Deduplication vs Privacy Tradeoffs in Cloud Storage

Rodrigo Silva
rodrigo.santos.silva@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professor Luís Rodrigues and Professor Miguel Correia)

Abstract. Data deduplication is a technique that reduces the amount of redundant data that is kept in a storage system. If two or more files have equal chunks, these chunks are treated as a single unit, and not as multiple, distinct units, that need to be stored independently. Deduplication can offer significant savings, in particular, in cloud storage systems. Unfortunately, if the users of the storage system decide to encrypt the stored data (namely, for privacy reasons), it becomes hard to apply deduplication. This report surveys the main techniques that have been proposed to conciliate deduplication and the possibility of storing data in encrypted form, in a manner that can ensure data confidentiality for the clients. Inspired by these works, we propose an architecture that leverages trusted computing environments to increase the performance of encrypted deduplication systems.

Table of Contents

1 Introduction

It has been observed that, in systems that store large amounts of data from multiple users, such as Dropbox [1] and Google Drive [2], it is possible to find large amounts of repeated data. For instance, when users store program sources in the cloud, there are many libraries that may be shared by several projects. Similarly, when users backup their computers, different users are likely to backup the same files, such as software binaries, music files, among others. *Data Deduplication* is a technique that allows to identify files that are identical, or that have identical pieces (also known as *chunks*), and treat all copies of the same chunk as a single unit. In this way, the system needs to maintain just a few copies of each unique chunk, enough to provide fault tolerance, regardless of the number of users that store that chunk.

Unfortunately, to apply data deduplication to encrypted data is hard. If the users encrypt their data before storing it, and use different keys to perform encryption, the same chunk will result in different encrypted versions. This prevents the storage system from identifying which data is redundant and, therefore, from applying data deduplication. *Encrypted Deduplication* is the name given to techniques that attempt to combine file encryption and data deduplication. Typically, this combination requires some form of direct or indirect coordination among different users, to ensure that identical chunks are encrypted with the same keys. The challenge is to perform this coordination in a manner that is both efficient and allows to preserve the confidentiality of the information stored by each user. Our take on confidentiality is the same as the one in the literature, i.e., "*... protect outsourced storage from the unauthorized access by malicious users or even the cloud providers that host the outsourcing services*" [3].

This report surveys the main techniques that have been proposed to conciliate deduplication and the possibility of storing data in an encrypted form, in a manner that can ensure data confidentiality for the clients. Inspired by these works, we propose an architecture that leverages trusted computing environments to increase the performance of encrypted deduplication systems.

The rest of the report is organized as follows. Section 2 presents the goals and the expected results of our work. Section 3 introduces the main concepts relevant for our work and Section 4 describes the main techniques to implement encrypted deduplication. Section 5 describes the proposed architecture to be implemented and Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of implementing encrypted deduplication, while ensuring good performance, and protection against privacy attacks. Namely:

> *Goals:* This work aims at leveraging trusted executed environments to improve the efficiency and robustness of encrypted deduplication systems.

To achieve this goal, we will assume that the storage provider has computers equipped with Intel processors with the Software Guard Extensions (SGX). When storing data, the client will interact with trusted software running on the enclave. The trusted component is in charge of selecting the keys used to encrypt each chunk, such that data deduplication can be implemented, while ensuring resilience to several attacks, including frequency analysis.

The project is expected to produce the following results.

> *Expected results:* The work will produce i) a specification of an encrypted deduplication algorithm; ii) an implementation of the algorithm for the Intel SGX architecture, iii) an extensive experimental evaluation using, for example, a disk I/O block-based benchmark for deduplication systems, such as DEDISbench [4, 5].

3 Background

In this section, we introduce concepts relevant for our work. We start by introducing deduplication (Section 3.1) and encrypted deduplication (Section 3.2), then we list the main attacks against encrypted deduplication (Section 3.3), and finally we introduce Trusted Execution Environment (TEE)s, and their security mechanisms and properties (Section 3.4).

3.1 Data Deduplication

Data deduplication is a high-coarse grained compression technique that prevents the same content from being stored with a much larger redundancy degree than strictly needed, since systems only need to store enough copies to provide fault tolerance, and create pointers to the content. Deduplication identifies duplicated data at the granularity of files or a portion of files, denoted as file *chunks*. If two or more files, from the same or different users, share a given chunk, the system needs to maintain just a few copies of that chunk, enough to provide fault tolerance, regardless of the number of files that include it.

Unlike traditional compression techniques that eliminate redundancies within a file or a small group of files (usually stored in the same operation), data deduplication aims to eliminate large data sets stored at different times by uncoordinated users and activities [6].

The effectiveness of deduplication is measured by the deduplication gain. The deduplication gain is defined as the number of duplicates that are actually eliminated, thereby reducing the storage space required to store some piece of data. Studies have shown that data deduplication can achieve significant storage savings in a production environment, for example, saving 50% on primary storage [7] and up to 98% on backup storage [8].

Another way to save storage space using deduplication is the use of *cross-user* deduplication. This means that each file or chunk is compared with the data of other users, and if the same copy is already on the server, data deduplication is performed.

This method is popular because it saves storage and bandwidth, not only when a single user has multiple copies of the same data, but also when different users store copies of the same data.

3.1.1 Chunks

The unit of data deduplication is called the chunk. The size of the chunk and the technique used to divide data into chunks affects the performance of the system [6].

One of the most straightforward manners to divide data into chunks is to consider file boundaries [9], which is known as whole file chunking. Another simple approach consists in dividing files into fixed-sized chunks(e.g., 4-8 KB each). Unfortunately, none of these approaches is very effective at producing chunks that can be deduplicated. For instance, consider two files, where one file only differs from the other by a single, additional, byte placed at the beginning of the file. Despite that most of their content is the same, the two files are different and their fixed-sized chunks would also be all different. This is known as the boundary-shifting problem [10].

Several algorithms to divide files into chunks have been proposed in the literature [11–13]. These techniques are orthogonal to our work.

3.1.2 Deduplication Approaches

To achieve deduplication, one should match existing stored data with the data the client wants to store, if there is a match, deduplication can be performed. This can either happen at the client-side or at the server-side.

In *source-based deduplication* the client is responsible for checking if the data can be deduplicated before sending it to the server. For this purpose, the client queries the Storage Service (SS) about existing duplicated data, and only uploads the missing data to the SS. A challenge in this technique is that the client needs to prove that it owns the data before being given access to the deduplicated copy. Also, this approach leaks information to clients, regarding the existence of other clients with the same data.

In *target-based deduplication* the client always sends the data to the SS, which is responsible for performing deduplication. This prevents the client from extracting information about which files are deduplicated, and also allows the SS to trivially verify that the client owns the data. However, it does not prevent deduplicated data to consume network resources.

3.2 Encrypted Deduplication

Encrypted deduplication augments deduplication with support for data encryption. Typically, not only the data is stored in encrypted form by the SS, but the SS also has no access to the plaintext and cannot infer the content that is being stored. Support for encrypted deduplication is relevant because often the data is confidential and the clients want to keep information regarding data ownership private.

The challenge of implementing encrypted deduplication is to design schemes that will result in having the same chunks encrypted with the same keys without violating confidentiality. Notice that, if each client encrypts the chunks independently, without any sort of coordination with other clients, the same chunk from different clients would result in different encrypted chunks, preventing cross-user deduplication.

3.3 Attacks

Deduplication is more effective when it is applied across multiple users (cross-user deduplication). However, this approach has serious privacy implications, which open the doors for many attacks with the intent of getting insight about the system and the content stored by users.

3.3.1 Brute Force If each user encrypts its own chunks with a different key, only known to that user, cross-user deduplication becomes almost impossible. Thus, as we will see later in the text, approaches that support cross-user deduplication have to make sure that the same files are encrypted with the same keys, even when stored by different users. If not done carefully, this may open the door for attacks, including brute-force attacks.

One of the simplest strategies to ensure that different clients encrypt the same file with the same key consists in deriving the encryption key deterministically from the content of the file [14]. An example of this approach is Message-Locked Encryption (MLE) [15], that we will present in detail later in the text. This technique is vulnerable to brute-force attacks, as follows:

Consider that the SS want to check if a given user stores copies of a given book. Given that the encryption key is derived from the content, the SS can obtain a copy of the book, derive the key, encrypted the book, and compare the encrypted version with the files stored by the user. This attack can be executed even if there are several editions or versions of the book, by using this strategy for all versions.

As we will discuss, several systems to support encrypted deduplication avoid the use of MLE, and use other techniques that are less vulnerable to brute-force attacks.

3.3.2 Deduplication Detection One privacy issue that presents itself when dealing with systems that provide deduplication is the identification of deduplication, this is, if deduplication occurred on the data that was sent to the SS. Though it seems harmless at first, a malicious client can discover if a given file or chunk of data was already stored. This may somehow release personal information, so it may be a privacy problem.

Most vendors will not try to hide the fact that they use data deduplication. This can be checked by reading the upload status message, check the upload speed to see if the file upload is completed in a much shorter time than the client computer upload bandwidth, or monitor network traffic and measure the amount of data transferred (this is the most common de-duplication detection method). Cross-user source based deduplication [16] is particularly vulnerable to this sort of attacks.

Suppose that the attacker wants to know information about a given user. If the attacker suspects that this user owns a certain sensitive file that is unlikely to be owned by any other user, the attacker can use deduplication to check whether this conjecture is true. All he needs to do is to send file X to the SS and check if deduplication has occurred. Dropbox [1] was vulnerable to this attack [17] [16], a malicious client could infer if other users had stored the same file previously by observing the network traffic and see the size of the data being transferred.

3.3.3 Frequency Analysis

To get insight about the data that has been stored, attackers developed techniques which allow the mapping of a plaintext to its ciphertext based on its frequency ranking. Frequency analysis is an inference attack that has been used to predict and recover plaintexts from substitution-based ciphertexts. A simple example of a substitution-based ciphertext is the Caesar Cipher [18], which replaces each letter in a given message by a letter at some fixed position down the alphabet. This type of attack has been shown to be useful for breaking deterministic encryption.

During a frequency analysis attack, an attacker has access to a set of plaintexts and a set of ciphertexts and the goal is to make a relation between each ciphertext and plaintext in both sets. To launch the attack, an attacker ranks the plaintexts and ciphertexts which he has access to by their frequency, then associates each ciphertext to the plaintext in the same frequency rank. In many cases, the attacker can then infer that the most frequent plaintext chunk maps to the most frequent ciphertext.

After a mapping between a plaintext and a ciphertext is made, there are attacks that enhance the severity of classical frequency analysis attacks by exploiting the locality of chunks [19], designated *Locality-Based* attacks. Chunk locality states that chunks are likely to re-occur together with their neighbour chunks across storage backups. A locality-based attack can occur if a plaintext chunk M of a prior backup was identified as the original plaintext chunk of a ciphertext chunk C of the latest backup, then both(left and right) neighbours of M are also likely to be the original plaintext chunks of the left and right neighbours of C, this can happen since chunk locality implies that the ordering of chunks is likely to be preserved across backups.

3.4 Trusted Execution Environments

A TEE is a secure area in the processor that allows to run code and store data isolated from the operation system and from user level processes. As the need for digital trust grows and concerns about protecting connected devices increases, TEEs become more and more important [20, 21]. The motivation for using TEEs in encrypted deduplication systems is improve performance while still maintaining security, bandwidth efficiency, and storage efficiency, by running sensitive operations in TEE.

Enclaves are a type of TEE technology that has been made available in many common Intel CPUs. The *enclaves*, provided by the SGX architecture [22], are TEEs that leverage hardware mechanisms such as *hardware secrets*, *remote attestation*, *sealed storage* and *memory encryption*. The hardware secrets are the root provisioning and root seal keys. Remote attestation is enforced for the client in order to prove to the service provider that an enclave is running a given software, inside a given CPU, with a given security level. Sealed storage is used to save secret data to untrusted media, required to persist data between reboots or failures, because the enclave state is stored in volatile memory [22]. The enclave relies on a hardware-guarded memory region called the Enclave Page Cache (EPC) for hosting any protected code and data. An EPC comprises 4KB pages, and any in-enclave application can use up to 96 MB. If an enclave has a larger size than the EPC, it encrypts unused pages and evicts them to the unprotected memory, suffering a performance penalty [23]. SGX provides two interfaces: ECALLs, used by an application to invoke enclave functionality and OCALLs, used by the enclave code to access an outside application.

4 Related Work

This section presents related work in the areas of encrypted deduplication, key-generation, privacy attacks and defenses against such attacks, as well as the use of TEEs in encrypted deduplication. These are the works that inspire our proposal to address the goals set in Section 2.

The section is organized as follows: Section 4.1 presents MLE, an approach to key generation which derives the key from the data content. Section 4.2 presents Duplicateless Encryption for Simple Storage (DupLESS), a system which employs a dedicated key server for encrypted deduplication. Section 4.3 presents the use of MinHash in encrypted deduplication to protect against frequency analysis attacks. Section 4.4 presents a system that enables tunable-encrypted deduplication, specifying the trade-off between storage efficiency and data confidentiality. Finally, Section 4.5 presents a solution that uses TEEs to offload expensive criptographic operations, and improve performance in encrypted deduplication.

4.1 MLE: Message Locking Encryption

Some encrypted deduplication approaches preserve the deduplication capability by deriving the key for encryption and decryption from the chunk content, usually via the hash or fingerprint of a given chunk. MLE [15] is an example of this approach.

The key generation algorithm in a MLE scheme maps a message M to a key K. The encryption algorithm takes two inputs, a key K and a message M, and produces as output a ciphertext C. The decryption algorithm takes as input a key K, and a ciphertext C, and produces a plaintext M, allowing for the recovery of the original plaintext. MLE also defines a tagging algorithm which maps a ciphertext C to a tag T, this tag simplifies the identification of duplicates. If $M1$

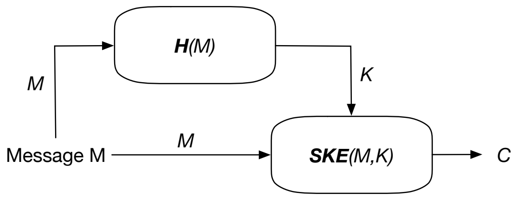

Fig. 1. Convergent Encryption

and *M2* are the same, their ciphertext will be the same, and the same tag will be generated. It is then possible to use the tags to detect duplicates, instead of comparing the entire ciphertext.

An important property of MLE is Tag Consistency (TC), that ensures the integrity of the stored ciphertext (meaning that an attacker cannot make an honest client download and recover a message different than the one that the client stored encrypted). Some MLE schemes provide a strictly stronger property, named Strong Tag Consistency (STC), that makes additionally hard to create (M, C) such that $T(C) = T(SKE(K(M), M))$ but $D(K(M), C) = \perp$, which prevents an attacker from erasing an honest client's message.

A trivial form of achieving MLE is by letting the key K be equal to the message M. However, this solution has no storage advantages since the client must store as key the entire file, resulting in no storage savings. A requirement in MLE is that keys must be shorter than messages, and ideally these should have a fixed, short length. The are several variants of MLE. Most are based on the use of a hash function H and a Symmetric-Key Encryption (SKE) scheme.

The first major variant is Convergent Encryption (CE) [15], illustrated in Figure 1, in which the key is the hash of the message M, the ciphertext is the encryption of M using K using a SKE scheme, and finally, tag T, which is generated as the hash of the ciphertext:

$$K = \mathbf{H}(M) \tag{1}$$
$$C = \mathbf{SKE}(K, M) \tag{2}$$
$$T = \mathbf{H}(C) \tag{3}$$

Another major variant is Hash and Convergent Encryption 1 (HCE1) [15]. HCE1 still generates the key K as the hash of M, as seen in Equation 1, but it generates ciphertext C as the concatenation (|| is the concatenation operator) of the encryption of M and the tag T, generated as the hash of K:

$$C = \mathbf{SKE}(K, M) \,||\, T \tag{4}$$
$$T = \mathbf{H}(K) \tag{5}$$

The reasoning behind HCE1 is to provide better performance for the server, which can retrieve the tag T from the second part of the ciphertext, so it doesn't need to compute it by hashing the ciphertext, which can be possibly long. Unfortunately, HCE1 is vulnerable to duplicate faking attacks and cannot ensure TC.

A duplicate faking attack is one where a legitimate message is undetectably replaced by a fake one, for example, when a client has already stored C, with a generated tag T, and later an attacker sends $C\acute{}$ with the same tag T, replacing the original message.

In order to achieve better performance, while still providing consistency, two new schemes were introduced, namely Hash and Convergent Encryption 2 (HCE2) [15] and Randomized Convergent Encryption (RCE) [15].

HCE2 is just as efficient as HCE1, it requires two passes through the data, one for generating the key, and a second one for encryption. HCE2 modifies HCE1 to directly include a mechanism called protection decryption, which helps it achieve TC security. The decryption routine now also checks for tags embedded in the ciphertext by recomputing the tags using the just decrypted message

RCE is even more efficient, since it only requires a single pass through the data in order to generate the key, encrypt the data, and produce the tag. RCE is able to achieve such high performance by using a random scheme, while CE, HCE1, and HCE2 were deterministic.

The RCE encryption scheme first picks a fresh random key L and computes $C = SKE(L, M)$ and $K = H(M)$ in the same pass, finally, it encrypts L with K as a one-time-pad, together with tag T, which is generated the same way as in CE:

$$L = \mathbf{RAND}() \tag{6}$$
$$C = \mathbf{SKE}(L, M) \tag{7}$$
$$K = \mathbf{H}(M) \tag{8}$$
$$C\text{`} = C \,||\, K \oplus L \,||\, T \tag{9}$$

Table 1 shows the comparison between the multiple schemes of MLE.

Table 1. MLE Schemes Comparison

Scheme	Speed	Integrity	
		TC	STC
CE	Slow	yes	yes
HCE1	Fast	no	no
HCE2	Fast	yes	no
RCE	Faster	yes	no

HCE1 does not provide tag consistency. The good news is that CE, , and RCE all implement TC security, so an attacker can't get the client to recover a different file than what he uploaded. But only CE provides STC security, which means that the server cost reduction provided by HCE1, HCE2 and RCE comes at a price, i.e. loss of STC security. This lets us conclude that there is a trade-off between performance and integrity. If we wish to have the best performance possible and still provide TC, then using RCE is the best option. If we wish to provide the integrity properties(TC and STC), then the only option is to use CE.

4.2 DupLESS

CE allows us to store data encrypted, with integrity properties, while still allowing deduplication. However, as discussed before, CE is subject to an inherent security limitation, namely, the susceptibility to brute force attacks [17].

DupLESS [17] is a system that strengthens the security of encrypted deduplication against offline brute force attacks. It deploys a dedicated Key Server (KS) for MLE key generation, separate from the SS. Deduplication is performed at file level instead of chunk level. DupLESS also introduces deduplication *heuristics*. They are used to determine whether if a file that is about to be stored on the SS should be selected for deduplication, or processed using a randomly generate key. One example can be very small files, or sensitive files that can be prevented from being deduplicated.

DupLESS was designed such that it can be integrated into existing systems, such as Dropbox [1] and Google Drive [2], as illustrated in Figure 2.

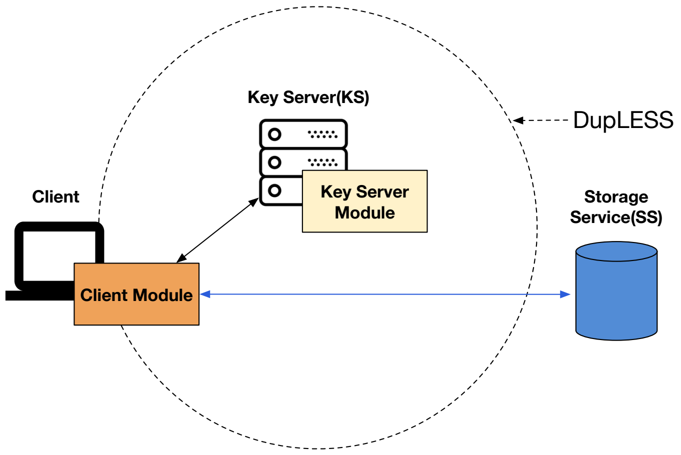

Fig. 2. DupLESS System Design

To encrypt a plaintext chunk, a client first sends the fingerprint of the plaintext chunk to the key server, which returns the MLE key via both the fingerprint and a global secret maintained only by the key server:

$$K = \mathbf{H}(k \,||\, P) \tag{10}$$

where k is the global secret owned by the KS and P is the fingerprint of a given chunk. This increases the security of MLE and still retains the deduplication properties.

The advantage of having a dedicated KS is to have a main line of defense against different types of attacks. If we want to deal with an external attackers, like a compromised SS, a possible solution could be that the KS should authenticate the clients that is speaking to. If the attacker is more powerful and resourceful and is able to compromise a client, then we are dealing with brute-force attacks from this client, but unlike CE, here the offline brute-force attacks are online, they have to involve the KS, they're going to be much slower and easier to detect.

If face of a severe attack, the KS being compromised, the attacker could be able to retrieve all the keys for all ciphertexts generated on the KS. DupLESS is not vulnerable to this problem since it uses an Oblivious Pseudorandom Function (OPRF) [24] protocol based on RSA blind-signatures [25–27] when communicating with clients; the KS learns nothing about the client input, and the client only learns K.

An OPRF protocol is something that stands between a client and a server, as shown in Figure 3, where f is the original input, for example the chunk data. The server holds a global secret, and the objective of the client is to receive the output of the OPRF, at end of the communication, the client must not know anything else than the output, and the server should not know any of the inputs given by the client.

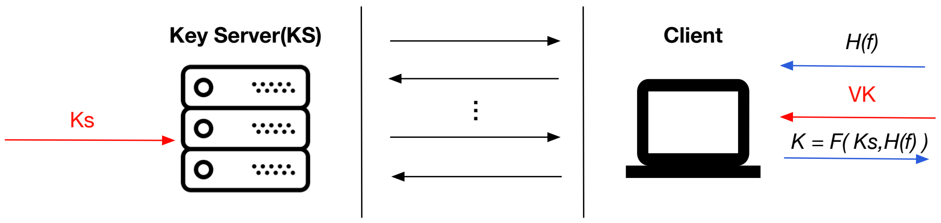

Fig. 3. OPRF Protocol

If the KS is down, overloaded or subjected to a denial-of-service attack, encryption resorts to SKE, generating a random key as the encryption key. This ensures availability but there is no deduplication during that period of time.

The significant increase in security comes at the cost of a moderate price in terms of performance and a small increase in storage requirements relative to the base system. The low performance overhead is partly due to the optimization of the client-to-KS OPRF protocol, but also to ensure that DupLESS uses fewer interactions with SS.

In short, DupLESS is a system which provides encrypted deduplication with the aid of a KS. It was the first solution to provide secure deduplication without compromising resilience.

4.3 MinHash Encryption

Most state-of-the-art encrypted deduplication systems adopt a deterministic encryption approach to encrypt each plaintext chunk with a key derived from the content of the chunk itself. However, such a deterministic approach reveals the underlying frequency distribution of the original plaintext chunks, allowing attackers to launch frequency analysis attacks against the resulting ciphertexts, and infer the content of the original plaintext chunk. MLE suffers from another security issue, which is it cannot fully protect against content leakage, because the encryption approach is deterministic.

To resist frequency analysis, the idea is to disturb the frequency ordering of ciphertext chunks. To this end, it was considered an encryption scheme called MinHash encryption, which derives an encryption key based on the smallest fingerprint on a set of adjacent chunks, so that some identical plaintext chunks can be encrypted into multiple different ciphertext chunks.

MinHash encryption works the following way. First, it groups multiple consecutive plaintext chunks into segments. We can view a segment as a set of chunks.

For each segment, it derives a key as the minimum fingerprint of all chunks in the segment, for example, comparing only the less significant bits. Then it encrypts all chunks present in the segment using that same key. In backup workloads, the segments are often similar with a large fraction of duplicated plaintext chunks, so the keys or minimum fingerprints that will be generated for similar segments are likely to be the same. In this way, most duplicated chunks are encrypted by the same key, making deduplication viable after encryption.

As for security advantages, MinHash encryption has been shown to effectively reduce the overhead of server-aided MLE since it sends only as many fingerprints as the number of segments to the KS, increasing bandwith efficiency. It can also be used to break the deterministic nature of encrypted deduplication and disturb the frequency ranking of ciphertext chunks.

MinHash is robust against the the locality-based attack, by breaking the deterministic nature of encrypted deduplication.

Still, there are deduplication issues when using MinHash encryption. Some of the same plaintext chunks may still reside in different segments, with different minimum fingerprints and different keys, so the ciphertext chunks they generate will be different and cannot be deduplicated, resulting in a slight decrease in storage efficiency.

However, such near-exact data deduplication is enough to change the overall frequency ranking of the ciphertext chunk by using different keys to encrypt a small part of the repeated chunks, thereby invalidating the frequency analysis.

Additional issues have been raised in other works [3], namely, that MinHash provides limited protection and limited configurability. As for limited protection, MinHash encryption is based on the assumption of file similarity to ensure its deduplication effectiveness, so its storage efficiency may not be suitable for general workloads. More importantly, the randomness of the smallest chunk fingerprint in the segment is limited (otherwise the deduplication effect will be lost), so MinHash encryption only slightly undermines the certainty of MLE, and does not provide security guarantees for frequency analysis.

For limited configurability, MinHash does not provide a configurable mechanism to quantify the trade-off between storage efficiency and data confidentiality. MinHash encryption disrupts the frequency ordering of ciphertext chunks by sacrificing storage efficiency (for example, repeated plaintext chunks in different segments are encrypted with different keys and cannot be deduplicated after encryption).

4.4 TED

Tunable Encrypted Deduplication (TED) [3] appeared to solve the issues presented at the end of the previous section. TED is a cryptographic primitive that provides an adjustable mechanism that allows users to balance storage efficiency and data confidentiality.

Like in DupLESS [17], TED uses a dedicated KS, which is responsible for key generation, and a SS responsible for performing deduplication. The only fully trusted component is the KS. The justification for this assumption is that the KS is deployed by companies or individuals that outsource deduplication. The SS can be external, this means that its considered trustworthy but curious.

TED is an encrypted deduplication primitive that aims to achieve *Configurability*, i.e., quantify the trade-off between storage efficiency and confidentiality of data such that information leakage is minimized. Also, TED keeps a record, Count-Min Sketch (CM-Sketch) [28], of the frequency of each chunk. There are two advantages of using CM-Sketch. First, it limits the amount of memory used to track the frequency of all chunks, and errors are bounded. Secondly, the approximate count protects chunk information from being affected by the KS, which is a security requirement in DupLESS [17].

The principle behind TED is to derive the key of each plaintext chunk based on two additional inputs: its *current frequency* f, i.e., the number of duplicate copies of M that have been upload by all clients; and the *balance parameter* t, which controls trade-off between storage efficiency and data confidentiality. The key K for M is generated by the KS in the following way:

$$K = \mathbf{H}(k \,||\, P \,||\, \lfloor f/t \rfloor) \tag{11}$$

where k is the global secret owned by the key manager, P is the chunk fingerprint, and $\lfloor f/t \rfloor$ is the maximum integer smaller than *f/t*.

f is a cumulative function, this means that it increases as more copies of the same data have been uploaded, meaning that key K will be updated as the value of $\lfloor f/t \rfloor$ increases.

Therefore, according to the value of t, copies of M will generally be encrypted with different keys. If $t = 1$, each copy of M has a different K and TED is reduced to SKE; if $t \to \infty$, all duplicates of M have the same K, and TED is reduced to MLE. Intuitively, t can be seen as the maximum number of duplicate copies of a ciphertext chunk.

To upload a file in TED (see Figure 4), the client divides the file data into chunks. It generates a key for each chunk through interaction with the KS, encrypts each chunk with the corresponding key, and then uploads the chunk to the SS. In addition, for file reconstruction, the client generates a *file recipe*, which lists the chunk fingerprint and chunk size according to the chunk order in the file, as well as a key recipe that retains the keys of all chunks. It uses the master key of each client to encrypt the *file recipe* and key recipe for protection, and uploads them to the SS together with the ciphertext chunk. The SS performs data deduplication on the ciphertext chunk. It maintains a *fingerprint index*, which is a key-value store, used to track fingerprints of physical chunks for duplicate detection. The SS does not apply deduplication to metadata, instead, it directly saves the file recipe and key recipe (in encrypted form) in physical storage.

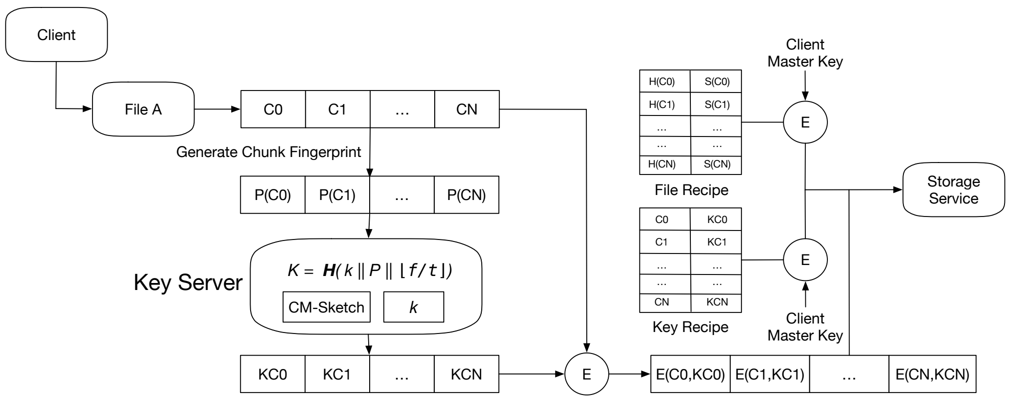

Fig. 4. TED File Upload

To download a file (Figure 5), the client first retrieves the file recipe and key recipe from the SS, and uses its master key to decrypt them. Then it retrieves the ciphertext chunks from the SS according to the file recipe and decrypts them with the key stored in the key recipe.

The key generation scheme in Equation 11 raises a security problem. For the same file with the same chunk sequence, Equation 11 will return the same keys, which also result in the same ciphertext chunk sequence, allowing the attacker to infer whether the two encrypted files are initially the same. Therefore, the key generation must generate different ciphertext chunk sequences for the same file while maintaining the effectiveness of deduplication.

To solve this issue, TED implements a probabilistic key generation method that can non-deterministically encrypt the same file (with the same plaintext

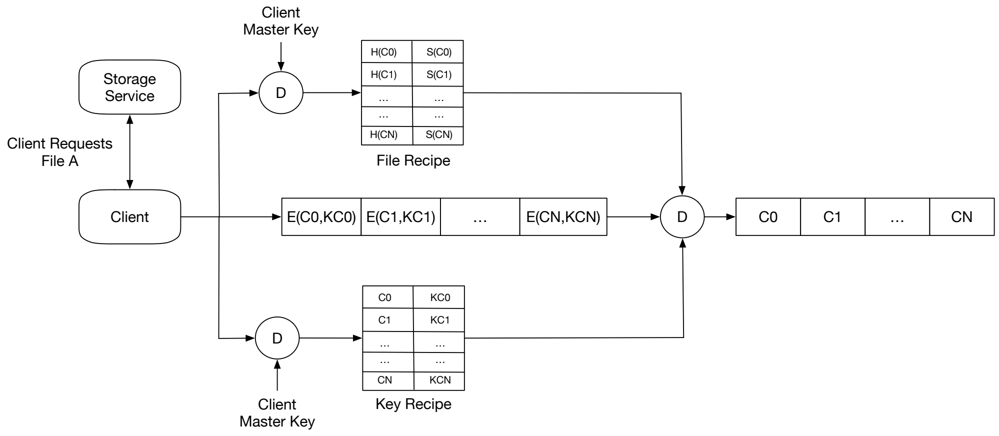

Fig. 5. TED File Download

chunk sequence) into different ciphertext chunk sequences, while maintaining the effectiveness of deduplication.

The insight of TED is to randomly select a chunk's key from a set of candidates, instead of returning the same key as in Equation 11. Specifically, for each plaintext chunk M, let $x = \lfloor f/t \rfloor$, where f is the current frequency of M, and t is the balance parameter. After receiving the hash value of M, the key manager calculates a key seed candidate k_x as:

$$k_x = \mathbf{H}(k \,||\, h_1(M) \,||\, h_2(M) \,||\, ... \,||\, h_r(M) \,||\, x) \tag{12}$$

Then a key seed is uniformly selected from the candidate set $\{k_0, k_1, ..., k_x\}$:

$$k \xleftarrow{\$} \{k_0, k_1, ..., k_{x-1}, k_x\} \tag{13}$$

The client can then derive the key of M as

$$K = \mathbf{H}(k \,||\, P) \tag{14}$$

where P is the fingerprint of M. TED did not use k as the key of M to prevent the key manager and attackers who can eavesdrop on the key manager's reply from directly accessing the key.

As we observe more duplicates of M (that is, an increasing f), the most recent duplicates of M are encrypted based on some old key seeds in $\{k_0, k_1, ..., k_x\}$ used before. Therefore, TED maintains the effectiveness of deduplication by allowing the same key seed to protect some duplicates. At the same time, the generation of ciphertext chunks is uncertain because they come from a randomly selected key seed (as opposed to the deterministic key generation in Equation 11).

Generally speaking, a plaintext chunk with a higher frequency will be encrypted into a more diverse set of ciphertext chunks, because more candidate key seeds can be selected as f increases.

4.5 SGXDedup

Encrypted deduplication retains the effectiveness of deduplication on encrypted data and is attractive to outsourced storage. However, the existing encrypted deduplication methods are based on expensive encryption primitives, which can cause a significant drop in performance.

Server-assisted key management (like DupLESS [17] and TED [3]) requires expensive encryption operations [24] to prevent the key server from knowing the plaintext chunk and key during key generation.

Intel SGX provides a type of TEE, called a enclave, that allows data processing and storage with confidentiality and integrity [29]. The expensive cryptographic operations of encrypted deduplication can be offloaded by directly running sensitive operations in enclaves, thereby improving the performance of encrypted deduplication while maintaining its security, bandwidth efficiency, and storage efficiency.

SGXDedup [30] is a high-performance SGX-based encrypted data deduplication system. SGXDedup is built on server-aided key management like DupLESS [17], also using MLE key generation, but performs efficient cryptographic operations inside enclaves. It uses source-based deduplication and Proof-of-ownership (PoW) [31]. PoW is an encryption method that enhances source-based deduplication to prevent side channel attacks while maintaining the bandwidth savings of source-based deduplication.

The idea is to let the SS verify that the client is indeed the owner of the ciphertext chunk and is authorized to have full access to the ciphertext chunk. This ensures that a malicious client cannot query the existence of other clients' chunks. Specifically, in PoW-based source-based deduplication, the client attaches a PoW certificate to each fingerprint sent to the SS, and the SS can use it to verify whether the client is the true owner of the corresponding ciphertext chunk. The SS-only response after successful proof verification prevents any malicious client from recognizing ciphertext chunks owned by other clients.

Figure 6 presents the architecture of SGXDedup, which has multiple clients, a KS, a SS and two enclaves: the *key enclave* and the *PoW enclave*.

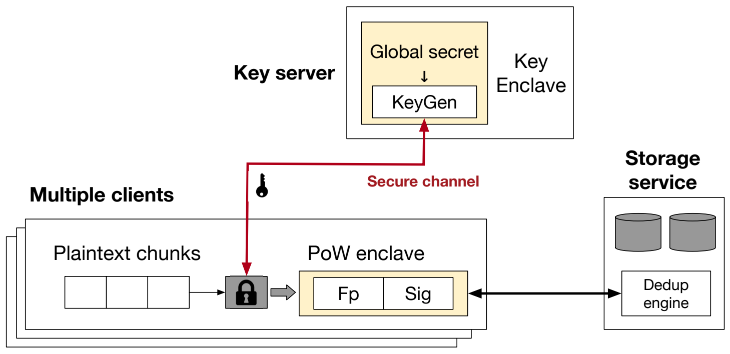

Fig. 6. SGXDedup Architecctture

SGXDedup deploys a key enclave in the KS to manage and protect the global secret of server-aid MLE [15], preventing a malicious or compromised KS from leaking the secret.

In order to perform MLE key generation, the key enclave and the client first establish a secure channel based on the shared blinded key, after that the client submits the fingerprint of the plaintext chunk through the secure channel. The key enclave calculates the MLE key as a cryptographic hash of the global secret and fingerprint:

$$K = \mathbf{H}(k \,||\, P) \tag{15}$$

It then returns the MLE key through a secure channel. The key enclave benefits both performance and security. It protects fingerprints and MLE keys through a secure channel based on a shared blinded key, so that the key server cannot learn any information from the MLE key generation process.

SGXDedup deploys a PoW enclave on each client to prove the authenticity of the ciphertext chunk in source-based deduplication. The PoW enclave first establishes a shared PoW key with the SS. SGXDedup uses Diffie-Hellman Key Exchange (DHKE) to implement the key agreement.

After the PoW key is generated, the client encrypts each plaintext chunk into a ciphertext chunk.

The PoW enclave takes the ciphertext chunk as input, calculates the corresponding fingerprint, and uses the PoW key shared with the SS to create a fingerprint signature. Then the client uploads the fingerprint and signature to the SS.

The SS verifies the authenticity of the fingerprint based on the corresponding signature and PoW key.

Only when the fingerprint is authenticated, the SS will continue to check whether the fingerprint corresponds to any duplicate ciphertext chunks that have been stored. The client verifies the ownership of the ciphertext chunk, not the ownership of the plaintext, to protect the original information from the SS.

SGXDedup has advantages over previous solutions [17] [3] [19], it has better bandwidth efficiency because of the use of source-based deduplication, it has better key generation performance due to the fact that blind fingerprints (OPRF protocol) is replaced with ECALLs to enclaves, and better secret storing security, since the secrets are store inside enclaves.

However, SGXDedup does not provide a mechanism to quantify the trade-off between storage efficiency and data confidentiality, meaning that the same plaintext will always map to the same ciphertext, making it vulnerable to frequency analysis attacks. Moreover, since it uses source-based deduplication, the client only uploads the non-duplicated ciphertext chunks to the cloud, meaning that it is able to detect if deduplication occured(Deduplication Detection).

4.6 Discussion

In this section, we discuss and compare the previously present work, summarize the tradeoffs that each solution offers and how their mechanisms can assist in achieving our goals. Table 2 presents the main aspects and mechanisms of each solution.

Table 2. Comparison of the main characteristics of different systems and approaches to deduplication.

System / Approach	**Deduplication Approach**	**Crypto approach to avoid key server analysis**	**Tunable**	**Key Generation Approach**
MLE	Target and Source based	-	No	CE/HCE1/HCE2/RCE
DupLESS	Target-based	Blinded Fingerprint	No	Convergent Encryption(MLE)
MinHash	Target-based	Minimum fingerprint of chunks in segment	No	MinHash Encryption
TED	Target-based	Blinded Fingerprint	Yes	Probabilistic Key Generation
SGXDedup	Source-based	Enclaves	No	Convergent Encryption(MLE)

Deduplication systems and approaches that offer target-based deduplication such as DupLESS, TED and MinHash incur in extra communication load for uploading all the data to the server. On the contrary, source-based deduplication systems such as SGXDedup are intrinsically vulnerable to deduplication detection from the client-side, despite the use of the enclave.

There is a visible trend in the related work by requiring the support of a key server, as a trusted third party to support cross user deduplication by centralizing the key generation. This solution followed by the majority of the systems [3,17, 30] requires the use of heavy cryptographic operation to avoid revealing sensitive information, such as the chunk content, to the key server. DupLESS and TED use the OPRF protocol [24], which can impose a significant performance overhead. Additionally, the constant communication with the KS for the key generation of each chunk can add significant latency and turn the KS into a bottleneck. In Li *et al.* [19] they send the minimum fingerprint of the segment, as this depends on all the chunks fingerprints inside the segment, and SGXDedup guarantees that the KS can't record any information since all the interaction are done between the client and enclave.

As for quantifying the trade-off between storage efficiency and data confidentiality, only TED provides a tunable mechanism, while other solutions [15,17,19] do not offer such configurability.

The final relevant characteristic of these systems is their key generation algorithm. Both DupLESS and SGXDedup rely on CE, the most secure variant of MLE described in Section 4.1, where each chunk will always map to the same key. MinHash also implements a deterministic key generation mechanism, however, the key depends not only on the chunk but on the whole segment, since the key for encryption is generated based on the minimum fingerprint of the segment chunks. Unfortunately, the use of deterministic mechanisms for the key generation leaves these systems vulnerable to frequency analysis attacks. Finally, TED uses probabilistic key generation, meaning that a chunk may map to different keys with the progression of time, hiding possible frequency patterns in the stored data.

The related work aims to design storage services that offer deduplication while protecting the data content and achieving high performance, in Table 3 we compare these systems against the possible attacks and performance metrics, described in Section 3.3. Also, we provide an entry for what our goal system should provide, when compared to these systems.

Table 3. Comparison of the protection against attacks and performance metrics in different systems and approaches, where N is the number of chunks.

System / Approach	Protection against brute force attacks	Protection against deduplication detection	Protection against frequency analysis	Key generation performance	Number of round trips
MLE	No	No	No	High	2 * N
DupLESS	Yes	Yes	No	Low	2 * N
MinHash	Yes	Yes	No	Low	2 * N
TED	Yes	Yes	Yes, to some degree	Low	2 * N
SGXDedup	Yes	No	No	High	> 2 * N
Goal System	Yes	Yes	Yes, to some degree	High	N

All previous solutions [3, 17, 19, 30] have protection against brute-force attacks, except MLE, this is due to the fact that all these systems and approaches employ a dedicated KS, responsible for generating and issuing the keys for encrypting chunks, because of this, all clients have to interact with the KS before encrypting a chunk and sending it to the SS. Also, the KS employs rate-limiting tactics, which limit the number of requests in a given unit of time, making attacks such as this one not practical. Our goal system should also protect itself from brute force attacks, by using the same techniques.

Against deduplication detection attacks, DupLESS, TED and MinHash use target-based deduplication to protect from this attack, this means that the SS is responsible for performing deduplication on ciphertext chunks.

However, SGXDedup does not have full protection against this attack, due to the fact that it uses source-based deduplication. A malicious client can always observe side channels to detect if enclave (on the client-side) uploads or performs deduplication on the desired chunk. The client can use side channels such as measuring the network bandwidth or the response time from the enclave, and others.

Again, our goal system should not allow anyone to detect if deduplication occured, so the use of target-based deduplication is a must.

Probabilistic key generation ensures that only a fixed number of plaintext chunks will be encrypted using the same key, and posterior identical plaintext chunks will use different keys. This approach degrades storage but improves security against frequency analysis, which is an acceptable trade-off. For this reason, our goal system shall use the same key generation as TED, protecting itself against frequency analysis.

Performance during key generation is also an important aspect, as stated before, DupLESS and TED use the OPRF protocol to avoid input analysis from the KS, which is a heavy operation. Since this operation is costly, solutions such as SGXDedup, that use an enclave to perform sensitive criptographic operations(e.g., key generation) will have a better performance, since it does not require heavy operations to hide client input. Our goal system will also need to achieve such key generation performance as SGXDedup.

Finally, all the previous systems require 2 * N round trips to store information in the SS, where N is the number of chunks. This is because the keys for encryption must be obtained by interacting with the KS, and later, all encrypted chunks must be sent to the SS. Even though solutions such as SGXDedup query the SS in order to check what chunks it must send, it incurs in extra round-trips to retrieve that information. Our system should minimize the number of round trips.

5 Architecture

In this section we present our system architecture. Our design is inspired by the previously presented systems and aims to achieve the goals that we set in Section 2. Our scheme removes the dependency on a remote KS, that imposes a constant latency overhead, by deploying a enclave directly at the SS, with the functionality of a KS, in order to protect sensitive information from the untrusted SS. We chose to follow the same scheme as TED, to achieve protection from the deduplication attacks while leveraging TEEs, similarly to SGXDedup. This strategy allows our solution to achieve a more efficient and robust storage system supporting encrypted deduplication.

Similarly to the related work, we assume that clients/users use a cloud storage system to store their data, they may download or upload data on demand. They do so by establishing a secure connection with the enclave at the SS, which is then responsible for encrypting the data and placing it in encrypted storage. The enclave will locally choose the key to encrypt the chunks, following the protocol proposed in TED. We now describe in more detail the operation to upload and download a file in our scheme.

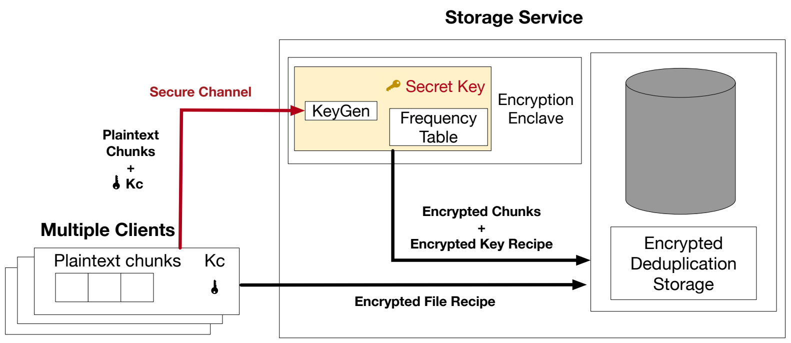

Fig. 7. Data Upload

The operation to upload a file is summarized in Figure 7. A client that wants to upload a file F, first breaks the file into multiple chunks. It also generates a file recipe, which lists the chunk hash and the chunk size based on the chunk order in the file. Then, the client encrypts the file recipe with a client file key Kc (unique for each client) and sends it to the SS. The client then sends all chunks and the client file key to the encryption enclave. The encryption enclave will then generate a key for each chunk, just as in TED, and encrypt each chunk with the respective key. After all ciphertexts have been generated, the encryption enclave creates a key recipe, which has the keys for all chunks, it encrypts the key recipe with the client file key Kc, and stores the ciphertext chunks and the encrypted file recipe on the SS.

To download a file (see Figure 8), a client retrieves the encrypted file recipe and the encrypted key recipe from the SS, as well as the ciphertext chunks. It then decrypts the key and file recipe using the client file key Kc. Finally the client decrypts each ciphertext text chunk using the respective key, and reconstructs the file based on the file recipe.

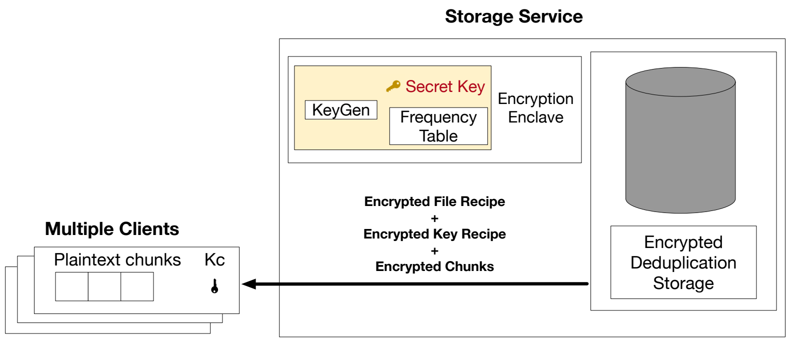

Fig. 8. Data Download

Our system improves on previous solutions [17] [3] [30] by only needing one single interaction with the encryption enclave, while in previous work they would have as many interactions with the KS as the number of chunks, in order to obtain the key for each chunk.

It also removes the need to perform OPRF [24] blind fingerprints to hide input information from the key server as previous solutions [17] [3] did, since we use an enclave to perform key generation and encryption, we trust the encryption enclave and the code running on it, so we know that its not recording information about user input.

Our solution is not trivial and brings a major challenge: any in-enclave application is limited to 96 MB. If an enclave has a larger size than the EPC, it encrypts unused pages and evicts them to the unprotected memory, suffering a performance penalty. For this reason, our frequency table must be designed in a way that accessing it does not incur in extra overhead. We will assume in a first step that the frequency table size is able to fit inside the enclave. In a second step, we will come up with a more scalable solution.

From what we presented before, we improve the performance, but what about security? Attacks such as the ones mentioned in Section 3.3 have as goal the retrieval of information about the contents stored by clients of the system. Our system must be able to protect itself against attacks of this sort.

To protect against brute force attacks, our system shall employ rate-limiting tactics on the encryption enclave, limiting the number of requests per unit of time, not allowing a malicious SS to perform multiple uploads in a given unit of time, slowing down the attack time.

As for deduplication detection attacks, we will use target-based deduplication, the client will always send the plaintext chunks to the encryption enclave. Also, to avoid any sort of analysis (time or network) done by an attacker, the encryption enclave will always perform encryption on all plaintext chunks. After encrypting all plaintext chunks, it will send them to the SS storage.

Finally, to protect against frequency analysis, our system will employ the same type of key generation as in TED [3], providing probabilistic key generation. To that end, our encryption enclave will have a secret key used in key generation, and also a frequency table, to keep the current count of each chunk.

6 Evaluation

To evaluate our solution, we will use similar metrics as previous works, such as TED [3] and SGXDedup [30]. We will evaluate our system in two different aspects: 1) performance on the process of uploading data; and 2) security against malicious users and SSs.

Additionally, we will also use a benchmarking tool called DEDISbench [4,5]. DEDISbench is a disk I/O block-based benchmark for deduplication systems. In DEDISbench, the data to be written is generated in a realistic fashion that mimics the content found on real storage distributions. This is a relevant property to have since we want to test the system in way that mimics as much as possible real environments.

6.1 Performance

Previous solutions [17] [3] rely on a dedicated KS to generate the keys for encrypting chunks, but to avoid input analysis from the KS these solutions perform an operation to hide the input. This operation is costly, it degrades performance. In our system there is no need to hide the input from the enclave, as the enclave is trusted and the code running on it has been attested. To evaluate the performance penalty, we will measure how much time our system takes from the moment a client starts uploading a file until that file has been safely stored in the SS.

6.2 Security

To evaluate the security of the system, we will simulate malicious users and storage services that try to perform each one of the attacks mentioned in Section 3.3. For brute force attacks, we will take into consideration the time it takes to evict an user who has been trying to repeatedly upload files in a small time window. For deduplication detection, we will create many clients who will try to upload the same file repeatedly and analyze if there is any indicator which can tell if deduplication occurred. Finally, for defense against frequency analysis attacks, we will compare the effect of probabilistic key generation vs deterministic key generation.

7 Scheduling of Future Work

Future work is scheduled as follows:

- January 14 - May 9: detailed design and implementation of the proposed architecture, including preliminary tests.
- May 10 - June 10: perform the complete experimental evaluation of the results.
- June 11 - July 23: write a paper describing the project.
- July 24 - September 15: finish the writing of the dissertation.
- September 15: deliver the MSc dissertation.

8 Conclusions

With the increasing growth of data in the wild, the need for online storage services increased too. To allow the storing of duplicate data without compromising storage efficiency, new techniques to keep storage efficiency were introduced.

Data deduplication is a technique that reduces the amount of redundant data that is kept in a storage system. If data is already stored, it creates a pointer to that copy of the data, instead of storing additional copies of the data.

To address user privacy, encrypted deduplication was introduced. The idea is to combine deduplication with confidentiality, allowing users to store their data encrypted, while allowing storage services to maintain the deduplication capability.

There are multiple attacks that can occur in these types of systems, these attacks have as an objective to get insight about what's stored in the system, and information about the user data. To solve these issues, many techniques were used, such as dedicated key servers, rate limiting techniques, probabilistic key generation, and enclaves.

In this work, we propose a secure way of performing encrypted deduplication without the need of dedicated KS, while still providing a tunable mechanism that balances the trade-off between storage efficiency and data confidentiality. Our solution is based on Intel SGX enclaves and aims to address the limitations of these TEEs.

Acknowledgments The author is grateful to Cláudio Correia for the fruitful discussions and comments during the preparation of this report.

References

1. Dropbox. Available at https://dropbox.com/
2. Google Drive. Available at https://www.google.com/drive/
3. Li, J., Yang, Z., Ren, Y., Lee, P.P.C., Zhang, X.: Balancing storage efficiency and data confidentiality with tunable encrypted deduplication, New York, NY, Association for Computing Machinery (April 2020)
4. Paulo, J., Reis, P., Pereira, J., Sousa, A.: Dedisbench: A benchmark for deduplicated storage systems. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". (September 2012)
5. Paulo, J., Reis, P., Pereira, J., Sousa, A.: Dedisbench. Available at https://github.com/jtpaulo/dedisbench Accessed: 2022-01-04.
6. Paulo, J., Pereira, J.: A survey and classification of storage deduplication systems. ACM Computing Surveys (CSUR) **47** (2014) 1–30
7. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. ACM Transactions on Storage (ToS) **7** (2012) 1–20
8. Wallace, G., Douglis, F., Qian, H., Shilane, P., Smaldone, S., Chamness, M., Hsu, W.: Characteristics of backup workloads in production systems. In: FAST, San Jose, CA, USENIX Association (February 2012)
9. Bolosky, W.J., Corbin, S., Goebel, D., Douceur, J.R.: Single instance storage in windows 2000. In: Proceedings of the 4th USENIX Windows Systems Symposium, Seattle, WA, USENIX Association (January 2000)
10. Eshghi, K., Tang, H.K.: A framework for analyzing and improving content-based chunking algorithms. Hewlett-Packard Labs Technical Report TR **30** 1–10
11. Kruus, E., Ungureanu, C., Dubnicki, C.: Bimodal content defined chunking for backup streams. In: 8th USENIX Conference on File and Storage Technologies (FAST 10), San Jose, CA, USENIX Association (February 2010)
12. Bobbarjung, D.R., Jagannathan, S., Dubnicki, C.: Improving duplicate elimination in storage systems. ACM Transactions on Storage (TOS) **2** (2006) 424–448
13. Lu, G., Jin, Y., Du, D.H.: Frequency based chunking for data de-duplication. In: 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Miami Beach, FL, USA, IEEE (August 2010)
14. Jurado, M., Smith, G.: Quantifying information leakage of deterministic encryption. In: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, New York, NY, USA, Association for Computing Machinery (November 2019) 129–139
15. Bellare, M., Keelveedhi, S., Ristenpart, T.: Message-locked encryption and secure deduplication. In: Annual international conference on the theory and applications of cryptographic techniques, Berlin, Heidelberg, Springer (May 2013)
16. Harnik, D., Pinkas, B., Shulman-Peleg, A.: Side channels in cloud services: Deduplication in cloud storage. IEEE Security & Privacy **8** (2010) 40–47
17. Keelveedhi, S., Bellare, M., Ristenpart, T.: Dupless: Server-aided encryption for deduplicated storage. In: 22nd USENIX Security Symposium USENIX Security 13), Washington, D.C., USENIX Association (August 2013)

18. GeeksforGeeks: Caesar cipher in cryptography. Available at https://www.geeksforgeeks.org/caesar-cipher-in-cryptography Accessed: 2022-01-04.
19. Li, J., Lee, P.P., Tan, C., Qin, C., Zhang, X.: Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. ACM Transactions on Storage (TOS) **16** (2020) 1–30
20. Ning, Z., Liao, J., Zhang, F., Shi, W.: Preliminary study of trusted execution environments on heterogeneous edge platforms. In: Proceedings of the 1st ACM/IEEE Workshop on Security and Privacy in Edge Computing, Bellevue, WA, USA (October 2018)
21. Ekberg, J.E., Kostiainen, K., Asokan, N.: The untapped potential of trusted execution environments on mobile devices. IEEE Security & Privacy **12** (2014) 29–37
22. SGX101: Overview - sgx 101. Available at https://sgx101.gitbook.io/sgx101/sgx-bootstrap/overview Accessed: 2022-01-04.
23. Correia, C., Correia, M., Rodrigues, L.: Omega: a secure event ordering service for the edge. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Valencia, Spain, IEEE (June 2020)
24. Naor, M., Reingold, O.: Number-theoretic constructions of efficient pseudo-random functions. Journal of the ACM (JACM) **51** (2004) 231–262
25. Bellare, M., Namprempre, C., Pointcheval, D., Semanko, M.: The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. Journal of Cryptology **16** (2003) 185–215
26. Camenisch, J., Neven, G., et al.: Simulatable adaptive oblivious transfer. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Berlin, Heidelberg, Springer Berlin Heidelberg (May 2007) 573–590
27. Chaum, D.: Blind signatures for untraceable payments. In: CRYPTO'82, Santa Barbara, CA, USA, Springer US (January 1983) 199–203
28. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms **55** (2005) 58–75
29. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS) **33** (2015) 1–26
30. Ren, Y., Li, J., Yang, Z., Lee, P.P., Zhang, X.: Accelerating encrypted deduplication via sgx. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21), Remotely, USENIX Association (July 2021)
31. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: Proceedings of the 18th ACM conference on Computer and communications security, New York, NY, USA, Association for Computing Machinery (October 2011)