NTNU

Norwegian University of
Science and Technology

Simulating secure cloud storage schemes

Jorge Campos de la Mata

Master of Telematics - Communication Networks and Networked Services
Submission date: June 2017
Supervisor: Colin Alexander Boyd, IIK
Co-supervisor: Gareth Davies, IDI

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: Simulating secure cloud storage schemes

Student: Jorge Campos de la Mata

Problem description:

With the increasing amount of data generated by enterprises and general users, cloud storage services have gained popularity because they provide a affordable solution to store this data. Existing cloud services providers such as Google Drive or Amazon S3 provide storage for millions of different clients, and they might take advantage of this fact enabling cross-user data deduplication. Data deduplication techniques allow to reduce storage costs and gain efficiency.

At first sight, cross-user data deduplication and encryption are incompatible. The former consists of deleting duplicated copies of data uploaded for different users and the latter (at least in the classical point of view) consist of encrypting data with personal user keys. Therefore, the same file encrypted with different user keys will generate different files, disabling the advantages of the data deduplication.

To address the problem, various schemes have been proposed. This project will focus on one of these schemes, Message-Locked Encryption (MLE), whereby the key is generated from the file itself. The project aims to analyze and evaluate the security of using MLE in combination with different deduplication strategies. A testing environment will be developed to assess the possible information leakage incurred by the combination of various file-chunking strategies and MLE.

Responsible professor: Colin Alexander Boyd

Supervisor: Gareth Thomas Davies

Abstract

Cloud storage services have become a popular solution to store large amounts of data generated by users and enterprises, because they provide an affordable and practical solution. In order to gain efficiency and reduce storage costs, cloud storage servers may remove duplicated copies of the same stored data. This process is called cross-user data deduplication. However, this beneficial procedure is not carried out if the users encrypt their files with their personal keys. To make deduplication and encryption compatible, we can deterministically encrypt a file using a key generated from the file itself. This process is called Message-Locked Encryption (MLE).

This thesis aims to analyze and evaluate the security of using MLE in combination with different deduplication strategies. The information leakage incurred by MLE and conventional encryption is studied. A testing environment is also developed to test these schemes in order to fullfil the objectives.

After the experiments, we have confirmed that a curious cloud storage server may obtain information about the stored files even when they are encrypted. This leakage is more significant for MLE scheme, but it also exists when the users encrypt the files with their personal keys. This confirms and advances the work of Ritzdorf et al. [18].

Preface

This thesis is the final work of my master's degree. It has been carried out at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway.

Firstly, I would like to thank my home university, Universidad Politécnica de Madrid, to give me the opportunity to study abroad my last year of the master. This is an experience that I may recommend to everyone and it has been very rewarding.

Secondly, I would like to thank and acknowledge my supervisors Colin A. Boyd and Gareth T. Davies, for providing guidance and support during all of the work of my thesis.

Last but not least, I would like to thank the people that I met in Trondheim during my exchange for making this year in Trondheim one of the best years of my life.

Trondheim, 12th of June 2017

Jorge Campos de la Mata

Contents

List of Figures

List of Tables

Chapter 1
Introduction

With the increasing amount of data generated by enterprises and general users, cloud storage services have gained popularity because they provide a affordable solution to store this data. Existing cloud services providers such as Google Drive or Amazon S3 provide storage for millions of different users, who also may have one or more clients (devices), and they may take advantage of this fact by storing duplicated information only once. That is, if two different users upload the same data to the cloud server, this data will be stored only the first time. This allows saving storage costs and gain efficiency. This process is called *data deduplication* in the literature [4]. Studies have shown that data deduplication can save large volume costs in applications where the possibility of finding duplicated content is relatively high.

Cloud storage services have become a popular solution to store data generated by users and enterprises, because they provide an affordable and practical solution. In spite of this, several enterprises and general users are still reluctant to store very sensitive data on them. It is worth stressing that when clients upload any information to the cloud storage provider, they are losing the control over their data. Therefore, they are putting their trust in the cloud server integrity and in the security mechanisms that it uses. Thus, cloud storage clients could be interested in uploading their data encrypted by themselves to the cloud, in order to achieve a *secure cloud storage system* that protects their data.

At first sight, both concepts mentioned above, *deduplication* and *encryption*, are incompatible. That is, the same file encrypted and uploaded by two different users (each one with their personal key) will produce two different encrypted files in the cloud. Data deduplication will not be possible because these files are completely different after the encryption. Therefore, an encryption scheme (at least in its conventional type, where each user has their key) disables the resource savings that deduplication may provide.

To address this problem, the literature features several proposals for the concept

called *secure data deduplication* in the cloud [1] [2] [6] [16] [17] [19] [11]. All these solutions share the objective of enabling deduplication and encryption at the same time, in order to assure data confidentiality and resource savings. The main idea of these proposals consists of obtaining the key from the file itself instead of from the user identity. In this way, the same file generated from different users will have the same encrypted content and the deduplication will be possible. These solutions are called *Message-Locked Encryption* (MLE) in the literature [3].

In this thesis, we analyze the information leaked in both schemes, conventional and MLE encryption, when they are used in a cloud storage service context. This study may allow us to reach several conclusions related with the trade-off between resource savings and information leaked in each encryption scheme, pointing out which of them is more suitable for cloud storage service applications.

1.1 Objectives

The objectives of the document are summarized as follows:

- In order to carry out the experiments, a complete **testing environment** is developed. This testing environment simulates a cloud storage provider, it includes mechanisms to upload and download files from a client software and saving them in a database hosted on a server. In this implemented environment it will be possible to choose the encryption scheme and the deduplication algorithm, in order to accomplish the following objectives.
- Data deduplication process is analyzed in detail. There are several **deduplication algorithms** which split data in several small blocks in order to obtain a better performance. These algorithms are implemented, tested and compared between them through a set of experiments. To do that, a dataset with approximately 78 GB of data is created and used to fulfill this goal.
- Encryption process in cloud storage systems is also analyzed. A Message-Locked Encryption (MLE) scheme is implemented as well as a conventional encryption one (where each user has their personal key). The **information leakage** that it is possible to infer in each scheme is studied and tested through several experiments. The trade-off between resource savings and security is also discussed.

1.2 Structure of the report

The remainder of the document is structured as follows. Chapter 2 and 3 represents the theoretical background related with the topic, this knowledge will have great

importance in followings chapters. On the one hand, in Chapter 2, we describe the data deduplication process and its variants in detail through previous literature. On the other hand, Chapter 3 is related to the security issue in the cloud, through previous literature, we explain the information that a cloud storage system may leak in the proposed encryption schemes.

In Chapter 4, the developed testing environment is described in detail. Chapters 5 and 6 refer to the performed experiments, showing and discussing their results. The former refers to experiments related to deduplication algorithms, and the latter refers to tests performed in order to study the information leakage in cloud storage schemes with encryption and deduplication enabled. Lastly, we conclude the document in Chapter 7, extracting several conclusions.

Chapter 2
Data deduplication

Data deduplication is a technique that consists of deleting redundant copies of identical data saved in a datastore. This procedure is used to save storage resources in large databases, reducing the amount of stored data on them. The first data deduplication solutions appeared almost at the same time as large-scale storage systems, due to the necessity of an efficient management of redundant copies of data. Firstly, it was used in back-up and local applications [4], but its growth appeared with cloud storage providers.

Studies have shown that cross-user data deduplication can save volume costs by more than 68% in primary storage systems and 83% in back-up storage [12]. In addition, deduplication might improve the storage management stack, I/O efficiency [10] and network bandwidth consumption if the deduplication process is done in the client side [13].

Basically, data deduplication process consists of assigning one fingerprint (hash, checksum...) for each deduplication unit (file, block...). In this way, it is possible to uniquely identify each block using its fingerprint. This fingerprint will be used to compare different blocks in order to detect and delete duplicated copies of data. That is, if two blocks have the same fingerprint, it means that they come from the same content and therefore, one of them could be deleted on the database.

Data deduplication techniques may be classified attending different categorization criteria[15]. We will focus on the classifications based on locality and granularity.

2.1 Deduplication techniques categorized by locality

The location where the deduplication is carried out is one of the most common categorization criteria. Basically, we may find server-side deduplication and client-side deduplication.

2.1.1 Server-side deduplication

In server-side deduplication, the deduplication procedure is made on the server. In Figure 2.1 we can appreciate a server-side deduplication architecture. The client uploads the file directly to the server, where the deduplication is performed. Thus, the client is unaware of any deduplication process carried out in the server.

When the server receives the file, it splits the file in several blocks (*chunking*), and it assigns a tag to each block (*tagging*). In the figure, the file is chunked in four blocks; F_1, F_2, F_3 and F_4, and the tags T_1, T_2, T_3 and T_4 are assigned to each block, respectively. Before storing the generated blocks, the server checks if they are already stored in the database, comparing their tags. If the tag exists, the server will not store the associated block (case of F_1 and F_2 in the figure), if not, the server will store it on the database (case of F_3 and F_4).

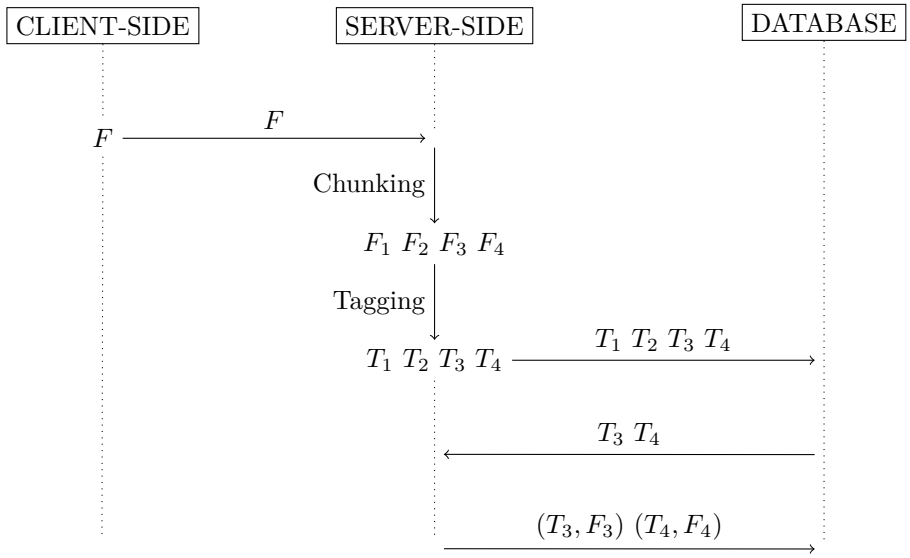


Figure 2.1: Server-side deduplication architecture.

Notice that in server-side deduplication solution, the system is saving resources in the storage process, but not in the communication client-server, since the user is always sending the whole file, regardless of whether the file was already stored (this fact is checked afterwards).

2.1.2 Client-side deduplication

In client-side deduplication, also called source-based deduplication, the deduplication process is made in the client part of the architecture. In this architecture, the system

is saving resources not only in the storage process, but also in the communication client-server.

The Figure 2.2 represents a client-side deduplication architecture. Firstly, the client chunks the data into several blocks and hashes them. In the figure the client generates the tags T_1, T_2, T_3 and T_4 from the chunks F_1, F_2, F_3 and F_4. Then, the server compares the generated tags with the stored tags on the database, checking for their existence, and it returns to the client only the tags which are not stored. In this way, if one tag is already stored on the database, the client will not send the associated data segment over the network, preventing in this way duplicated copies of the same data on the database and the bandwidth is saved in the communication. In the figure, the chunks F_1 and F_2 were stored in the database previously, so the storage server responds to the client sending only the new tags in order to the client knows this fact as well. Finally, the client sends the new chunks to the server ($F3$ and $F4$ in the case of the figure) and they are stored on the database.

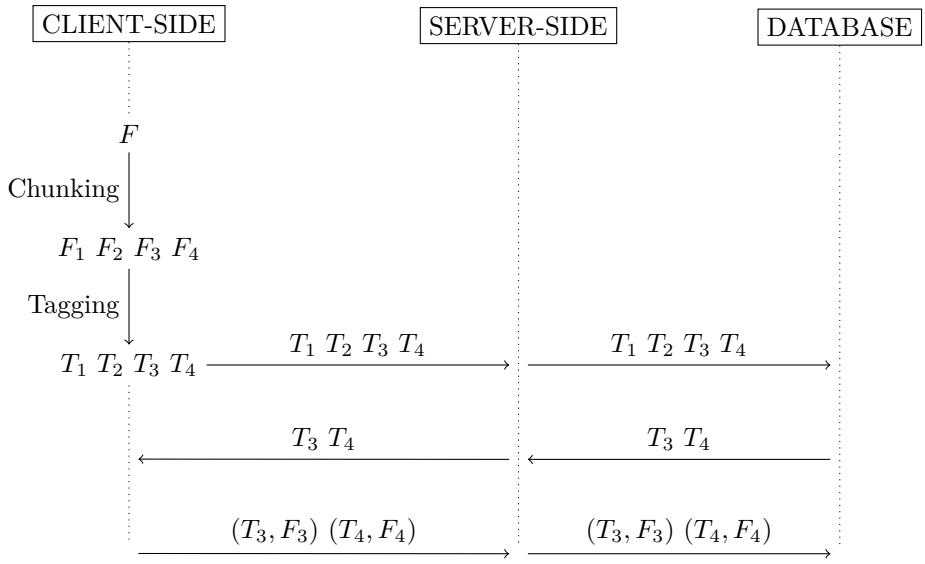


Figure 2.2: Client-side deduplication architecture.

We can appreciate that each data segment sent to the server is compared with the actual stored data (data originated from other users), that fact is called cross-user data deduplication. As we have mentioned previously, the main advantage of client-side data deduplication is the bandwidth saving, since it is very common that different users upload the same content (films, programs...). Therefore, these large files are only transferred over the network the first time. In spite of these advantages, is well-known that client side deduplication has side-channel security issues [9]. In

Section 3.2.1, these attacks will be explained in more detail.

2.2 Deduplication techniques categorized by granularity

Another common classification for data deduplication systems is regarding the block size or granularity. Granularity refers to the technique used to split data into blocks. These blocks will be the basic unit for deleting duplicates.

2.2.1 File-based deduplication

One of the simplest approaches to split the data into chunks is file-based deduplication, where the chunk is the whole file. One tag is assigned to each file, usually using a hash function, and then the tags are compared to decide if two files are identical. On the one hand, this method has fewer blocks to index and avoids the chunking process. On the other hand, the storage space saved is less than in other strategies, although it always depends on the target application. For instance, file-based deduplication could be a good solution in applications where identical files are uploaded to the server (licenses, software versions or files which always present the same content, without modifications).

2.2.2 Fixed block-size deduplication

Another common granularity strategy is fixed block-size deduplication. It means partitioning the file in small chunks with the same size and hashing their content to obtain their fingerprints. In this case, the amount of data to compare in the deduplication process is less than in file-based deduplication. Therefore, the matches will be more probable and the deduplication performance will increase, but the index table will be larger than in file-based deduplication.

Fixed-block size strategies could fail deduplicating in several situations. For example, we suppose that two large files A and B have exactly the same content except for one single byte at the beginning of the file B. In this particular case, all the resultant blocks after the fixed-size chunk process will be different, although the content is exactly the same except for the first byte. This well-known problem related with fixed block-size deduplication is called "boundary-shifting" problem [7]. In Figure 2.3 is shown a graphical example of this issue, where F_1 and F_2 have been chunked using a fixed block-size of 5. F_1 and F_2 contents are shifted by one byte, but their blocks are different and therefore, all the blocks will be stored. This problem is addressed using content-based chunking (CDC) [13].

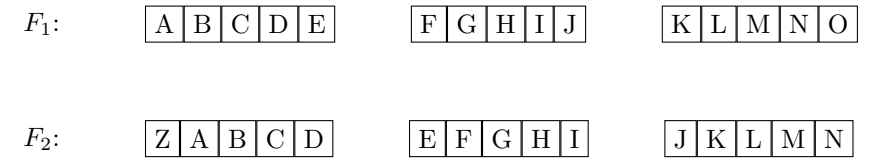


Figure 2.3: Boundary shifting problem example.

2.2.3 Content-defined chunking (CDC)

Content-defined chunking (CDC) refers to a set of algorithms which share the same purpose, that is to address the boundary-shifting issue associated with fixed block-size deduplication. The chunk boundaries in fixed block-size deduplication are set considering only the fixed size, but not the content of each block, and that is the source of the "boundary-shifting" problem. CDC main purpose is to generate content-based file chunks, thus obtaining variable size blocks. In this way, the block boundaries are set depending on the content, fixing the "boundary-shifting" problem mentioned above, as we may appreciate in Figure 2.4, where only the first block in F_2 is different from the blocks in F_1 due to their contents are shifted by one byte.

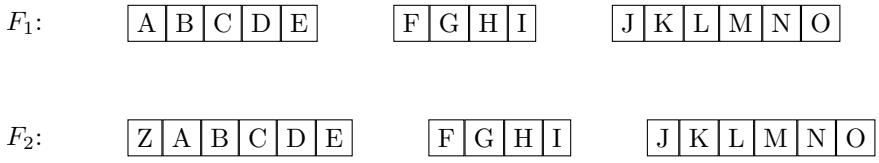


Figure 2.4: Boundary shifting problem fixed.

2.2.3.1 Basic Sliding Window (BSW)

Basic Sliding Window (BSW) CDC algorithm, proposed for Low-bandwidth Network Filesystem (LBFS) [13], is one of the first CDC algorithms. In BSW, the boundaries are set considering the block content. However, maximum and minimum block size thresholds are usually set in this algorithm. A minimum block size threshold is desirable because if a set of very small blocks is obtained after the chunking procedure, it will generate an index table too large on the server. This fact could cause that the storage savings associated with deduplication will not be reached due to the overhead. On the other hand, a maximum block size threshold is also desirable because if a set of very large blocks are obtained, the deduplication matches are more difficult on the server. Implementations of this algorithm have set these values around 4KB for the minimum and 12KB for the maximum [20].

The purpose of the algorithm is sliding, byte by byte, a fixed-size window (e.g. 64B) across the file content. The sliding-window starts from the position of the

minimum block size, and checks if the window content satisfies a certain condition in that position. To evaluate it, the window content is hashed. If the hash satisfies a pattern (e.g. the lowest N bits are all zeros) then a new chunk boundary is found. If not, the sliding window moves ahead to the next byte. In this way, the sliding-window is moving across the file content until the window content satisfies the condition. In the particular case that the sliding window reaches the maximum block size (the block boundary has not been found), the block boundary is forced on this point, thus obtaining a block with the maximum chunk size.

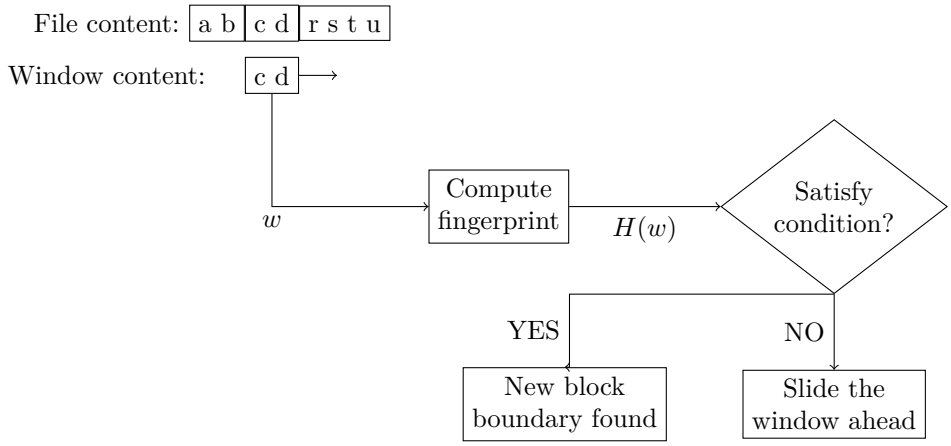


Figure 2.5: Basic Sliding Window algorithm (BSW).

To obtain the hash (in order to check the condition), any hash function could be used. But a rolling hash function as *Rabin fingerprint* (Definition 2.1) is more efficient for this purpose, since the computation of the Rabin fingerprint of a region B can reuse some computation of the region A, when A and B overlap.

Definition 2.1. Rabin fingerprint Rabin fingerprint scheme consists of n-bit message $m_0, ...m_{n-1}$ as input. This rolling hash computes this message as a polynomial of degree n-1 over the finite field GF(2): $f(x) = m_0 + m_1 x + ... + m_{n-1} x^{n-1}$. Then, a random irreducible polynomial $p(x)$ of degree k over GF(2) is selected, and we define the fingerprint of the message m to be the remainder $r(x)$ after division of $f(x)$ by $p(x)$ over GF(2).

It is worth stressing that sliding-window-based CDC algorithms require an extra cost computation than solutions as fixed block-size deduplication. Recall also that the sliding window moves byte by byte across all the range, and a rolling hash computation has to be done in each byte. Although the complexity of rolling hash computation is not high, large numbers of processes (large files) could affect the system performance. Furthermore, the deduplication performance in BSW is not

optimal since this algorithm only adopts one condition (one judgment function). This fact could result in a high number of forced boundaries with the maximum chunk size, and these forced chunks are not content-based.

2.2.3.2 Two Threshold Two Divisors CDC algorithm (TTTD)

The Two Threshold Two Divisors CDC algorithm (TTTD) [7] is designed to improve the deduplication performance provided by BSW. TTTD adds a secondary condition (divisor) in the algorithm, softer than main condition, whose purpose is to determine back-up breakpoints. Except for this, the operation of this algorithm is similar to BSW.

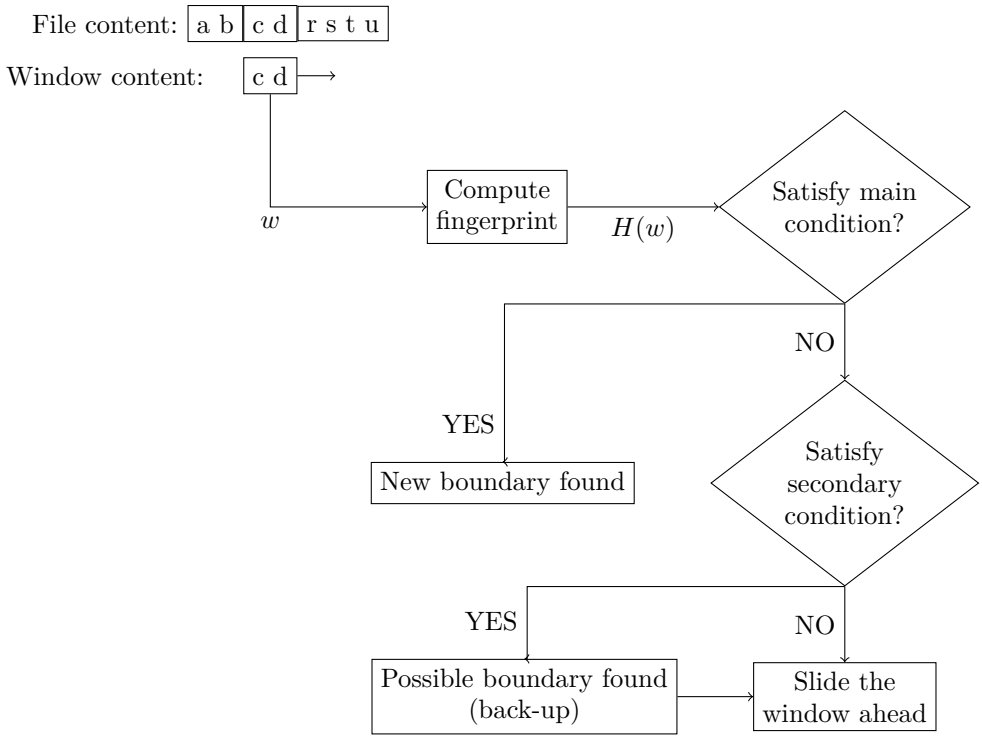


Figure 2.6: Basic Sliding Window algorithm (BSW).

In TTTD, the sliding window starts from the position of the minimum chunk size, computing a rolling hash function of the window content as in BSW. But this time, it is checked if the fingerprint satisfies the main condition, if not, it is checked if it satisfies the secondary one. If the fingerprint satisfies the secondary condition, a back-up boundary is set, and the sliding window moves ahead. When the sliding window reaches the maximum chunk size and any point has satisfied the

main condition, the new boundary will be the first point that satisfied the secondary one, if it exists.

TTTD can greatly reduce the number of forced breakpoints, and therefore, the deduplication performance. However, the computation cost is the same or even worse than BSW, because TTTD does nothing to reduce the execution time of rolling hash process, and this time the judgment function is done twice in each byte, one for each condition.

2.2.3.3 Leap-based CDC algorithm

In order to reduce the executing cost of the algorithm, leap-based CDC algorithm was suggested [20]. The computation complexity of the judgment function in leap-based CDC algorithm could be less than 2.5 times that of the BSW, depending on the parameters.

In contrast to BSW algorithm, in which every position of the file corresponds to one window, in leap-based algorithm every position corresponds to M windows, where M is a fixed number. For instance, in Figure 2.7 M = 5.

A new chunk boundary is found when all the windows associated with the same position satisfy the judgment function. Figure 2.7 illustrates the associated window contents for the position "h" in the file. If one of the M windows related with the point does not satisfy the condition (unqualified window), the new target position will be set starting from the last point of this unqualified window.

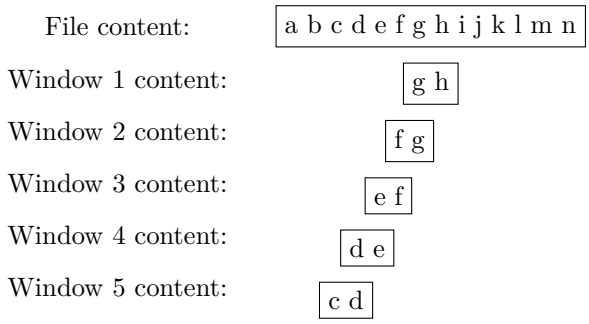


Figure 2.7: Multiple windows in Leap-based CDC algorithm.

For instance, we suppose that in Figure 2.7 window 1 has satisfied the condition, but window 2 does not (unqualified window). In this situation, it is not necessary computing the rest of windows to know that "h" position will not be a boundary. Then, the target point changes M-1 positions. So, the new target point will be "l", window 1 will have to content "k l", and the last window (5 in the Figure) will have

the content "g h". In this way, the algorithm is leaping over some target positions in the process of searching new chunk boundaries.

If the maximum chunk size is reached and new chunk boundaries have not been found, the chunk boundary will be forced in this point as in the BSW, originating a chunk with the maximum chunk size.

2.3 Summary

Data deduplication is a desired process in cloud server providers, since it allows saving large number of resources. The data deduplication process consists of splitting files in several blocks and associate a tag to each one. If a certain block is already stored in the cloud server, this block will not be stored once again although another user uploads it. This fact allows avoiding duplicates in the cloud server. In Figure 2.8 it is shown the ways that we have used to categorize this process. On the one hand, we have seen that they exist several chunking strategies in order to separate the files. These strategies are as follows:

- **File-based deduplication**. There is no chunking process. The block is the file itself.
- **Fixed block-size deduplication**. The file is separated in blocks of the same size.
- **Content-defined chunking (CDC)**. The file is split in blocks with variable sizes depending on the content of the file.

On the other hand, it is possible to classify the deduplication process depending on where it is located.

- **Client-side deduplication**. The chunking process is done by the user, and it allows saving resources not only in the storage server, but also in the communication.
- **Server-side deduplication**. All the deduplication process in done by the server, and the user is unaware of it. The resources are only saved in the storage server.

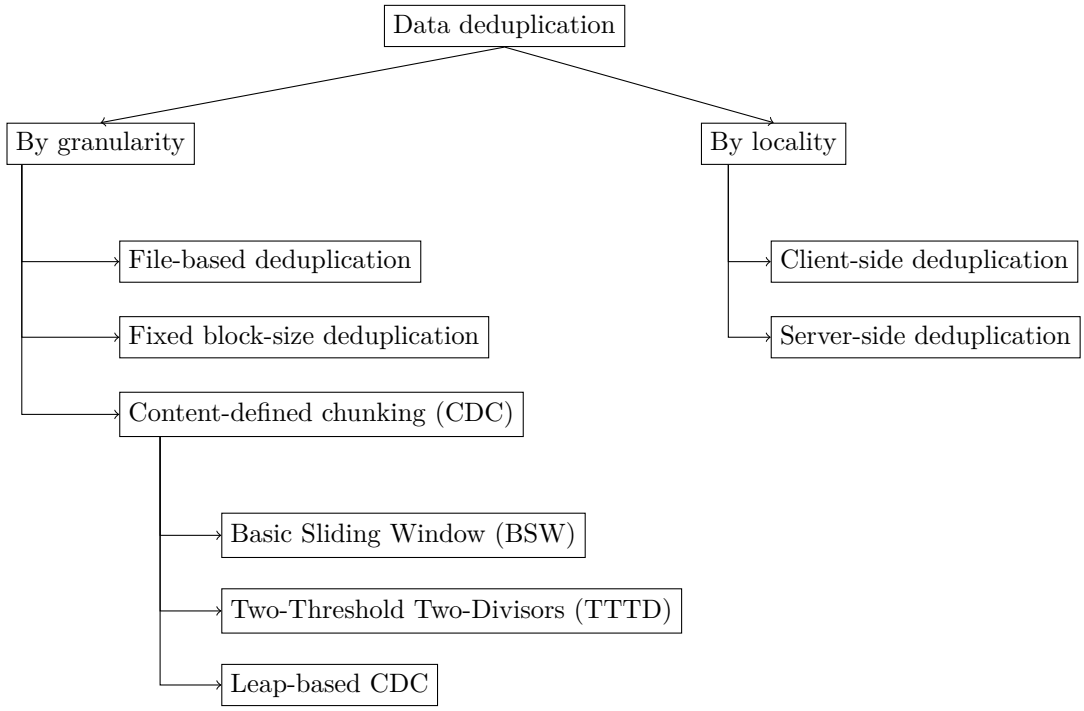


Figure 2.8: Deduplication summary

Chapter 3
Encryption and information leakage in cloud systems

When clients upload some information to the cloud storage provider, they are losing the control over their data. Therefore, they are putting their trust in the cloud server integrity and in the security mechanisms that it uses. Previous literature has shown that this decision involves a certain degree of risk, which will be summarized in this section.

An encryption mechanism is desirable to protect the confidentiality of the uploaded data. The use of conventional encryption (a scheme where each user has their own key) disables data deduplication and its advantages. In this section, Message-Locked Encryption (MLE) is explained, one alternative of conventional encryption that permits the use of data deduplication over a scenario with encryption.

In addition, the usage of data deduplication may imply an information leakage that it could be carried out both by third parties and the storage server itself. In this section, it will be also explained what information that a cloud storage system with deduplication could leak even if encryption is enabled.

3.1 Message-Locked Encryption (MLE)

Conventional encryption and data deduplication processes performed at the same time may present incompatibilities in cross-user deduplication scenarios, i.e., when a file uploaded from different users is deduplicated. We suppose that two different users, Alice and Bob, desire to upload the same file F to the cloud storage provider. If the encryption process is skipped, Alice and Bob will upload the same file F into the storage server and the deduplication process will be done correctly. Nevertheless, we assume that Alice and Bob desire to encrypt the file before uploading it to the server, to protect it against third parties, using for this purpose their user keys K_a and K_b. The resultant files after the encryption will be F_a and F_b respectively, being different between them, since they have been generated from different user keys. In

this case, the deduplication process will not carry out in the server because F_a and F_b are completely different.

As we may observe, conventional encryption comes at odds with deduplication. The literature features a number of proposals for *securing data deduplication* in the cloud. All these solutions share the objective of enabling cloud storage providers to deduplicate encrypted data, ensuring confidentiality at the same time that they can take advantage of the deduplication benefits.

To address the incompatibility between deduplication and confidentiality, Douceur et al. [6] proposed a solution called Convergent Encryption (CE), that solved the issue. The main idea consists of obtaining the key from the own file instead of from the user identity. In this way, the same file generated from different users will have the same encrypted content and the deduplication will be possible. In CE, the key is the hash value of the file content.

As an example, we suppose that Alice and Bob desire to upload the same file F, but they also want to assure confidentiality over the file content. Alice hashes the file F in order to obtain the key $K = H(F)$, where H is a hashing function. Bob does the same process, getting also K. Both Alice and Bob have the same key, and after the encryption $C = E(K, F)$ they will obtain the same ciphertext C. Thus, the same file encrypted by different users results in the same ciphertext, enabling deduplication. Notice that the scheme is deterministic [8], and this attribute is causing that encryption and deduplication can coexist.

This idea has been very significant in subsequent literature. As an example, ClearBox [1] or DupLESS [2] secure deduplication solutions include CE or small variations of it. The set of solutions that come from CE encryption are called Message Locked Encryption (MLE) in the literature, because the message is locked under itself.

Generally, any MLE scheme is based in a five-tuple of PT algorithms $MLE = (\mathcal{P}, \mathcal{K}, \mathcal{E}, \mathcal{D}, \mathcal{T})$ [3], the last two deterministic. However, to simplify the implementation, $\mathcal{P}$ (the public parameter generator) is omitted:

$K \leftarrow \mathcal{K}_p(M)$ The algorithm $\mathcal{K}_p$ is in charge of generate a key from a message M given as input. To accomplish this task, a hash function is required. Hence, $\mathcal{K}$ algorithm could be any hash common function as SHA-256 or SHA-3. Therefore, the resultant key is a fingerprint of the file (or block) content, and it may be used afterwards to encrypt, regardless of which user performs it. The user has to store this key K securely.

$C \leftarrow \mathcal{E}_p(M, K)$ The algorithm $\mathcal{E}_p$ is the encryption function. This algorithm needs the key K generated with the algorithm $\mathcal{K}_p$ and the message M. The resultant value is the ciphertext C. As encryption algorithm, it is common to use block cipher algorithms such AES, with fixed Initialization Value (IV) due to it has to be a deterministic function (same input has to result in the same output).

$M \leftarrow \mathcal{D}_p(C, K)$ The algorithm $\mathcal{D}_p$ is the decryption function. On inputs key K and ciphertext C, this algorithm is able to recover the original message M. As well as in the encryption function $\mathcal{E}_p$, a block cipher decryption algorithm such AES is common to use.

$T \leftarrow \mathcal{T}(C)$ The algorithm $\mathcal{T}$ is used to tag the ciphertext, in order to obtain a fingerprint that represents the ciphertext C. The algorithm only needs the ciphertext C as input, and it returns the desired tag T This function is useful to perform the deduplication process easily, comparing tags instead of large chains of ciphertexts. As well as in the $\mathcal{K}_p$ algorithm, a hash function as SHA-256 or SHA-3 may be used for this purpose.

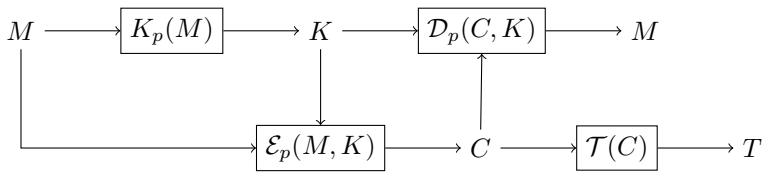


Figure 3.1: Message-Locked Encryption (MLE). Algorithms $\mathcal{K}, \mathcal{E}, \mathcal{D}, \mathcal{T}$ and their relations.

Therefore, an application of Message-Locked Encryption (MLE) mechanism could be as follows. A client of the storage provider wants to upload a file M. To simplify the example, it is supposed that file-based deduplication technique is used, and therefore, the chunking process is skipped. Firstly, the user obtains the key K from the file M using $\mathcal{K}_p(M)$, and it is stored locally by the user.

In the *upload protocol*, the user encrypts the file M using the previously stored key K, applying the function $\mathcal{E}_p(M, K)$. When the ciphertext C is generated, the user sends it to the storage server, where the deduplication process will be carried out. The server tags the ciphertext C received from the user, applying $\mathcal{T}(C)$ and it checks if it is already stored a ciphertext with the same tag. If not, the ciphertext C is stored together with its associated tag T.

In the *download protocol*, the user requests the file M to the server. The latter sends to the user the ciphertext C stored on it. Then, the user decrypts $\mathcal{D}_p(C, K)$

using the C received, the public parameter P and the corresponding key K stored in the upload protocol. Finally, the user recovers the original file M.

3.2 Information leakage in cloud storage systems

As it is explained before, data deduplication is a very usual process carried out in cloud storage providers with the goal of save resources. However, the usage of this technique may imply an information leakage that it could be carried out both third parties and the storage server itself.

Basically, it is possible to infer some information about the stored data in the servers where deduplication is enabled. This is possible taking advantage of two leaks: client-side information leakage and access traces information leakage.

3.2.1 Client-side information leakage

Harnik et al. [9] pointed the possibility of performing several attacks in a cross-user data deduplication scenario with client-side architecture.

On the one hand, cross-user data deduplication is the most common cloud storage scenario when the data is outsourced to a cloud storage server. That is, each deduplication unit (file or block) is compared with the data of other users, and the deduplication is performed if an identical copy is already stored in the cloud storage server. On the other hand, client-side deduplication (Section 2.1.2) is commonly used due to the bandwidth and storage savings that it provides.

It is worth mentioning that these attacks can only be performed in a scenario where encryption is disabled or it is deterministic (the same input produces always the same output).

3.2.1.1 File presence

This attack allows knowing if a certain file is stored (presence) or not (absence) in a storage server. We suppose that a law authority wants to know if an illegal file is stored in the cloud server, and this law authority has a copy of the illegal file.

To perform the attack, the law authority only has to try to upload the file and check if the deduplication occurs. Firstly, the law authority will send the tags of the file blocks. Then, the server will check if it has already stored the content associated with these tags. If all the blocks are already stored, the server will send nothing to the client later, and it will mean that the whole file is stored in the server. However, if at least one uploaded tag does not coincide with the stored tags, it will mean that the file is not stored.

As we may observe, this attack is very straightforward, and it takes advantage of the deterministic encryption leakage. Although the files are encrypted before the uploading process, in deterministic encryption, the same plaintext will always generate the same ciphertext. Therefore, the same file encrypted by one user and the law authority will generate the same ciphertext, and the deduplication will occur in the server.

Notice that in server-side deduplication this attack cannot be performed by a third party as in client-side deduplication. That is because the server will not return information to the user about the deduplication process. In spite of this, the storage server itself could accomplish this attack in server-side deduplication as well, because it knows when the deduplication is occurring.

3.2.1.2 Template attacks

The attack described above is only able to check if a file is stored or not in the cloud provider server. Nevertheless, template attacks are able to guess some specific parts of the target file. To perform this, a brute force attack is deployed.

As an example, we suppose that Alice and Bob are employees of the same company. The company uses a common contract template to set the salaries and further information about the employees. Alice is curious to know Bob's salary. She will be able to get it taking advantage of the information leakage of client-side deduplication. She has to fill the contract template with Bob's name and inserting a set of possibles values in the field *salary*. The process is the same as the file presence attack described above. Alice will check if the deduplication occurs (the file is not stored) or not (the file is new). When Alice finds a salary when the deduplication does not occur, it will mean that the inserted salary is the Bob's salary.

3.2.2 Access traces information leakage

Ritzdorf et al. [18] analyzed the information leakage associated with data deduplication on a curious storage server, even if the information is encrypted. This leakage comes from the access traces generated in the communication between client and server.

It is worth remembering that to perform client-side attacks, encryption has to be disabled or it has to be deterministic. However, in access traces attacks, the curious storage server could acquire information even if conventional encryption is used (each user encrypts the files with this own key).

In the scenario proposed by Ritzdorf et al. [18] it is supposed that the curious storage server cannot guess or acquire the encryption keys, and it may only observe

limited information related with the communication packets (as object ID, object size and timestamp).

3.2.2.1 Storage Graph and deduplication fingerprints

Basically, the curious storage server has to generate a storage graph G and update it when a user uploads new file blocks. Then, G will be useful to extract considerable information about the stored files.

To construct the storage graph G, each file f is modeled as a tree $T(f)$. Each leaf node represents a deduplication unit of the file f. That is, if the used chunking algorithm was content-defined chunking (CDC) or fixed block-size, each block is represented in the graph as a leaf node of $T(f)$. In file-based chunking algorithm, $T(f)$ will have only one leaf node.

Initially, G is empty and the file trees are added when the clients upload new files. G is populated as follows: If the whole file f is deduplicated, we leave G unchanged. Otherwise, we create a new tree $T(f)$, initially only the root node. If an object of f (we refer *object* as deduplication unit) cannot be deduplicated, a new leaf node is created and linked with the root node of $T(f)$. Otherwise, if the object can be deduplicated, the root node of $T(f)$ is connected to the node of G that represents the stored object.

We may appreciate that each root node is connected to exactly one leaf node in case of file-based deduplication. In fixed block-size deduplication, each root is connected to at least one leaf node and most of the nodes have the same size (except the last block in each file). In CDC, each root is connected to at least one leaf node as well, but the nodes may have different sizes.

Definition 3.1. Deduplication fingerprint "The deduplication fingerprint is a feature of a file that it has been chunked previously in the deduplication process. The deduplication fingerprint consists of the number of blocks and the sizes of each block."

In G, each file has its deduplication fingerprint (Definition 3.1), and it is denoted as $T(f)$. Given two files f_1 and f_2, we may say that they have the same deduplication fingerprint if $T(f_1)$ is a valid isomorphism of $T(f_2)$ [18]. An isomorphism between two deduplication fingerprints is valid if preserves number of nodes, edges and the sizes of the leaves. For instance, we suppose that file f was chunked in three blocks with sizes 20, 30 and 25 kB respectively. The deduplication fingerprint of f will be $[20, 30, 25]$. We stress that the deduplication fingerprint is not valid to uniquely identify files, because another file with the same number of blocks and block sizes will have the same deduplication fingerprint, as we may observe in Figure 3.2. In the

figure, f_1 and f_3 are the same file, thus they share the same deduplication fingerprint 2-3-3. However, f_5 is completely different, but the deduplication fingerprint is the same as f_1 and f_3. So, deduplication fingerprint is not valid to uniquely identify files. Therefore, it is not actually a fingerprint, but it is a feature that can be used to distinguish files in most of the cases.

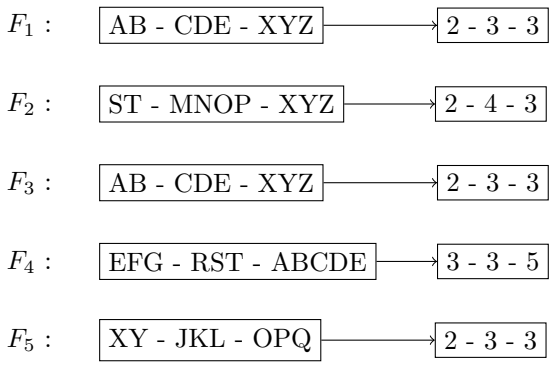


Figure 3.2: Deduplication fingerprints example.

3.2.2.2 Anonymity set

Definition 3.2. **[18] Anonymity set** The anonymity set of f is the set of all possible files that have the same deduplication fingerprint as f"[18]. That is, $A(f) = \{f' \in \{0,1\}^* : T(f') \simeq T(f)\}$.

The anonymity set (Definition 3.2) clearly depends on the chunking algorithm used in the system. Below it is explained how to quantify the anonymity set in file-based, fixed-sized blocks and content-based fingerprints.

Firstly, file-based deduplication consists only in one block, that is, the whole file. In this algorithm, the deduplication fingerprint of a file with n bytes will have exactly one node of size n. Therefore, all the files with the same size n will have the same deduplication fingerprint (although their contents are different).

Secondly, fixed-sized blocks deduplication consists in a set of blocks with the same size, except the last one, that contains the remaining bytes. Therefore, if we have a file f with n bytes, where B is the chosen block size, we will obtain n/B blocks with size B and one block (the last one) with size $n \ mod \ B$. Notice that all the files with the same number of blocks of size B and the same block size in the last chunk will have the same deduplication fingerprint.

Lastly, CDC consists in a set of blocks of variable size found taking into account the content of the file itself. Unlike previous schemes, a file f with size n could have different deduplication fingerprints depending on its content. Notice that CDC algorithm causes an additional leakage, that is, two files chunked using CDC will share the same deduplication fingerprint only if they have the same number of blocks and all the block sizes match. Besides, it is worth remembering that in CDC the blocks are separated depending on the file content. Therefore, if two files share the same deduplication fingerprint, the probability that both files are actually the same file is higher than in file-based or fixed-size blocks schemes.

3.2.2.3 Candidate set

Definition 3.3. [18] **Candidate set** The candidate set of f in G (denoted as $C(f, G)$) is the set of files stored on S— and hence represented in G— that have the same deduplication fingerprint as f.

To compute $C(f, G)$ the procedure consists in going through each root node of G. That is, we start initially with $C(f, G) = \varnothing$, then the deduplication fingerprint of each root node is checked. In the case that the deduplication fingerprint of the root node f' is the same as the deduplication fingerprint of f, f' will be a candidate and therefore f will be added to $C(f, G)$. When all G has been parsed, we have obtained the number of stored files with the same deduplication fingerprint as f.

3.2.2.4 Possible attacks

Taking advantage of the leaked information by the processes described above, a set of attacks may be performed. Firstly, $C(f, G)$ may provide some information about the stored files in the database without the knowledge of the encryption keys. For example, a straightforward leakage consists in checking the absence of a given file f. If $C(f, G) = \varnothing$ after parsing all G, it means that none of the stored files has the same deduplication fingerprint as f. Therefore, the adversary is certain that f is not stored in the database. However, if $C(f, G) \neq \varnothing$ does not mean that f is stored in the database, we only know that at least one stored file shares deduplication fingerprint with f.

To know more information about the presence of a given file in the database, the anonymity set is required. If the size of the candidate set is exactly the same as the anonymity set, it means that all possible files that have the deduplication fingerprint of f are stored on the database, and therefore, it is possible to conclude that the file is stored in the database. Otherwise, if the size of the candidate set is smaller than the size of the anonymity set, it is not possible to affirm that f is stored. It is

only possible compute the probability that the file f is stored, which it is denoted by Probability of Storage (PoS) [18].

To obtain PoS, the model is based in a Bayesian Network which consists of three random variables, X, D and G. Firstly, variable X represents the set of all files stored on S. The probability distribution of X depends on the file popularity and storage combinations (for example, some chapters of the same series has high probability to be stored together). Secondly, variable D denotes the deduplication algorithm used in the scheme (it is considered that it is known). Lastly, variable G represents the known storage graph. It is possible to obtain PoS as:

$$PoS(f, G_0, D_0) = P(f \in X | G = G_0, D = D_0)$$

3.3 Summary

Encryption is a desired process in cloud server providers, since clients outsource their confidential data to them. Following the Figure 3.3. Encryption process may be done in two different ways:

- **Conventional encryption**. Each user encrypts the files with his personal key before uploading the content to the server. This encryption scheme makes no possible cross-user deduplication process, since the same file uploaded by two different users will produce two different ciphertexts in the server.
- **Message-Locked Encryption (MLE)**. MLE is a deterministic encryption scheme where the key is generated from the file (or block) itself. Therefore, the same file uploaded by two different users will produce the same encrypted output. This feature makes possible cross-user deduplication in the server.

Previous literature has shown that it is possible to infer some information from the deduplication process both third parties and the cloud server provider itself. The leakage may be classified by its nature in two different types.

- **Client-side information leakage**. This leakage comes from the information that the client may infer about the deduplication process (if it occurs or not). Performing side-channel attacks, the adversary may know the presence or absence of a given file in the server. It is also possible to carry out template attacks. In client-side deduplication, side-channel attacks can be accomplished by third parties and the cloud server itself. However, in server-side deduplication, they can be only performed by the cloud server itself. It is worth mentioning

that these attacks can only be carried out in scenarios where the encryption is disabled or it is deterministic (e.g. MLE).

– **Access traces information leakage**. The leaked information in this vulnerability may be only inferred by the cloud server itself. The leaked information has less value than side-channel attacks, however, this information may be also obtained in scenarios where conventional encryption is used. It is possible to determine the absence and the Probability of Storage (PoS) of a given file.

Both encryption schemes will be tested in Chapter 6 in order to check and verify the information leakage described in this section.

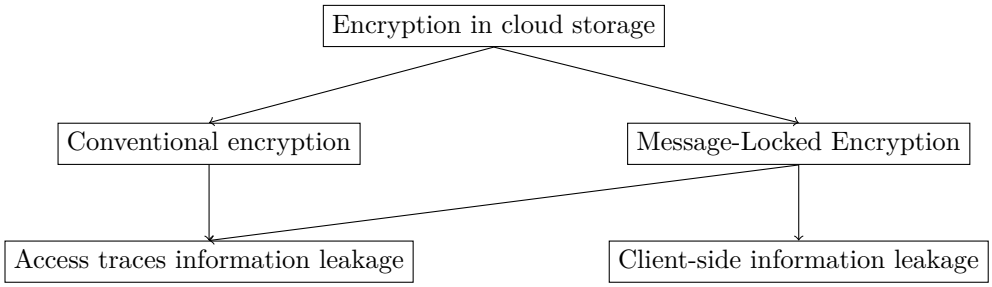


Figure 3.3: Encryption scheme summary

Chapter 4
Testing environment

In order to satisfy the project goals, a testing environment has been developed where several experimental tests will be performed afterwards. This testing environment fulfills all the needed requirements to allow us to carry out the desired experiments. This section presents a detailed description of the environment operation, its implemented features and usage documentation. In addition, the dataset that will be used to perform the experiment is explained, its features and how it has been generated.

4.1 General description

The system architecture represents a cloud storage provider where deduplication and encryption is enabled. Basically, the architecture is divided in two parts, the client-side and the server-side (the database belongs to the server-side). The server side represents the cloud storage provider itself, meanwhile the client side represents the software used by the clients to upload and download files. It is supposed that the server is located in the cloud, therefore, the server is shared for all the clients of the service. This scenario is called cross-user deduplication.

4.2 System features

In this section, it will be described the available characteristics which have been implemented in our testing environment in order to accomplish the proposed objectives.

4.2.1 Available chunking algorithms

The developed system allows choosing the chunking algorithm by the user. Four chunking algorithms are available:

- **Fixed block-size chunking**. In this algorithm the input file is separated into blocks of the same given size.

- **Basic Sliding window (BSW)**. BSW is one of the three implemented CDC algorithms in the system. BSW operation was explained in Section 2.2.3.1. In this algorithm, in order to check the condition, a hash function is required. The purpose of this hash function is not related to a security issue. The goal is to obtain a tag of the content as fast as possible to check if the tag satisfies a certain condition. Therefore, the security features of the hash function are not an important characteristic for this purpose. MD5 function has been selected in the implementation, but another hash function (e.g. SHA-256) or a rolling hash function (Rabin fingerprint Definition 2.1) could have been selected as well. In this implementation, the window content satisfies the condition when the last 12 bits of its fingerprint are zero. To set this condition, several tests were done in order to obtain a suitable value.
- **Two Threshold Two Divisors (TTTD)**. TTTD is one of the three implemented CDC algorithms in the system. TTTD operation was explained in Section 2.2.3.2. As BSW, MD5 was selected as a hash function to compute the window content. In this implementation, a content of the window satisfies the first condition when the last 12 bits of its fingerprint are zero. For the second condition, only the last 11 bits have to be zero. To set these conditions, several tests were done in order to obtain the suitable values.
- **Leap-based CDC algorithm**. Leap-based CDC algorithm is the last CDC implemented algorithm in the system. Leap-based algorithm operation was explained in Section 2.2.3.3. As in the others CDC implemented algorithms, MD5 was selected as a hash function to compute the window content. In the implementation, 10 windows are set, and all the windows must have a zero in their last bit to satisfy the condition. To set the condition and the number of windows, several tests were done in order to obtain the suitable values.

4.2.2 Available encryption schemes

It is also possible to choose the encryption scheme by the user. Two encryption schemes have been developed to be selected:

- **Conventional encryption**. In this scheme, the file is encrypted and decrypted with the user personal key. The user has to provide a personal key saved locally to the software. If the user does not have a key, the software will generate and save it in the user directory. AES operating in Counter Mode (CTR) is used as the block cipher for the encryption and decryption processes.
- **Message-Locked Encryption (MLE)**. This scheme was detailed in Section 3.1. In this algorithm, the user does not have to provide any user key to the software since the own algorithm will generate the keys from the file content.

On the one hand, to generate the keys and to tag the resultant blocks, SHA-256 is used as the hash function. On the other hand, for encryption and decryption processes, AES operating in Counter Mode is used. The counter CTR has a fixed value in this algorithm for all the users, because the algorithm has to be deterministic.

4.3 System documentation

The system documentation is divided into two subsections: upload and download protocols. The former refers to the processes between the user provides a file to the client software and the server stores it in the database, and the latter refers to the processes between the user requests the stored file and the user obtains it.

In both processes, the library Crypto++ [5] has been used to perform cryptographic operations. Crypto++ is a free C++ class library of cryptographic schemes where the individual files in the compilation are all public domain. Crypto++ functions have been validated by NIST and CSE for FIPS 140-2 level 1 conformance [14].

4.3.1 Upload protocol

In this section, the implemented upload protocol is described in detail.

4.3.1.1 Upload process description

In Figure 4.1 it is shown the upload process for MLE. The user passes by input the file that the user wants to upload to *upload.cpp*. This script acts as a software client and it has been developed in C++. *upload.cpp* chunks the input file (using a chunking algorithm selected by the user) and hashes the resultant blocks, obtaining K_1, K_2 and K_3 in the figure example. These hashes will be the keys to encrypt the file blocks, and they will be also stored on the user directory (they will be needed for the download protocol). To each encrypted block is wrapped a tag (T_1 T_2 and T_3) before connecting with the server. These tags are generated from the keys. In the server, it has been implemented a service for the upload process, called *store_file.php*, and it will have the role of saving the blocks on the database (a SQL table) performing the deduplication (note that in the Figure C_3 was already stored on there, and it is not stored again).

For conventional encryption, the process has small differences from MLE. In Figure 4.2 we can appreciate that the key is also a input that the user has to pass to the *upload.cpp* script. In case that the user does not pass the key, *upload.cpp* generates a new user key and stores it on the user directory. In this way, the user can use that key in future occasions. Another difference from MLE process is that

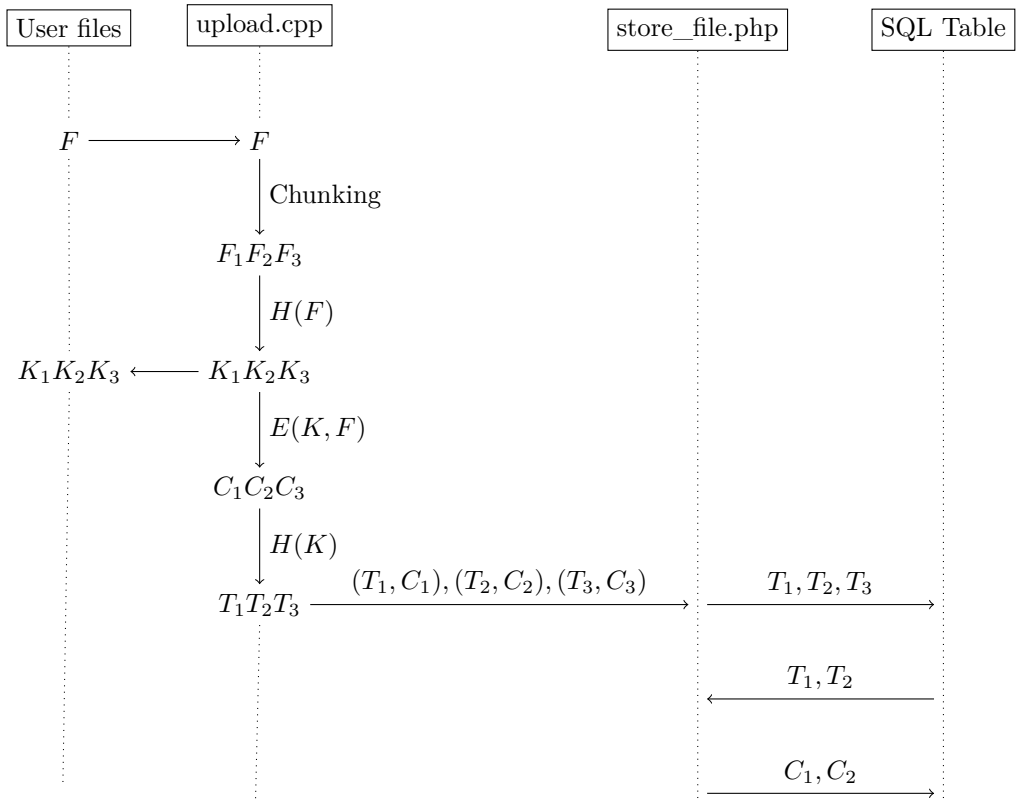


Figure 4.1: MLE: Implemented upload protocol

the tags are generated from the encrypted blocks, and these tags are also stored on the user directory (in this case, the tags will be the input for the download process).

4.3.1.2 Argument management

In order to upload and store a certain file, the script *upload.cpp* has to be called from the command line or a proper Integrated Development Environment (IDE) providing the appropriate value arguments.

- **input_file**. The target file path has to be passed. This argument is compulsory.
- **keygen_mode**. This argument is in charge to set the encryption scheme. If a personal key path is passed, the system will use it in a conventional encryption scheme. However, if the value *conv* is passed, the system will use a conventional encryption scheme as well, but this time a new personal key will be generated

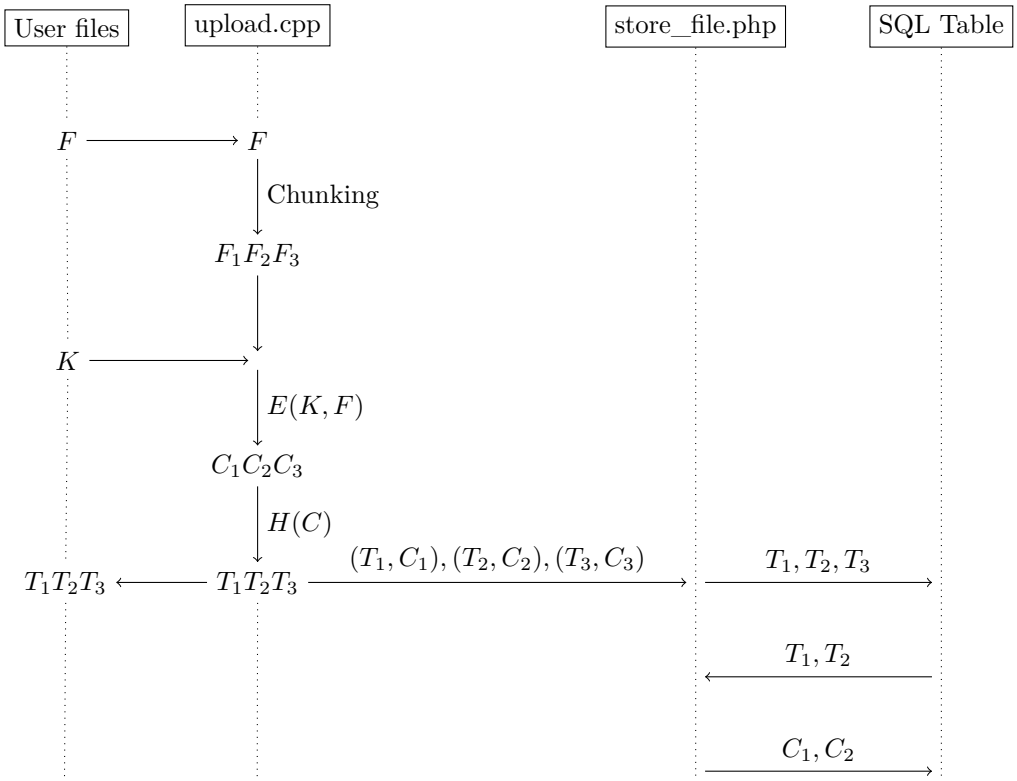


Figure 4.2: Conventional encryption: Implemented upload protocol

and saved in the user directory, for future occasions. Lastly, if the value *mle* is passed, Message-Locked Encryption scheme will be used. This argument is also compulsory.

– **chunking_algorithm**. This argument is used to select the chunking algorithm. The possible values are *fixed* to fixed block-size chunking algorithm, *bsw* to Basic Sliding Window (BSW), *tttd* to Two Threshold Two Divisors (TTTD) and *lb* to leap-based CDC algorithm. This argument is also compulsory.

– **chunk_size**. This argument refers to a fixed chunk size selected by the user. This argument is optional, and it is only used if the value *fixed* is set as chunking algorithm. 8kB is the default value.

– **min_chunk_size**. This argument is used to set a minimum chunk size. It is optional, and it is only used in CDC algorithms. 4kB is the default value.

– **max_chunk_size**. This argument is used to set a maximum chunk size. It is optional, and it is only used in CDC algorithms. 12kB is the default value.

4.3.1.3 Client-server data exchange

The script *upload.cpp* is in charge of splitting a given file and encrypting the resultant blocks. In addition, it associates each block with one tag. The script will create, in the user directory, a new text file with these tags (this information will be required in the download protocol in order to recover the file from the server). When this task is finished, *upload.cpp* sends all this information to the cloud server. Because it is only a testing environment, localhost is used as a cloud server for this purpose, but any other server could be used as well. A service to receive and store files from different users has been developed in the server, called *store_file.php*.

Therefore, a JSON is sent using a POST request from *upload.cpp* to *store_file.php*. This JSON is filled with an array of JSON Objects, each one with two properties, *tag* and *content*.

4.3.1.4 Data storing service

In the cloud server (in our case, hosted in localhost), a MySQL database has been created in order to store all file blocks uploaded from *upload.cpp*. This database includes one table, called *StoredBlocks*, where the incoming data is stored. This table has two columns, *tag* and *content*. The former is the PRIMARY KEY, and it is used to identify the latter. The SQL query to create the table is as follows:

```
CREATE TABLE StoredBlocks (
    tag VARCHAR(255) NOT NULL PRIMARY KEY,
    content MEDIUMTEXT NOT NULL
);
```

To save new data, it is checked if the content block is already stored, in order to perform the deduplication process. Below it is shown the SLQ query used to accomplish this task.

```
INSERT INTO StoredBlocks (tag, content)
VALUES ('tag_value', 'content_value')
ON DUPLICATE KEY UPDATE tag=tag;
```

4.3.2 Download protocol

In this section, the implemented download protocol is described in detail.

4.3.2.1 Download process description

In Figure 4.3 it is shown the download process for MLE scheme. The generated keys in the upload protocol are the input for this process. *download.cpp* script generates again the tags that correspond to the uploaded blocks and it request them to *retrieve_file.php*, an implemented service on the server for this purpose. When *download.cpp* receives the encrypted blocks from the server, it only needs to decrypt them (using the passed keys) and joins the blocks to obtain the original file again.

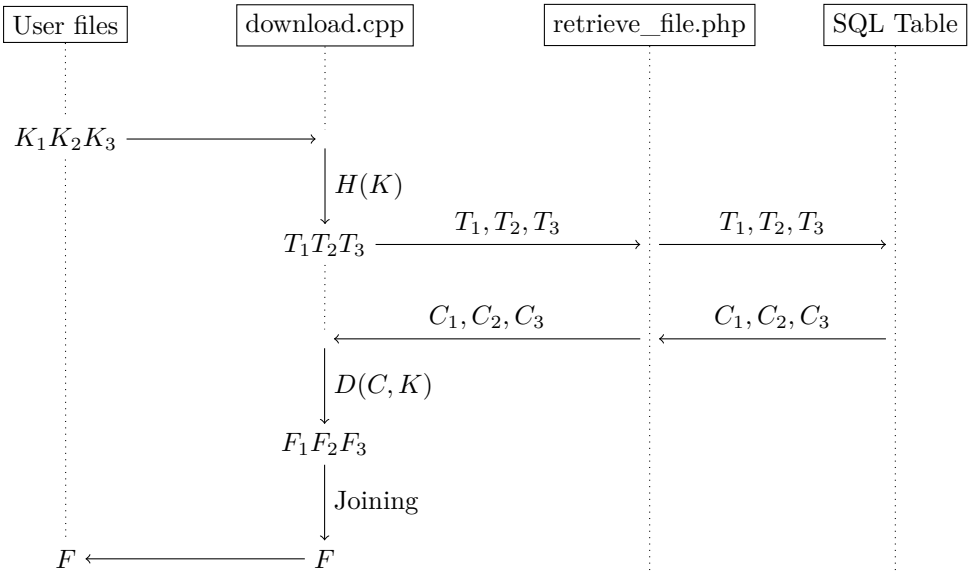


Figure 4.3: MLE: Implemented download protocol

The process does not change too much for conventional encryption scheme (Figure 4.4), where the tags generated from the upload process are passed to *download.cpp*. This time, the decryption process is done with the user key, which is necessary to be passed to *download.cpp* as well.

4.3.2.2 Argument management

In order to download a file from the server, the script *download.cpp* has been developed. This script may be called from the command line or an Integrated Development Environment (IDE) providing the appropriate value arguments. These arguments are as follows:

– **tag_file**. The path of the generated tag file in the upload process. This

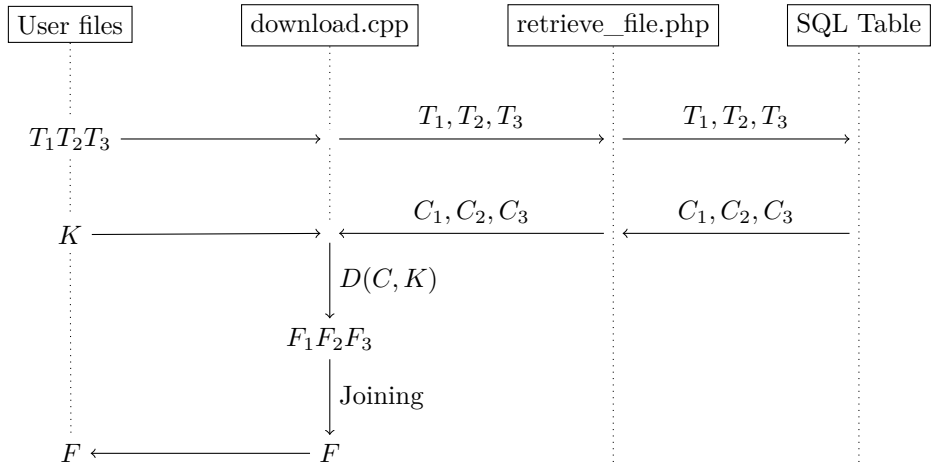


Figure 4.4: Conventional encryption: Implemented download protocol

information allows the software knowing which file is requested to be download. This argument is compulsory.

- **decryption_mode**. This argument refers to the decryption scheme. There are two possible values: *conv*, to use conventional encryption, and *mle*, to use Message-Locked Encryption. This argument is also compulsory.

- **key_file**. The path of the personal key. This argument is only required in conventional encryption scheme.

4.3.2.3 Data retrieval service

The script *download.cpp*, after reading the arguments, prepares a request to the server with the tags provided by the user, using JSON format. In the server, it has been developed a service hosted, called *retrieve_file.php*. In this service, the server gets the tags provided by *download.cpp* and checks for their existence in the table *StoredBlocks*, retrieving their associated content. The SLQ query used is as follows:

```
SELECT content FROM StoredBlocks
WHERE tag = 'tag_value'
```

The service *retrieve_file.php* fills a JSON similar than the one used in the upload protocol, with two properties, *tag* and *content*. This information is sent to the user script *download.php*, where the blocks will be decrypted and the original file will be reconstructed.

4.4 Dataset generation

The dataset represents the set of files that will be uploaded to the cloud storage server. These files have to satisfy several conditions in order to obtain relevant results in the experiments. The dataset requirements are as follows:

- The files should have overlap between them. That is, the files have to be compound with similar data, because it is valuable that deduplication occurs in the experiments. It is worth remembering that one of the goals of the project is measuring the deduplication performance in different chunking algorithms, and it will not be possible to fulfill if it does not exist overlap between files.
- The dataset has to be large enough in order to obtain relevant results.
- The file size distribution has to be similar to a real cloud storage dataset, to reproduce a reliable scenario.

Considering the requirements mentioned above, the selected procedure to generate the dataset is divided in two stages. Firstly, a large file with 5 GB of random data is generated. Secondly, the dataset files are created from the content of the large file, therefore, the content of the files will be small parts of the large file and it will not be difficult to find overlap between these files.

The creation of each file consists of two random variables, *startpoint* and *filesize*. The former means the point that the file begins to read over the large file. For example, if the *startpoint* of the file F is 3.128, it will mean that the file starts in the byte number 3.128 in the large file. The latter means the file size, that is, the number of bytes that it will read in the large file starting from the *startpoint*.

The *startpoint* variable may be completely random in the range (0, 5GB), but *filesize* variable has to follow several patterns in order to reproduce a real cloud storage server scenario. It has been used as a reference the dataset used in Ritzdorf et al. document [18]. That dataset corresponds to 13.4 TB of data extracted from a subset of a publicly available collection.

Finally, it has been created 50.000 files, generating a dataset of 78.8 GB with an average file size of 4 MB. A detailed distribution of file sizes is shown in Table 6.2:

Table 4.1: Dataset file size distribution

File size range	Number of files
All	50.000
1B - 1KB	12.500
1KB - 4KB	12.500
4KB - 8KB	5.000
8KB - 16KB	4.500
16KB - 32KB	3.500
32KB - 100KB	3.500
100KB - 300KB	3.500
300KB - 1MB	3.500
1MB - 10MB	1.000
10MB - 100MB	400
100MB - 1GB	100

Chapter 5
Chunking algorithms experiments

The chunking stage in every deduplication scheme has great importance in order to save storing resources in the cloud. In this section it is explained the performed experiment in order to compare the four developed chunking algorithms: fixed block-size, Basic Sliding Window (BSW), Two-Threholds Two-Divisors (TTTD) and leap-based CDC algorithm.

5.1 Experiment description

For this experiment, the generated dataset 4.4 will be uploaded to the cloud storage server using different chunking strategies. During the upload process, several parameters will be caught in order to extract conclusions about the deduplication performance that provides each algorithm. The four algorithms used in the experiment are explained in Section 2.2, and they are:

- **Fixed size blocks chunking** With a block size of 8KB.
- **Basic Sliding Window (BSW)** With minimum block size of 4KB and maximum block size of 12KB, getting a block size average of around 8KB.
- **Two-Thresholds Two-Divisors (TTTD)** As in BSW, it has been set 4KB and 12 KB as minimum and maximum block sizes, respectively.
- **Leap based CDC** As in the others CDC algorithms, it has been set 4KB and 12 KB as minimum and maximum block sizes, respectively. This values will generate a block size average of around 8KB.

In the upload process, the parameters that it has been measured are as follows:

- **Required time** This variable refers to the CPU time that is necessary to upload the whole dataset using a particular chunking algorithm. The CPU time

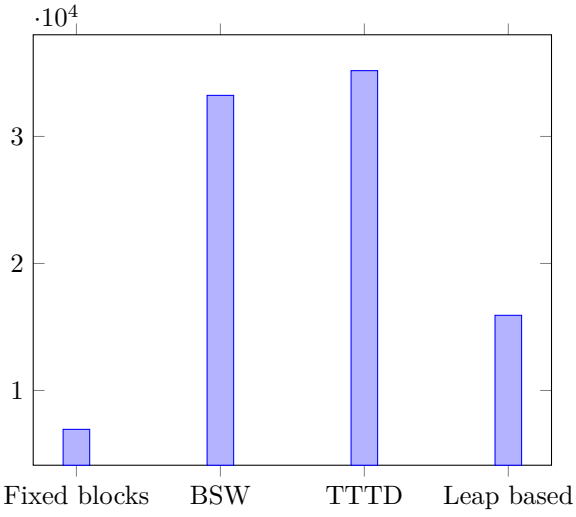
is the amount of seconds that the CPU needs to complete the process. Notice that this time may be different depending on the computer where the process is executed. Because of it, the dataset has been uploaded in the same computer for every chunking algorithm. Therefore, for this parameter, the relation between the different times for each algorithm will have more importance than the number of seconds that it has been required to complete the process.

- **Deduplication performance** This variable refers to the percentage of blocks that it has not been stored in the database because they already exist. That is, the percentage of removed blocks. Higher deduplication performance results in a more efficient way to save storage resources.

- **Percentage of forced blocks** This variable is only measured in CDC algorithms. It refers to the number of blocks that has the maximum size (forced blocks). The purpose of a CDC algorithm is finding block boundaries depending on the content of the file, and when the maximum size is reached, the boundary is forced and it does not respond to a file content reason. Measuring this variable will allow us to conclude if a higher number of forced blocks may make the final deduplication performance worse.

5.2 Results and discussion

In Figure 5.1 it is shown the required time to upload the whole dataset for each chunking algorithm.



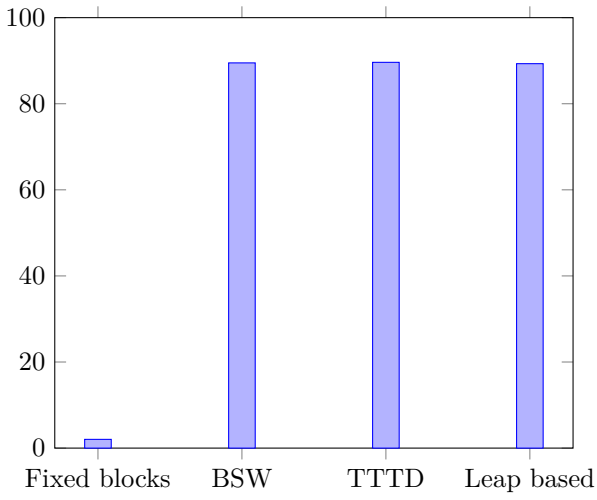
algorithm	time(s)
Fixed blocks	6.940
BSW	33.234
TTTD	35.180
Leap based	15.918

Figure 5.1: Required time to upload the dataset for each chunking algorithm

As we can appreciate, the fastest algorithm is fixed block-size chunking. This is because it is the simplest one. To implement fixed block-size algorithm is not necessary reading byte by byte the target file, and the computational cost to run is smaller than in the rest of the algorithms. With this algorithm, all the files were uploaded to the cloud server in 6.940 seconds.

Regarding CDC algorithms, they needed more time to finish the task. This fact is more dramatic for BSW and TTTD algorithms, which this task took around 5 times more time than fixed block-size algorithm to be completed. It is worth remembering that these algorithms consist of a sliding window which parses all the file byte by byte, computing one (BSW) or two (TTTD) hash functions for each position, and checking if the window content satisfies a condition. This fact is reduced in leap based CDC algorithm. This algorithm took around 2,3 times more time than fixed block-size algorithm. Leap based mechanism is similar than the other two analyzed CDC algorithms, but this one is focused on reducing the computational cost of CDC algorithms. Leap based mechanism permits skip several bytes in the parsing process, with the purpose of improving the efficiency. The obtained results show that indeed leap based algorithm fulfills their purpose, being the more efficient CDC algorithm with regard to execution time.

In Figure 5.2 is shown the deduplication performance for each chunking algorithm.



algorithm	% dedup.
Fixed blocks	2,03
BSW	89,49
TTTD	89,62
Leap based	89,31

Figure 5.2: Percentage of deduplicated blocks.

At first sight, it is clear the difference between the deduplication ratio obtained from fixed block-size algorithm and any of the CDC algorithms. Only 2,03% of the blocks have been deduplicated in fixed block-size algorithm, while in CDC algorithms

this number comes to 89% approximately.

With regard to the low percentage obtained in fixed block-size algorithm, it is due to the boundary shifting problem (explained in Section 2.2.2). This issue has been specially relevant in the experiment as a result of how the files have been generated.

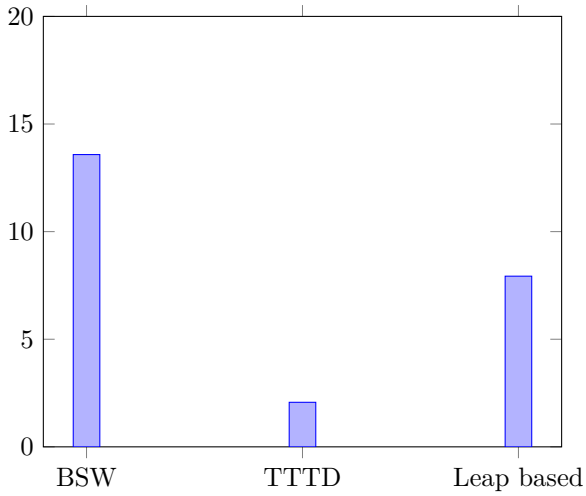
Boundary shifting problem is more significant when two files overlap in their content but they do not start at the same point. For example, if we have two files, F_1 and F_2, and both share exactly the same content, except from 1 byte at the beginning of F_2. The resultant number of deduplicated blocks will be zero because all the blocks are shifted by one byte.

To generate the dataset (Section 4.4), all the files (78,8 GB) are extracted from the same 5GB of content, but each file begins in a different start point of that 5GB of data, (this fact adds a difficulty and it is useful to check the behavior of each algorithm). In case of fixed the algorithm ineffective since the boundary shifting problem appears in almost all the blocks, failing in its purpose of remove duplicated content.

Notice that if all the dataset files have been generated starting from the same byte, the number of deduplicated blocks would have been much higher. In real datasets, where entire files share the same content, fixed block-size algorithm will have resulted in higher deduplication ratio as well.

With reference to CDC algorithms, the deduplication ratios obtained are similar between them. The percentage is high for all of them (around 89,5%) and it means that they have been able to manage correctly the boundary shifting issue which has caused a very low ratio in fixed block-size algorithm. In these algorithms, the boundary shifting issue only affects the first block of the file. These first blocks could have been the cause that the deduplication ratio is not higher. It is worth mentioning that the maximum deduplication ratio that it could be possible to reach is 93,65% in this experiment. That is because the used dataset (78,8 GB) comes from the same 5GB. Therefore, the minimum size that the server could store for this task is 5GB, and this number corresponds with a deduplication ratio of 93,65%.

In Figure 5.3 it is shown the percentage of forced blocks in each CDC algorithm. TTTD algorithm was created with the purpose of reducing the number of forced blocks. A forced block is a block with the maximum size, that is, blocks with forced boundaries instead of boundaries generated from the file content. The results show that TTTD purpose is reached, for this algorithm the percentage of forced blocks is only 2,07% of the total blocks. However, BSW and Leap based algorithms produce more forced blocks, 13,58% and 7,93% respectively.



algorithm	% forced blocks
BSW	13,58
TTTD	2,07
Leap based	7,93

Figure 5.3: Percentage of forced blocks (not content defined) for each CDC algorithm.

In spite of this, as we have seen in Figure 5.2, the deduplication ratios are very similar between them, and the possible differences originated from this fact has been irrelevant in this experiment.

To sum up, this experiment has shown that it is necessary that a chunking algorithm has to be based on the content of the file, generating boundaries from characteristic of the content itself. We have seen that fixed block-size algorithm has not satisfied the deduplication performance required for a cloud storage server application. This algorithm could be useful in particular applications, but never in a generic storage server.

With regard to the different CDC tested algorithms, the number of deduplicated blocks has been very similar, but leap based CDC algorithm has needed around 0,5 times less time to complete its task for all the files belonging to the dataset. Therefore, we may conclude that leap based CDC algorithm is suitable for cloud storage server applications, because it offers a good trade-off between deduplication performance and computational cost.

Chapter 6
Information leakage experiments

In this chapter the experiments related with the information leakage are explained. A curious cloud storage system, where deduplication is enabled, may obtain information about the stored data even when they is encrypted. In the experiments, two schemes are analyzed: Conventional encryption (each user has their personal key) and Message-Locked Encryption (MLE).

6.1 Information leakage in MLE scheme

6.1.1 File presence/absence attack

The first experiment that has been done in MLE encryption scheme consists of checking the presence or absence of a particular file in the server. In this experiment, it is supposed that the cloud server is curious about the uploaded files, in order to guess enough information about the files to verify if a certain file is stored on it.

To perform the experiment, firstly, the generated dataset (Section 4.4) is uploaded to the curious cloud storage server. In this process, the cloud server does not need to execute another process in parallel. That is, the server only has to carry out the deduplication procedure, associating a tag to each uploaded encrypted file.

Once all the files have already been stored, the server will check if two different files are stored on it. One of the files (F_1) belongs to the dataset, but not the other (F_2).

After the execution of the attack, the results were successful as expected. The curious server only had to encrypt the files F_1 F_2 with the keys generated from the files themselves, and hashing the encrypted files to check if the obtained tags are stored on the server.

6.1.2 Template attack

For this experiment, an enterprise contract template is created. This template consists of header information related to the enterprise, the name and salary of the employee. It is assumed that the curious storage server knows the contract template. As an example, the server administrator is an employee of the enterprise, and the contract template is the same for all the employees.

The curious server wants to know the salary of John Doe, an employee of the enterprise. Therefore, the server knows the header information, the field "Employee name: John Doe", and the salary "Salary: X\$", where X is the value that the server will try to guess performing a brute-force attack.

For each attempt, the storage server sets a value for the salary and it hashes the resultant file in order to obtain the key. Then, the server encrypts the file and it hashes the result, to get the tag. If the tag exists in the server, it means that the selected salary is indeed the right salary.

In the experiment, in each attempt, the salary was the previous computed value plus 10\$, starting from 700\$. John Doe's salary was set on 2000\$. The server took 63 seconds in guessing the value.

6.1.3 Discussion

It is worth remembering that the implemented cloud service does not give any information to the user about the deduplication process. Although the chunking process is done in the client side, the user sends the tags and the block files at the same time (the user does not send first the tags, and the block files afterwards, as in Figure 2.2). Therefore, these attacks can only be performed by the storage server in our case. Notice that, if a client-side deduplication system as in Figure 2.2 is developed, any third party (any attacker different from the cloud storage server) could accomplish both experiments because they could have access to the same information about the deduplication process that the cloud storage server has.

In our testing environment, if an attacker (different from the storage server itself) desires to get some information about the uploaded files from a certain user, he will have to eavesdrop the channel between the selected victim and the storage server. Then, the attacker could reproduce a storage database with the sniffed information (notice that this database will have information uploaded only from a particular victim). From this point, the attacks will have the same steps as mentioned above, but this time they will be targeted against the content uploaded by a selected user, instead of all the cloud service clients.

These attacks may be carried out due to the deterministic nature of MLE

encryption scheme. This deterministic feature allows the cross-user deduplication and that makes MLE valuable. That is, if several countermeasures which avoid the deterministic MLE characteristic are introduced (e.g. adding randomization in key generation process), MLE encryption will not able to gather deduplication and encryption in the same architecture, and it will be useless.

In spite of this, several countermeasures could be introduced to reduce the impact of these attacks. For instance, a trusted key server could be introduced as a third party. In this new architecture, the user will not generate the key himself and he will need to request a key for each file to a key server, the latter will always generate the same key for the same file. Notice that this architecture is also deterministic, but this time, the key server could deny the connection coming from a particular client if the latter is doing a high number of requests in a short period. In this way, the trusted key server could be used to avoid brute-force attacks (e.g. template attacks).

6.2 Information leakage in conventional encryption scheme

Conventional encryption scheme (where each user has their own key) does not allow data deduplication and, therefore, it does not permit a correct way to save the resources on the system. In spite of this, it is interesting to test the leakage related with this encryption scheme to compare it with the information leakage in MLE.

As we have explained in Section 3.2.2, a curious storage server is able to extract information about the stored files from the access traces generated in the communication between client and itself. In the scenario proposed by Ritzdorf et al. [18], it is supposed that the curious storage server cannot guess or acquire the encryption keys, and it may only observe limited information related with the communication packets (as object ID, object size and timestamp).

6.2.1 Experiment description

For this experiment it is worth remembering the concepts *deduplication fingerprint* (Definition 3.1) and *candidate set* (Definition 3.3). As we have seen in Section 3.2.2, in this scheme it is possible to know the absence of a file (if a file is not stored) as long as there are not collisions between deduplication fingerprints. Therefore, it is interesting to test if deduplication fingerprint collisions are common enough to disguise this leakage or they are unlikely and therefore, the leakage is completely feasible.

The experiment is based in a test called *cross-dataset validation* performed in Ritzdorf et al. document [18]. Due to the similarities between both experiments, the

obtained results will be compared with the results in Ritzdorf et al. document [18] in the discussion.

To perform the experiment, several changes were needed on the implemented server to transform it in a curious cloud storage server. In the new server, when a user uploads a file, the server computes its deduplication fingerprint and it is stored in a table called "Trees". This table will be checked each time that a user uploads a file. That is, the storage server will compute the number of candidates (number of deduplication fingerprint collisions) for each uploaded file by a user. In this way, if the number of candidates of an uploaded file is zero, it will mean that the file did not exist previously in the database. However, if the number of candidates obtained are one or more, it will mean that the file could be stored in the database previously, because one or more deduplication fingerprint collisions have been produced. In this case, it is not possible to ensure the presence or absence of the file. The process to set up the experiment was as follows:

- The generated dataset 4.4 (Dataset 1) will be uploaded to the implemented curious cloud storage server. In this upload process, each deduplication fingerprint related with each file will be stored into the table "Trees" on the server.

- A second dataset is generated (Dataset 2). All the files belonging to this dataset are different from the files in dataset 1. Thus, we previously know the absence of the dataset 2 files in dataset 1. The experiment will measure the frequency of deduplication fingerprint collisions in several file size ranges. In the dataset, the same number of files are set for each range. In total, 13 ranges have been set, generating in this way a line with 13 points in the graph. Further information related with file size distribution of dataset 2 is showed in Table 6.1.

- Dataset 1 and dataset 2 are uploaded for each deduplication algorithm implemented: Fixed block size algorithm, Basic Sliding Window (BSW), Two-Thresholds Two-Divisors (TTTD) and leap-based CDC. Therefore, each algorithm will mean a different curve in the resultant graph. For each file in dataset 2, the server will check the number of candidates for each uploaded file (notice that we previously know that all the files were not stored, and we can compute the precision of the attack). If the number of candidates is zero, it will mean that the file was not stored and the leakage has been achieved. However, if the number of candidates is one or more, it will not possible to infer the absence of the file, and the leakage is not achieved.

To sum up, for each file of dataset 2 we record the number of candidates in dataset 1, and determine the precision of the leakage knowing that the files in dataset 2 are

Table 6.1: Dataset 2 file size distribution where 8KB is the selected block size in fixed block size algorithm, and the range between 4KB and 12KB is the selected block size range in CDC algorithms.

File size range	Number of files
1B - 1KB	250
1KB - 4KB	250
4KB - 6KB	250
6KB - 8KB	250
8KB - 12KB	250
12KB - 16KB	250
16KB - 24KB	250
24KB - 32KB	250
32KB - 64KB	250
64KB - 100KB	250
100KB - 300MB	250
300KB - 1MB	250
1MB - 10MB	250

different from the files of dataset 1. So that, if the obtained number of candidates is zero, it means that the leakage works, and it does not work in any other case.

6.2.2 Experiment results

The obtained results are showed in Figure 6.1. The horizontal axis represents the file size. The vertical axis means the percentage of precision of the leakage, in other words, the percentage of files that the number of candidates obtained was zero (deduplication fingerprint collisions were not produced).

We may observe that all CDC algorithms present a similar curve, being different from fixed block-size one. This fact means that the selection of one of the three CDC algorithm implemented does not imply a different behavior in this graph. For this reason, CDC algorithms will not be discussed separately this time, and they will be grouped together as CDC algorithms.

Regarding CDC algorithm curves, they remain on zero until 2KB. Notice that the minimum block size is 4KB in the experiment, so that, in files with sizes lower than 4KB, their deduplication fingerprints consist of only one node with the file size. This fact causes that deduplication fingerprint collisions are common in this range, making the leakage inefficient in these file sizes (none of the uploaded files in this range returned zero candidates).

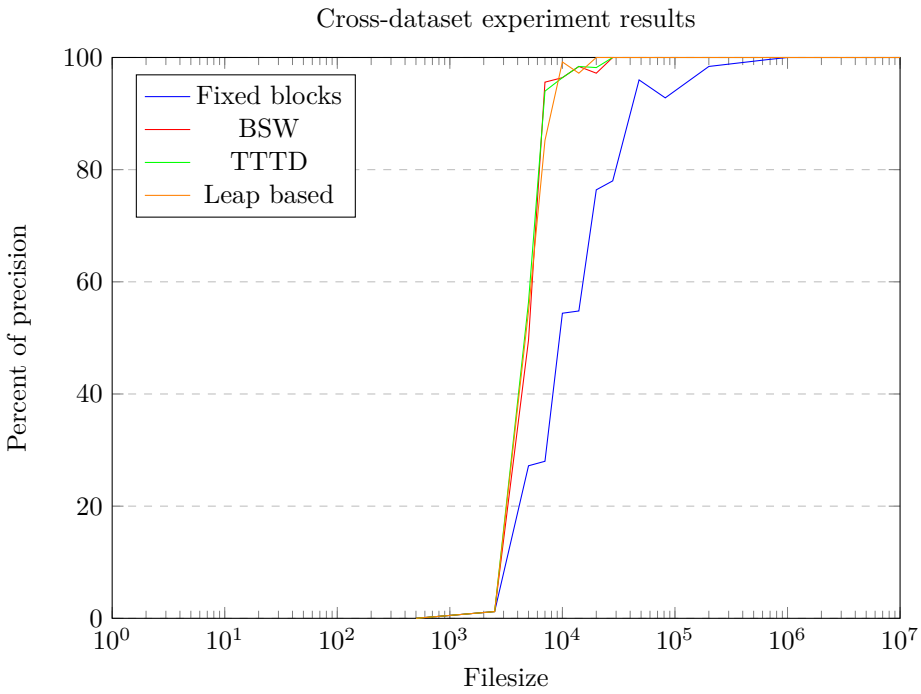


Figure 6.1: Cross-dataset experiment results. Block size 8KB in Fixed blocks and blocks between 4KB and 12KB in CDC algorithms.

Table 6.2: Cross-dataset experiment results. Block size 8KB in Fixed blocks and blocks between 4KB and 12KB in CDC algorithms.

File size range	Fixed blocks	BSW	TTTD	Leap based
1B-1KB	0	0	0	0
1KB-4KB	1.2	1.2	1.2	1.2
4KB-6KB	27.2	49.6	56.4	54.8
6KB-8KB	28	95.6	95.6	85.2
8KB-12KB	54.4	96.4	96.4	99.2
12KB-16KB	54.8	98.4	98.4	97.2
16KB-24KB	76.4	97.2	98.2	100
24KB-32KB	78	100	100	100
32KB-64KB	96	100	100	100
64KB-100KB	92.8	100	100	100
100KB-300KB	98.4	100	100	100
300KB-1MB	99.6	100	100	100
1MB-10MB	100	100	100	100

However, the situation changes in a short range, between 4KB and 8KB, where the percent of precision increases dramatically from 1,2% to 90% approximately. In this range, two files with the same size could have different deduplication fingerprints. This fact does not occur in file sizes lower than 4KB (minimum chunk size). For instance, a file with size of 5867 B could have these deduplication fingerprints: (5867, $4000-1867$, $4001-1866$, $4002-1865$, ... or $5866-1$). For this reason, as we increase the file size, the collisions are more unlikely and the precision of the leakage is higher. For file sizes more than 28KB, we can appreciate that the percent of precision is 100%. This means that the number of candidates was zero for each file higher than 28KB.

With regard to fixed block-size algorithm, the general shape is similar than CDC algorithms. But this time, the growth is done afterwards, that is, the percent of precision reaches 100% later (around 1MB). It means that the leakage is less probable than CDC algorithms in files with sizes between 8KB and 1MB. Therefore, fixed block-size algorithm disguises better than CDC algorithm against this leakage in medium-sized files. That is because the probability of collision between two files is higher using file block-size algorithm. As an example, two different files with size of 24016B, chunked with a block size of 8000B, they always will have the same deduplication fingerprint ($8000-8000-8000-16$, in this case). Notice that in CDC algorithms that fact does not occur because of the block can have any chunk size between 4KB and 12KB in the experiment.

As it was pointed before, this experiment was also done in Ritzdorf et al. document [18] with another dataset. The results obtained by them are shown in Figure 6.2. It is worth mentioning that the dataset used in that paper is 160 times larger than our dataset.

As we may appreciate, they are also comparing file-based deduplication. But for our experiment, we will focus only in CDC and fixed block-size results.

At first sight, we may observe a similar behavior of the curve which represents CDC. Nevertheless, the curve which represents fixed block-size has different shape. This difference comes from the dataset size. The more is the number of files, the more is also the probability of two files have exactly the same size, and therefore, the same deduplication fingerprint (only in fixed block-size algorithm case). Thus, we may notice that the number of file affects more in the percent of precision of this leakage for fixed block-size algorithm, however, in CDC algorithms this fact has less influence.

From this experiment, we can conclude that it is possible to know the absence of a file in a cloud server in most of the cases when the file is larger than 2-3 times the average block size in CDC algorithms. However, if the cloud server uses fixed

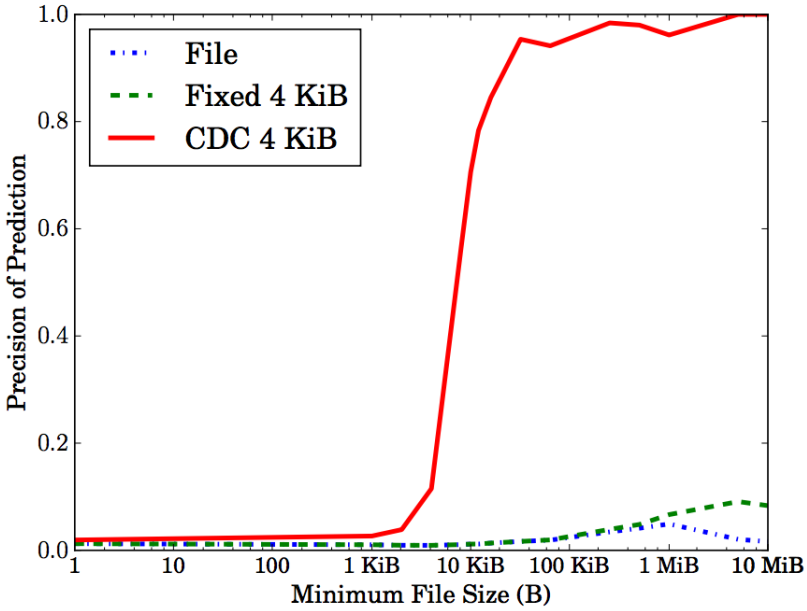


Figure 6.2: Cross-dataset experiment results. Fixed block-size of 4KB and block size average of 4KB for CDC algorithm. Source: Ritzdorf et al. document [18]

block-size algorithm to split the files in blocks, it will be required larger files to have more probability of success (zero candidates for a given file, that proves the absence of the file).

In order to analyze the possible countermeasures to avoid this leakage, it is worth recalling that the leakage comes from the possibility that a curious storage server has to know the right file size from the encrypted file. It is also worth stressing that the storage server has not the control of the encryption mechanisms performed by the user to encrypt the file, because the user sends the file encrypted by herself. Therefore, the user could add some bits at the beginning or at the end of the encrypted file in order to disguise the real size of the file. In this way, if the curious storage server desires to know the absence of that file, it will need to add the same number of bits at the beginning or at the end in order to obtain the same file size than the uploaded file to perform the attack. But this information is unknown by the storage server.

Chapter 7
Conclusions

Cloud storage providers are able to reduce their storage costs taking advantage of data deduplication procedure. Namely, they store the same content only once, removing duplicated copies and saving storage costs and bandwidth.

In this document, data deduplication process has been analyzed in detail. In addition, it has been studied the information leaked due to deduplication procedure to a curious storage server for different encryption schemes. Regarding the objectives indicated in Section 1.1, the concluding remarks are showed as follows:

- A **testing environment** has been implemented to test encryption schemes in a cloud storage service architecture. This environment has the capabilities to upload files to a storage server and download them from it. Besides, it allows to select the desired chunking algorithm to separate the files in blocks between the four available algorithms (fixed block-size, BSW, TTTD and leap based CDC) and it also permits to choose the desired encryption scheme between conventional encryption (each user has their own key to encrypt) and MLE. This testing environment has resulted useful to develop the experiments.

- **Data deduplication** process has been analyzed in detail. Four different chunking algorithms have been implemented and tested. After the experiments, we have confirmed that CDC algorithms have better deduplication ratio than fixed block-size algorithms, because they are able to find duplicates in a more effective way. This difference has been large enough to discard fixed block-size algorithm as a suitable algorithm to separate file in blocks in order to search duplicates. We also have noticed that CDC algorithms require more computational cost than fixed block-size. In this aspect, leap-based CDC algorithm has resulted the most efficient algorithm against BSW and TTTD (the other two CDC algorithms analyzed), where the computational cost was too much high.

- **MLE** scheme has been tested in order to analyze the information leaked in this architecture. As a result of the experiments, we may conclude that a curious cloud storage server may know the absence or presence of a given file. In addition, it is possible to carry out a template attack in order to guess certain information of a file when the attacker has the template which that file comes from. Besides, if client-side deduplication scheme has been selected, these attacks can be accomplished by any user of the cloud server. For all these reasons, we may conclude that the information leaked by MLE scheme is significant enough to consider it as an insecure scheme.
- **Conventional encryption** scheme has also been tested in order to compare the information leaked by this scheme and MLE. As a result of the experiments, we may conclude that in conventional encryption scheme a curious cloud storage server could know the absence of a given file. This leakage is more effective if CDC has been selected as chunking algorithm. If fixed block-size has been chosen as chunking algorithm, the leakage is more effective is the file size is large. As we may observe, this leakage is less significant than in MLE, but it is also possible to infer some information in this scheme.

After the experiments, on the one hand, MLE has shown to be an insecure scheme. In spite of this, the trade-off between confidentiality and resource savings could be valuable depending on the application. On the other hand, the information leaked from conventional encryption scheme is less significant, but this scheme does not allow an efficient way to save resources in a cloud storage server, disabling cross-user data deduplication.

For all these reasons, we may conclude that conventional encryption could be used in files where the confidentiality plays an important role. In addition, in order to avoid the leakage, several bytes could be introduced at the beginning or the end of the file, in order to disguise the real file size. For the rest of the files, MLE scheme could be used in order to save resources on the server. Therefore, the use of one of them will depend on the target application.

However, it is possible to continue working in this topic. Regarding data deduplication process, the design of a new chunking algorithm which may reduce the computational cost could be interesting in order to improve the existing algorithms. In addition, with regard to the coexistence of encryption and deduplication in cloud servers, MLE presents a relatively good trade-off between confidentiality and resource savings, but as we have shown in the conclusions, this scheme still presents several security issues. The development of another encryption scheme, or an improvement of MLE could be also interesting in order to test and analyze their security mechanisms and the possible leakage that they might incur.

References

[1] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O Karame, and Franck Youssef. Transparent data deduplication in the cloud. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 886–900. ACM, 2015.

[2] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Dupless: Server-aided encryption for deduplicated storage. *IACR Cryptology ePrint Archive*, 2013:429, 2013.

[3] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 296–312. Springer, 2013.

[4] William J Bolosky, John R Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 28, pages 34–43. ACM, 2000.

[5] Cryptopp. Cryptopp. free c++ library for cryptography. https://www.cryptopp.com/, 2016.

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

[7] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[8] Guy L Fielder and Paul N Alito. File encryption method and system, April 11 2000. US Patent 6,049,612.

[9] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[10] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[11] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick PC Lee, and Wenjing Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE transactions on parallel and distributed systems*, 25(6):1615–1625, 2014.

[12] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[13] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[14] NIST. National Institute of Standards and Technology. Security management & assurance. http://csrc.nist.gov/groups/STM/index.html, 2016.

[15] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[16] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. Block-level de-duplication with encrypted data. *Open Journal of Cloud Computing (OJCC)*, 1(1):10–18, 2014.

[17] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. Perfectdedup: Secure data deduplication. In *International Workshop on Data Privacy Management*, pages 150–166. Springer, 2015.

[18] Hubert Ritzdorf, Ghassan Karame, Claudio Soriente, and Srdjan Čapkun. On information leakage in deduplicated storage systems. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop*, pages 61–72. ACM, 2016.

[19] Mark W Storer, Kevin Greenan, Darrell DE Long, and Ethan L Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10. ACM, 2008.

[20] Chuanshuai Yu, Chengwei Zhang, Yiping Mao, and Fulu Li. Leap-based content defined chunking—theory and implementation. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–12. IEEE, 2015.