International Conference on the Theory and Applications of Cryptographic Techniques*, Barcelona, Spain, May 2007.

[14] D. Chaum, "Blind signatures for untraceable payments," in *CRYPTO*, Santa Barbara, CA, USA, Jan. 1983.

[15] J. Li, Z. Yang, Y. Ren, P. Lee, and X. Zhang, "Balancing storage efficiency and data confidentiality with tunable encrypted deduplication," in *European Conference on Computer Systems*, Heraklion, Greece, Apr. 2020.

[16] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig, "Benchmarking the second generation of intel SGX hardware," in *International Conference on Management of Data*, Philadelphia (PA), USA, Jun. 2022.

[17] M. Jurado and G. Smith, "Quantifying information leakage of deterministic encryption," in *International Conference on Cloud Computing Security Workshop*, London, United Kingdom, Nov. 2019.

[18] B. Dong, R. Liu, and W. H. Wang, "PraDa: Privacy-preserving data-deduplication-as-a-service," in *International Conference on Conference on Information and Knowledge Management*, Shanghai, China, Nov. 2014.

[19] SPDK, "Storage performance development kit," Available at https://spdk.io/, Accessed: 2022-10-31.

[20] SPDK, "Spdk github," Available at https://github.com/spdk/spdk, Accessed: 2022-10-31.

[21] M. Dworkin, "SP 800-38E. recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices," Available at https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=904691, Accessed: 2022-10-31.

[22] Live Optics, "Read / write ratio," Available at https://support.liveoptics.com/hc/en-us/articles/229590547-Live-Optics-Basics-Read-Write-Ratio, Accessed: 2022-10-31.