Aalto-university
School of Science

Markus Suonto

Deduplication system for user encrypted data

Master's thesis

17.03.2015

Instructors: Prof. Refik Molva, M.Sc. Pasquale Puzio
Supervisor: Prof. Jukka Nurminen

1. Summary

Author: Markus Suonto

Title: Deduplication system for user encrypted data

Track: Communication System Security

Date: 17.03.2015 Pages: 48

Supervisor: Prof. Jukka Nurminen

Instructors: Prof. Refik Molva, M.Sc. Pasquale Puzio

Language: English

This thesis explains the process of constructing a deduplication system of encrypted data. The goal of the project was to build a system prototype for secludIT. Primary attributes for the system were to provide substantial storage space saving by cross-user deduplication while preserving security. Secondary goals for the system were maximum throughput, minimum latency and support for file sharing between users.

The thesis explains different approaches to performing deduplication with their advantages and disadvantages. The difference between target- and source-based deduplication is explained, along with the distinction between file- and block level deduplication. The thesis further explains the effect of utilizing fixed- or variable-sized chunking when performing block-level deduplication. Block-level deduplication with variable-size chunking proves the most efficient and is selected as a deduplication method for the prototype.

State of the art layouts are analyzed. Security feature conflicts with file sharing are explained. Three levels of security are defined for the prototype to meet the requirements of different customers.

To evaluate primary attributes of the prototype, achieved space savings are measured along with the overhead caused by metadata. To address secondary goals, throughput and latency benchmarks are presented.

The final prototype is observed to meet the goals, except for throughput and file sharing support. The throughput bottleneck is identified. One security improvement is proposed to the state of art.

Keywords: deduplication, convergent encryption, file chunking, data security

2. Résumé

Ce document décrit le processus de réalisation d'un système de déduplication de données chiffrées. L'objectif du projet est de faire un prototype pour SecludIT. La caractéristique principale de ce système doit être de permettre d'importantes économies de stockage grâce à la déduplication entre utilisateurs, sans compromettre la confidentialité. Deuxièmement, comme objectifs secondaires, le système devra être fiable, capable de traiter de grands volumes de données avec une latence minimale et permettre le partage de fichiers entre utilisateurs.

Ce document discute différentes approches pour régler le problème de la déduplication, ainsi que leurs avantages et inconvénients. La différence entre la déduplication basée sur la cible et celle basée sur la source sera expliquée, ainsi que la distinction entre déduplication au niveau du fichier ou du bloc. On expliquera aussi l'effet de l'utilisation du découpage de fichier en blocs de taille fixe ou variable. La déduplication au niveau du bloc combinée au découpage à taille fixe est la stratégie la plus efficace et a donc été sélectionnée comme méthode pour le prototype.

Les différentes solutions existantes ont été analysées. Les conflits concernant les différentes fonctionnalités de sécurité seront expliqués. Trois niveaux de sécurité seront définis afin de satisfaire les besoins des différents clients.

Afin d'évaluer les caractéristiques principales du prototype, on a mesuré les économies de stockage ainsi que le surplus de stockage occasionné par le gestionnaire de métadonnées. Afin de répondre aux objectifs secondaires, le débit et la latence seront examinés.

Le prototype réalisé atteint les objectifs déclarés, sauf pour le débit et le support du partage de fichiers. La raison du goulet d'étranglement a été identifiée. Une amélioration de sécurité est proposée.

3. Acknowledgements

Thanks to Pasquale Puzio for his advice and instructions and to all the people who motivated me or participated in proofreading this thesis.

Nice, France, 24.01.2015 Markus Suonto

4. Table of contents

5. Terms and Symbols

Terms

Metadata	All information that is stored in order to be able to retrieve actual data. May include, for example, encryption keys, ownership references and data locations in the cloud storage
Deduplication group	A group of users, for which cross-user data deduplication is possible
Deduplicability	The ability to perform deduplication
FIFO	(First In First Out) Buffering protocol where packet that arrives first always leaves first, no priorization or fairness policies
API	Application programming interface

Symbols

B	Byte, 8 bits
KB	kilobyte, 1024 bytes
MB	megabyte, 1024 kilobytes, 1048576 bytes
Mbps	megabits per second, 1048576 bits / second

1. Introduction

In the past decade, cloud storage has become a widely accepted and utilized manner of storing one's data. To save space, smart storage providers, like Google and Dropbox, utilize data deduplication [1]. In other words, they exclude all duplicate files or data blocks from being stored. For example, if you upload a very popular file into your Dropbox, it is very likely that some other user of the cloud has already stored it, and they deduplicate your upload. This means that they will not actually store your entire file, but just a reference indicating that you also have the same file than the other user. Thus, they save substantial space. However, for some parties, the confidentiality of their data is a priority, and they encrypt their data using some cryptographically strong encryption, like AES. This makes it impossible, or at least very inefficient for the storage provider to perform deduplication, as two identical files encrypted with non-identical keys produce two different ciphertexts. Therefore, the cloud does not know that the original files were identical, thus losing the benefits of data deduplication.

Fortunately, some deterministic encryption methods, like convergent encryption [8], preserve deduplicability while providing data confidentiality. Hence, they introduce a possibility of building a system that would provide both privacy and deduplicability. On the other hand, the security of convergent encryption is not sufficient by itself. Indeed, several severe attacks have been identified against it, and are presented in chapter 5.

The objective of the project was to build a deduplication system prototype for secludIT. Primary attributes for the system were to provide substantial storage space saving by cross-user deduplication while preserving security. Secondary goals for the system were maximum throughput, minimum latency and support for file sharing between users.

Different approaches to performing deduplication are explained along with their advantages and disadvantages. The difference between target- and source-based deduplication is explained, followed by the distinction between file- and block level deduplication. Furthermore, the thesis explains the effect of utilizing fixed- or variable-size chunking when performing block-level deduplication. Variable-size chunking proves the most efficient and is selected as a deduplication method for the prototype.

In order to select optimal set of security features for the prototype, state of the art layouts are analyzed. Security feature conflicts with file sharing are explained. Three levels of security are constructed to meet the requirements of different customers.

To evaluate the primary attributes of the prototype, achieved space savings are measured along with the overhead caused by metadata. To address secondary goals, throughput and latency benchmarks are presented.

In terms of scientific value, the thesis examines the State of the Art for block-level deduplication security features and presents three combinations that provide a certain level of security with minimized key management costs and minimum restrictions on file sharing. Furthermore, the block-level deduplication efficiency is verified with a large set of real user email data. Also, the metadata overhead of a block-level deduplication system is measured in practice and throughput bottleneck is identified.

2. Deduplication

The objective of deduplication is to store each unique file or data block in the cloud only once. The fundamental idea is simple: whenever the cloud receives data, it uses an identifier, for example a hash, to determine whether the data already exists in the cloud. If so, the data itself will not be stored, but instead a reference indicating the ownership of the user to the previously stored data. Deduplication can be divided by user base into single-user and cross-user deduplication, depending on the service provider's ability to deduplicate files that were uploaded by different users. Another division can be made to source-based and target-based deduplication. In source-based, duplicates are identified and discarded by the client. In target-based, the operation occurs at the storage provider.

2.1. Cross-user deduplication

The requirement for cross-user deduplication is that the same data, encrypted and sent by two different users, arrives to the cloud as identical. Cross-user deduplicability can be achieved only if the users do not use any secret keys in encryption, or they share the same keys. A group of users, for which cross-user deduplication is possible, is in this thesis denoted by deduplication group. In practice, the group would in most cases consist of employees of a company. If the size of the deduplication group is one, the term to be used is single-user deduplication. In this case the user encrypts his or her files with a secret key that no one else knows, and deduplication is only possible if the files contain some internal redundancies.

2.2. Source-based deduplication

The objective of source-based deduplication is to reduce critical network traffic. It is important to understand the division between critical and non-critical traffic, so let us consider an example. Imagine a scenario, where the deduplication group consists of users of a company, and the target storage is Amazon object storage [27]. In this case, the traffic in the company's internal network is non-critical, whereas the traffic between the company and amazon object storage is critical. To minimize external bandwidth usage, the company might decide to perform data deduplication at their own premises, as the last operation before it is sent to the cloud. This solution would be further supported by the fact that the deduplication group is entirely inside their internal network. The requirement to perform source-based deduplication is that metadata must be managed at the source. However, the security of a deduplication system depends on the component layout. Placing all critical components in the same network introduces a potential security issue, as an attacker, or an internal curious network administrator, is sometimes able to compromise the network. Thus, bad component placing can be interpreted as a violation of the principle of having no single point of failure.

As a conclusion, source-based deduplication reduces the external network traffic, but introduces a potential security weakness that should be taken into consideration.

2.3. Target-based deduplication

If the bandwidth between the source and the target is not a priority, it allows more secure system layouts or larger deduplication groups. As an example of a larger deduplication group, consider Dropbox. As individual users normally do not encrypt their data before storing them, Dropbox is able to, and does, perform deduplication across the entire user base [9]. However, deduplicating across all files in the system, or all blocks of the files, requires the deduplication component to keep a table of all the files or blocks in memory. To reduce memory usage, the service provider can utilize statistical information, and apply deduplication only to the most frequent files or blocks. Such optimization is complex, and outside the scope of this thesis.

2.4. Data chunking

Deduplication efficiency in storage space savings can be increased by data chunking. It is applied by dividing files into chunks and assigning an

identifier for each chunk. The system then discards all incoming chunks whose identifiers match with any of the ones already in the storage. For assigning the identifiers, two requirements exist: for each unique chunk, the identifier must be unique, and for any amount of identical chunks, the identifiers must be identical. The event of two non-identical chunks generating identical identifiers is called a collision. Such event would cause the deduplication system to incorrectly discard the second chunk as a duplicate. As a final consequence to a client using the storage system it would mean that when he or she tries to download the file, a part of the file would actually be from another file. Hence, the file would be corrupt. On the other hand, if two or more identical chunks generated non-identical identifiers, it would result into storing all of them, thus violating the principle of storing every unique chunk only once. The identifiers can be properly generated, for example, by using a collision resistant cryptographic hash function, like SHA256.

The conclusion of *A study of practical Deduplication* [4] indicates that in practice chunked deduplication is always at least as efficient as file-level deduplication, but results in higher complexity and lower performance. The selection of the chunking method greatly affects the efficiency of deduplication. Chunks, or as sometimes called, blocks, can be of fixed or variable size.

2.4.1. Fixed-size chunking

In fixed size chunking, files are divided into pieces of fixed size. The main parameter that affects the efficiency is the chunk size. The more the size increases, the less likely it becomes to find an identical chunk in a non-identical file. On the other hand, the smaller the chunks are, the more overhead per data is generated, as every chunk generates the same amount of metadata. The efficiency impact of chunk size is examined in greater depth in chapter 16, Performance. Picture 1 demonstrates deduplication of two partially similar files that were chunked into fixed-size chunks.

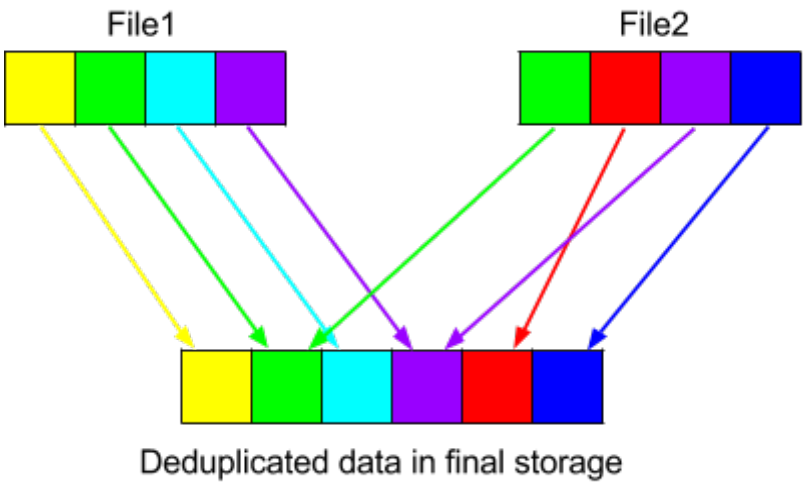

Figure 1: Deduplication of fixed-sized chunks

The main weakness of fixed-size chunking is that small differences, especially in the beginning of files, may lead to very bad results. For example, if a very small piece of data is inserted into one of two identical files, it will cause an offset for the division points of the files, possibly resulting in zero deduplicability.

2.4.2. Variable-size chunking

Variable-size chunking addresses the offset problem of fixed-size chunking by dividing files into pieces by a property of the block contents instead of a specific size. The property might be for example a checksum, or a special rule, like ending a block on every dot encountered. Picture 2 presents two files that can be partially deduplicated if they are divided into variable-size chunks. For clarity, identical blocks are identified with the identical colors.

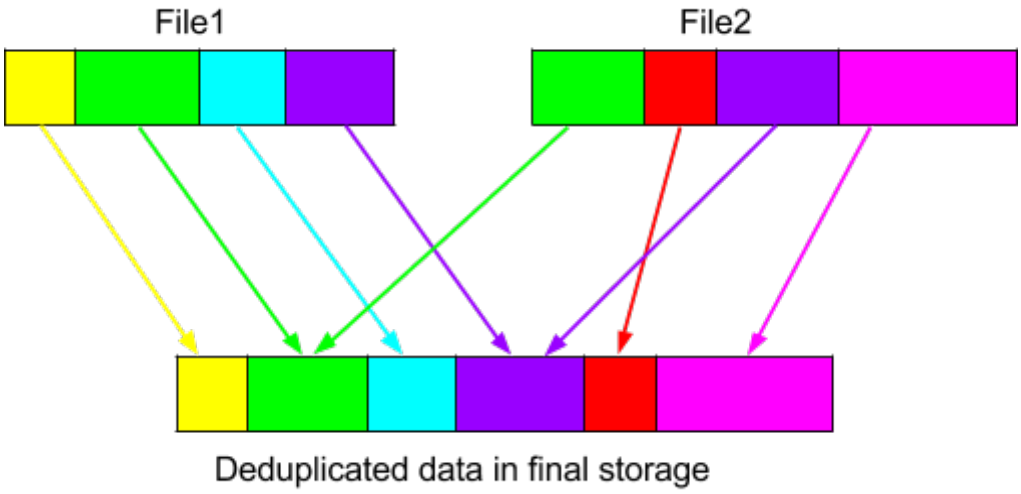

Figure 2: Deduplication of variable-sized chunks

A good candidate for division property is rabin-fingerprint [6]. A file can be divided by rabin-fingerprints by sliding a window over the file, and ending a chunk when a fingerprint of certain type is encountered. For example, if the window size is w, then the algorithm starts by computing a fingerprint for bytes from 0 to w, and then from 1 to w+1. In this case, the rule might be that the block will be ended if the first 10 bits of the fingerprint are zero. On the other hand, the same functionality could also be achieved by computing a cryptographic hash over each w bits. However, rabin-fingerprint provides better efficiency because it allows partial reuse of previous window's result when computing the new fingerprint.

2.5. Inefficiency of deduplicating symmetrically encrypted data

The number of different files in storage is notated by n and the number of different files in the world by N. Now, if none of the files are encrypted, the probability of a new file already being in the system is in the magnitude of n/N. This does not consider file frequencies or other advanced parameters. If all the files in the system are encrypted with a random 160-bit key, the probability for a new file being in the system is in the order of $n/(N^2 * 2^{160})$, which is negligible. As a conclusion, the probability of deduplicating symmetrically encrypted files, even in a large file base, is very low. In the deduplication perspective this means that two users that symmetrically encrypt their files with different keys, are never considered to be in the same deduplication group.

3. Economical motivation for deduplication products

In order to optimize profits, cloud storage providers charge their clients by the size of their data before they are deduplicated. Naturally, this fact serves as an incentive for the companies and third parties to perform deduplication themselves, and provide the cloud with only non-redundant data. By doing this, they reduce their storage costs, and deny the storage provider the deduplication ability. In consideration of this and the fact that most companies value data confidentiality, it is safe to assume that a market exists for products and services performing encrypted data deduplication.

4. Convergent encryption

To allow cross-user deduplication of encrypted data, the encryption operation must be deterministic. Convergent encryption, presented in *Reclaiming space from duplicate files in a serverless distributed file system* [8], approaches this by using the hash of the plain data as the encryption key. This way, identical plaintexts produce identical ciphertexts irrespective of the user. As a result, deduplicability is preserved for identical files that are encrypted by different users.

The convergent encryption key is determined by hashing the data:

$$K = (H(data)) \tag{1}$$

The final ciphertext is then acquired by symmetrically encrypting the data using the newly acquired key:

$$ct = E_K(data) \tag{2}$$

It is important to select a hash function that is both weakly and strongly collision resistant. In other words, for the hash function it has to be very hard to either find two input values that result to the same output or to find an input that results to a given output value. For example, SHA1, that produces a 160-bit hash, is a collision resistant hash function. For symmetric encryption, AES is the standard choice. AES mode needs to be selected depending on the situation. For example if the files are chunked to blocks, CTR is a good choice, as it allows parallel encryption of the blocks. AES requires the data to be padded into a multiple of the key length.

5. Attacks against deduplication systems

5.1. Confirmation of file (COF)

The confirmation of file attack (COF) was presented first in *Side channels in cloud services, the case of deduplication in cloud* storage [3]. The objective of COF is to acquire information about other people's files in the system.

5.1.1. Online COF by a client

In practice, a standard attacker is a client that is curious about other people's files in the storage and performs queries to extract information from the system. This is exactly the version of the attack was presented in *Side channels in cloud services, the case of deduplication in cloud storage*. To perform the attack, the adversary guesses a file, uploads it to the system and checks if it is deduplicated. Detection of deduplication is possible if the system is a source-based or a hybrid between source- and target-based system. In these systems, deduplication can be detected by timing the upload requests or by monitoring network traffic after the deduplication point. The attack is not very efficient, as it is an online attack. In other words, each guess requires a request to be made. The attack logic is further demonstrated in Figure 3.

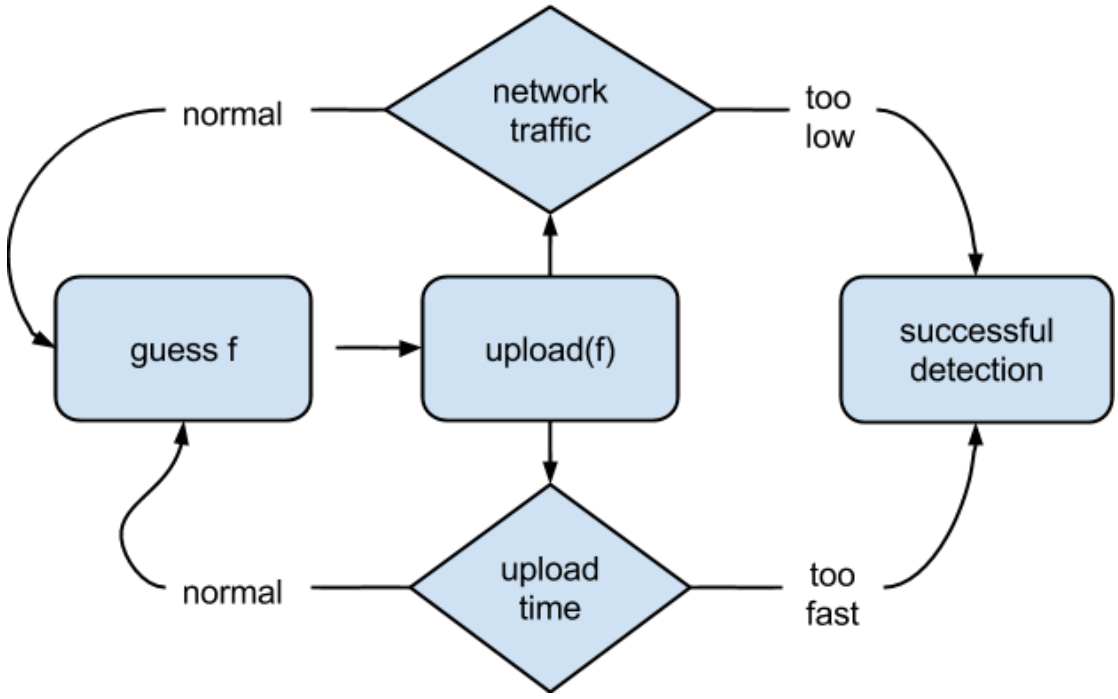

Figure 3: Client performed COF attack by measuring network traffic or response time

The attack was described against a system storing plain text files, but as also mentioned later in the thesis, convergent encryption does not provide protection against it. When convergent encryption is utilized in the system,

the adversary only needs to apply it to his or her guess and send the resulting ciphertext instead of the plaintext. In order to be successful, the adversary needs to succeed in guessing the contents of the target file. Hence, all files that are rare enough or have enough entropy are protected. On the other hand, if the adversary is unable to guess the file, then it does not matter whether convergent encryption was applied or not. As a conclusion, convergent encryption brings no protection against COF attack. Figure 4 demonstrates client COF attack and deduplication detection in a system that applies convergent encryption.

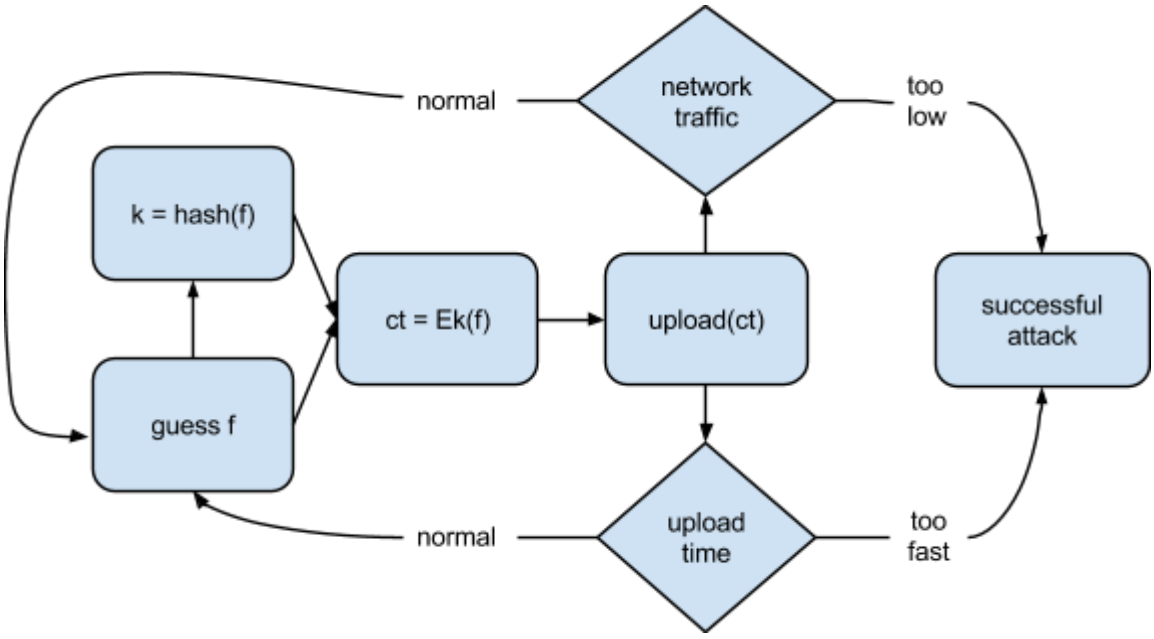

Figure 4: Client COF attack in a system utilizing convergent encryption

5.1.2. Server side offline COF against convergent encryption

The value of convergent encryption is that the plain data cannot be accessed after the data leaves the client. However the cloud storage and the component storing the metadata are powerful adversaries. For this scenario, assume that the convergently encrypted data is stored at a cloud storage provider, and the convergent encryption keys are stored at a separate component called metadata manager. In this scenario, both components are able to perform powerful offline dictionary COF attacks. To perform an attack against the keys stored at the metadata manager, the adversary generates a dictionary of files, their keys and resulting ciphertexts. Then all the keys in the system are compared against the dictionary. On a match, the plaintext is revealed. Figure

5 demonstrates the discovery of one plaintext and key by an offline dictionary attack against the keys in the metadata storage.

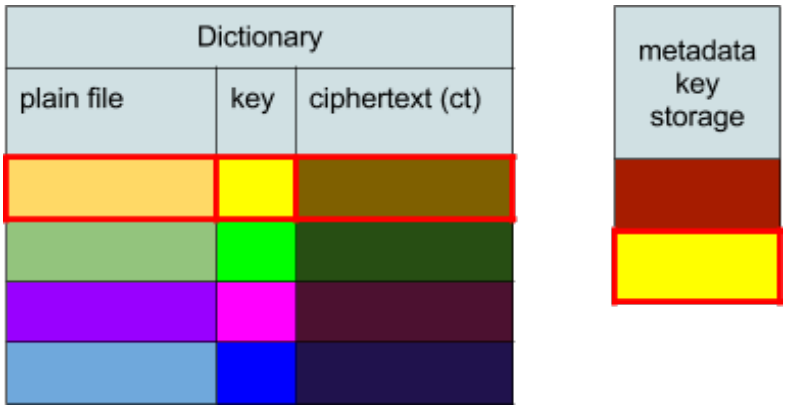

Figure 5: dictionary attack against convergent encryption keys

To perform a similar attack against the ciphertexts at the cloud, the curious cloud service provider can generate a similar dictionary and compare all the stored ciphertexts against it. The attack is demonstrated in Figure 6.

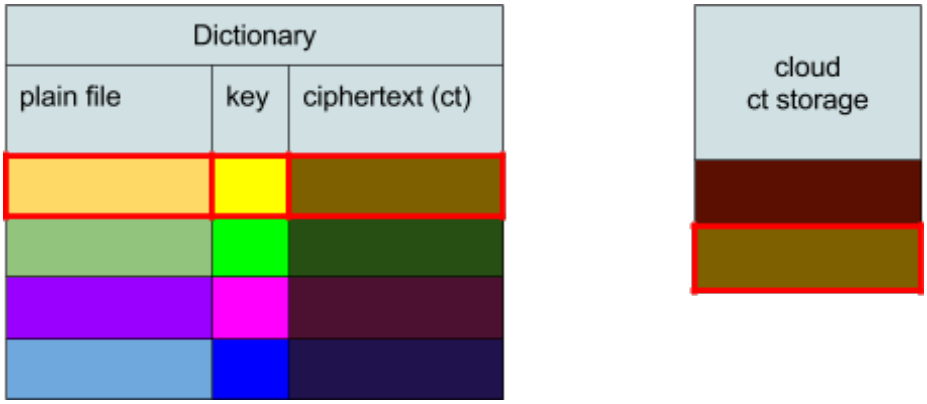

Figure 6: offline dictionary attack against convergently encrypted files

5.2. Confirmation of block

Confirmation of block can be utilized against systems that perform block level deduplication. The only difference to confirmation of file (COF) is that instead of a file, the attacker guesses only a block.

5.3. Learn remaining information

Learn remaining information (LRI) is possible if an adversary knows the template of a common file that is common in the system and contains for example a pin code or a password. In this case he or she can guess a password, add it to the template and utilize COF to check if a file with that password can be found in the system. The attack is especially dangerous if the template contains both an identifier of the user and the secret. For example, imagine a situation, where the attacker knows that a millionaire named Bob backs up all of his emails in the system, and that his new online banking credentials were sent to him by email, such as the one in Figure 7.

```
Subject: pin code
From: support@securebank.com
To: bob@hotmail.com

Welcome to SecureBank, Bob!

Please find herewith your
new SB-online credentials:

username:BOB
pin code:5120

Best regards,

SecureBank customer support
```

Figure 7: Example template containing confidential information

In this scenario, the adversary is able to target the LRI attack by always inserting BOB as a username and iterating over the possible pin code values. Hence, he or she would acquire Bob's secret by performing at most 10 000 COF attacks, each with the username BOB and a 4-digit pin code.

As a real world example, a French bank Banque Populaire initializes each user online credentials with a random identifier and a date-based default password. Users are forced to change their password upon first login, but for some users it might take days, weeks or they might not use the online banking at all. For security, the accounts are closed after 30 days if the user has not logged in. The credentials are received by mail, but for curiosity let us imagine that they were sent by email and ended up in a deduplication system as backups, like in the previous scenario. The user identifier is of the format *IXXXXXXX*, where *i* is a constant character and *X* presents a random number. The password is of the format *BPRRYYYY*, where *B* and *P* are for Banque Populaire, *RR* for the region in France and *YYYY* is the account opening year.

In Figure 8, an example of an account opened in Côte d'Azur (CA) is presented.

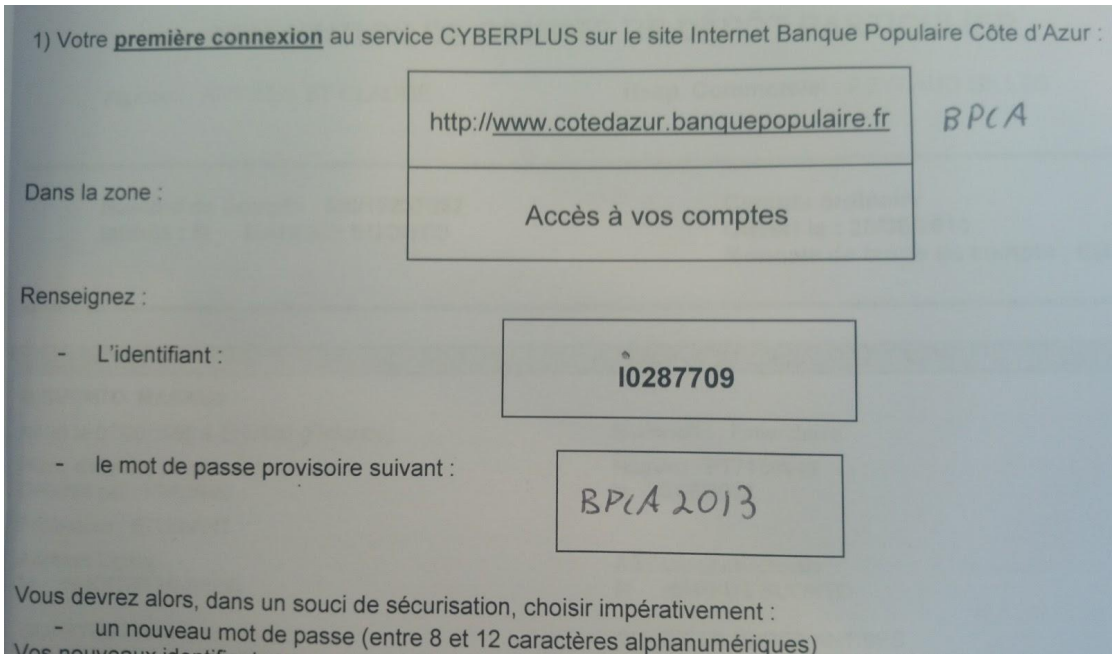

Figure 8: Banque Populaire account initial credentials after opening in 2013

In this scenario, an adversary could target the LRI against all of the users that he or she knows to have opened their account within the last 30 days and have not logged in to change their password. The difference to the previous scenario is that in this case the adversary would target the attack by setting the password to a specific value and iterating over the 10 million possible user identifiers. Furthermore, if the identifier is not randomly generated, for example if it was a counter, the attacker might expose the entire system just by executing one successful attack.

The severity of targeted LRI is increased by the fact that even if the banks or other systems had good mechanisms to detect bruteforcing attempts, it would not help, as the attack is executed in another system, thus being completely transparent to the bank system. Even in the deduplication system, such operation might be hard to detect, for example if the system has a developer API. For such systems frequent uploads and deletions of files by the same user are completely normal. Another difficulty in detecting the attack is that the secure systems are designed in such way that they do not have the ability to reveal what they are storing. Hence, it is impossible for the system to detect whether a user is iterating over values on successive data uploads.

The prerequisite of performing an LRI attack is the ability to perform COF attacks. Furthermore, the viability of this attack also depends on the difficulty of guessing the remaining information. Thus, if the remaining information is for example a very strong password or a hash, this attack is no longer viable.

As a slight advantage to the adversary, it can be noted that if the file is frequent in the system, for example if 100 000 occurrences of the template exist in the system, the attack is successful if the guessed password matches any of those in the files. Therefore, the efficiency of LRI-attack increases linearly to the frequency of the template in the system. However, this does not in any case make the attack viable against very strong passwords, like 32-byte hashes, as the efficiency decreases exponentially as the password strength increases.

5.4. Cover channel attack

If an adversary has managed to install malicious software to a client, the software may utilize COF attacks for communication. For instance, the malicious software, the bot, can be instructed to upload a file with certain content upon successful installation on a host. As a result, the master is able to utilize COF attacks to scan the system for all the pre-determined files. The former method basically transfers one bit of information. However, with multiple pre-determined files, messages of any length can be exchanged in the same manner [3].

6. Enforcing the system against attacks

Against curious clients, the only way to prevent COF and COB is by hardening the system against monitoring. If the attacker is unable to determine whether deduplication occurred or not, he or she is unable to perform COF or COB.

6.1. Preventing upload timing at client

In source-based systems, the most straightforward way for a client to determine whether deduplication occurred or not is to time the upload operation. If the upload operation is must faster than it should, then the client knows that the data was deduplicated. To harden the system against monitoring, the deduplication must be done transparently to the clients. Hence, no matter if the file is actually deduplicated, it must seem to the client like the upload occurred. If a client is unable to tell whether deduplication occurs to their current upload, the system reveals no information to a potential adversary, thus denying the opportunity to perform COF, COB or any other attack that depends on detecting deduplication.

6.2. Hardening against network traffic monitoring

A less straightforward way for a client to detect deduplication is to monitor network traffic. Hardening against it, is done at the expense of network load, as it requires the traffic to enter the network. Duplicate packets are dropped when they reach the deduplication component of the system. To position this component properly, it is important to determine who the potential attacker is, and what he or she is able to monitor. In order to reduce network traffic, the component should be placed as close to the client as possible. To harden against monitoring, it should be in a position after the client is unable to monitor the network. For example, if the metadata manager is located at the enterprise premises, and the attacker is able to monitor the traffic between the enterprise and the cloud storage, then he or she is also able to tell if the packets were dropped at the metadata manager. But on the other hand, hardening the system further would force it to be completely target-based, which means that no network savings are gained. And even then, it is possible that the adversary would have access to the deduplication information. Such a powerful adversary might be, for example, a curious metadata management service provider that has also access to a client of the system or some law authority with a warrant on the service provider. Considering this, hardening the system against monitoring should always be done by the basis of what kind of attacker should the restriction apply to, and how much it will cost in terms of network traffic.

7. State of the art

In this chapter, recent ideas for a secure system layout are presented.

7.1. ClouDedup

ClouDedup [5] is a high security block level deduplication system. It consists of four separate components: client, server, metadata manager and cloud storage. It has been designed in such a way, that none of the components can alone expose the system.

7.1.1. Client

In the ClouDedup system, each client stores a symmetric and an asymmetric encryption key. As the first step of uploading a file, it is divided into variable-

size blocks using rabin fingerprint algorithm. Then convergent encryption is applied to the blocks. The notation for the n^{th} block is as follows: block data B_n, ciphertext ct_n, key K_n.

$$K_n = (H(B_n)) \tag{3}$$

$$ct_n = (E_{K_n}(B_n)) \tag{4}$$

As a result, two sets are generated for each file: keys, and data. Now, if the client was to store all the keys locally, it would eventually become a burden, as the number of keys can become very high. Therefore, the keys are chained and only the first key is stored at the client. The rest of the keys are sent with the data through the gateway to be finally stored at the metadata manager. In the key chaining all the keys except the first one are symmetrically encrypted using the previous key as an encryption key. The resulting key ciphertexts are noted by *kct*.

$$K_n = H(B_n)) \tag{5}$$

$$kct_n = E_{K_{n-1}}(K_n)) \tag{6}$$

Correspondingly, when retrieving the blocks, the user decrypts the key ciphertexts *kct*, starting from the second block.

$$K_n = D_{K_{n-1}}(kct_1)) \tag{7}$$

After decrypting the keys, the user is able to decrypt the data with them:

$$B_n = K_n(ct_n)) \tag{8}$$

The advantage of performing block key chaining is that if an adversary desires to perform COB, he or she is forced to guess the first block to gain access to any of the following blocks. If he or she manages to guess the first block, then the second block becomes vulnerable for guessing. As a result, the attacker is required to guess the entire file instead of a block. To conclude, the key chaining acts as a protection against COB, but still allows COF.

It is not desirable that the metadata manager would be able to attempt COF on the file keys, so they are also symmetrically encrypted, using the client's secret key K_s. Finally, the form of the chained keys after the first one is:

$$ct_n = E_{K_s}(E_{K_{n-1}}(B_n)) \tag{9}$$

As a result, the metadata manager is unable to perform COF attacks against stored keys, except if a user's secret key has been leaked. Even if the keys are secured, the metadata manager can download the data blocks from the cloud storage and try COB attacks against them. It can even be done offline with certain preparations. If the metadata manager downloads each block, stores a hash of the block and drops the data, it can compress the encrypted data into an offline table. Then it can run COB with ciphertexts by comparing the hash of each ciphertext to the ones in the table.

To summarize: if the file is guessable, the security of the keys relies entirely on the symmetric encryption with the users secret key. If not, then key chaining protects the blocks against COB in case the secret key is leaked to the adversary. However, in practice, if an adversary has access to the users secret key, then it is very likely that he or she also has access to the first key of the chain, as they are both stored in the users computer. Considering this, one might argue that the practical security value of key chaining, in addition to symmetric secret key encryption, is low. The metadata manager is extremely powerful as an adversary, and can run offline COB against the data even if the keys are secured.

As a final step before sending the keys and the data to the gateway, the client appends each data block with a signature. The signatures are generated using the asymmetric key of the client. The purpose of the signatures is to tie the data to the owner. This way, the gateway, that possesses all the clients' public keys, has the ability to verify any data request by the public key of the requester.

7.1.2. Server

The most important invention in the ClouDedup system was the addition of a component called server. The server is basically a gateway proxy that performs additional symmetric encryption of data blocks for the entire deduplication group. To prohibit the metadata manager of performing COF or COB attacks against data, the gateway applies additional symmetric encryption to each data block that passes through. Additionally, the gateway enforces access control by checking the signature of every requested data block to verify that the user is an owner of the block. As a result, access control becomes databound.

7.1.3. Metadata manager

The metadata manager (MM) is the component responsible for storing the metadata of all the blocks in the system. The metadata consists of the data info

and the key info. Data info consists of entries about who owns a file and which blocks are part of it. Additionally, the MM also handles the actual deduplication and stores the deduplicated blocks in the cloud. The procedure is simple, if a block already exists, it is dropped and a counter is increased for the block. In terms of security, an optional additional symmetric encryption is proposed to be applied to the data that is uploaded to the cloud. The cost of this operation would be low, and it would guarantee that even if the symmetric encryption key of the gateway key is leaked, the cloud cannot perform offline dictionary attacks against the stored encrypted data. That is, of course, unless also the metadata manager symmetric encryption key would be leaked along with the gateway key.

7.2. DupLESS

DupLESS [1] system implements a protocol between clients and a key server. The system allows file-level deduplication while providing high level of security. Furthermore, it is almost transparent in terms of performance. The introduced protocol allows the clients to retrieve deterministic symmetric encryption keys for files while revealing no information about the file to the key server. With these keys the clients are able to directly access existing storage APIs while cross-user deduplicability is preserved.

7.3. Differences between ClouDedup and DupLESS

The main difference is that ClouDedup performs deduplication in block-level, and DupLESS in file-level. DupLESS does not introduce a component to manage the users' metadata, because for file-level deduplication the amount of metadata is very low. ClouDedup introduces the metadata manager to deal with the issue.

Considering the component server in ClouDedup and the key server in DupLESS, the functionality differs only slightly. In terms of performance, in DupLESS the query is made before uploading the actual data, as opposed to ClouDedup's transparent encryption proxy solution. As a result, the DupLESS imposes a slight latency penalty to file upload. In terms of security, the solutions are equal, except that in DupLESS the keyserver has no information about the actual files, as opposed to full exposure in ClouDedup, where all the data pass through the gateway. In ClouDedup, the gateway is considered to be partially trusted.

To address the question whether or not the DupLESS architecture could be adapted into a block-level deduplication system, it is necessary to consider the key server load. In the current layout, the key server receives a request per file, but if converted into a block-level system, it would receive a request per block. As a result, the load would multiply by the average number of blocks in a file. However, the system performance was measured to be decent for 3k files per second, so indeed it is possible that a conversion would work decently. However, in any case, the key server operation is much heavier than the simple encryption of the ClouDedup gateway. To conclude, the DupLESS system has a slight advantage in revealing no data to the server component, but ClouDedup architecture provides better performance.

8. Security feature analysis

The security of a convergent encryption based deduplication system can be measured by the difficulty of accessing plain data that is stored in the system. As shown by pictures 4 and 5, information can be acquired by breaking either the keys or the data. As a result of a security breach, two dangerous levels of access can be gained. If the adversary gains access to convergently encrypted data or plain convergent encryption keys, he or she has access to plain data through COF-based attacks. If the adversary acquires plain data or gains access to both plain convergent encryption keys and convergently encrypted data, he or she has plain data access. Besides encryption, higher level of security can also be achieved by improving access control. This chapter focuses on features that improve data confidentiality, thus protecting the data against attacks. The chapter 12, Prototype access control, focuses on features and options for assuring that every access to the data in the system is authorized properly.

8.1. Applying a secret in convergent encryption

If a secret is appended to the plain text before hashing, convergent encryption indeed becomes attack resistant. The strength added is equal to the strength of the secret as a key. In other words, every 8-bit character of the secret makes the attacks 2^8 (256) times more expensive. Adding a long key, let's say 16 bytes or more, is very secure, but if done, then convergent encryption is not needed any more, as the key would work directly as a symmetric encryption key. After all, the idea of convergent encryption is to generate the encryption key from the plain text. The ability to deduplicate files/blocks is only

preserved if the secret is the same for all clients involved in the deduplication group. So if an adversary can get access to the secret of one of the clients he also obtains the secret of the entire deduplication group. Thus, using secret as a reinforcement for deduplication systems, makes every client in the group a single point of failure in the group, which can be considered as a partial violation of cryptographic "no single point of failure" design principle. Also, each user can perform attacks on other users files.

8.2. Additional Symmetric Encryption

Additional encryption can be applied in two ways: by directly encrypting data after applying convergent encryption, like in ClouDedup, or by deterministically generating the final encryption key from the convergent encryption key before encrypting the data, like in DupLESS. If additional symmetric encryption is applied after convergent encryption, data cannot be recognized anymore with COF attack, thus preventing offline dictionary attacks of later components. The advantage of this method is that the additional encryption key does not have to be stored in the client. In ClouDedup the key is stored in the gateway, and in DupLESS it is stored in the key server. Storing the key in an additional component ensures that clients are no longer single points of failure for the system. However, in this case the component storing the key for additional encryption remains a single point of failure. One must remember though, that the security of deduplication systems depend on the security of both the keys and the data. Therefore, if additional encryption is applied to the data, then the keys should also be protected if they are to be stored among the metadata. Otherwise the metadata storage component can still perform offline COF against the keys.

8.3. Block key Chaining

The key chaining, as described in ClouDedup, protects keys against COB. Storing the first key at the client allows encryption of every successive key with the previous key, starting by encrypting the second key with the first one. As a result, in order to perform a dictionary attack against any block, the metadata manager is required to guess all previous blocks. In other words, the difficulty for a metadata manager dictionary attack against chained keys is somewhere between COF and COB. Key chaining imposes a key management cost of one key per file at the client.

8.4. Secret key encryption of block keys

8.4.1. Secret key per user

As an alternative to key chaining, the secret key encryption of keys at the client offers higher cryptographic strength and lower key management cost than key chaining. By encrypting each block key with the users secret key, all the keys are secured unless the secret key is leaked. Also, the key management cost is only one 32-byte secret key at the client. The feature was introduced in ClouDedup.

8.4.2. Secret key per file

To support file sharing, secret key per file encryption is proposed. In order to share a file when encrypting the block keys with a secret key, the collaborators have to share the same secret key. However, sharing a personal secret key between a group of users means that unless restricted with access control mechanisms, they share all of their files with each other, as if they were the same user. Therefore, it is wiser to generate an additional secret key just for the shared file, and replicate that key to all collaborators, instead of the personal secret key.

8.5. Post-MM (metadata manager) encryption

An additional symmetric encryption, performed by the metadata manager, as described in ClouDedup, prevents the gateway from exposing the system to COB attacks by collaborating with the cloud. In practice, the metadata manager encrypts the data blocks as a last operation before sending them into the cloud. This way the cloud is unable to run dictionary COB attacks against the data even if it possesses the gateway symmetric encryption key. The operation imposes a key management cost of one 32-byte key at the metadata manager.

8.6. Security feature conclusions

Applying a secret during convergent encryption at the client removes cross-user deduplicability unless the secret is the same among all clients in the deduplication group. Sharing the secret is a security risk, as each client becomes a single point of failure for the system.

Block key chaining, as proposed in ClouDedup, is an equivalent operation to secret key per file encryption of block keys in the sense that both operations provide security to the block keys by retaining a secret at the client. However, the secret key encryption is more secure, as the key chaining leaves the possibility to guess the first block of the chain and the corresponding key.

9. Security feature selection for the prototype

In this chapter, the security features are selected for the prototype, and their added security is analyzed by component. In other words, the cost of applying the features is compared to the value they provide. Furthermore, the value is analyzed by component.

Table 1 presents the possible security features for the prototype with their added security value as cryptographic strength, along with their key management costs. For every feature, the performance cost is considered acceptable, as they consist of only hashing and symmetric encryption. Therefore, the performance cost is not listed in the table. The cryptographic strength at each component is measured as the asset that is required or the attack that has to be performed in order to access the plain data at that component. For an example, the first row of Table 1 shows what level of security convergent encryption alone provides at each component. At the gateway column of row 1, the value - for keys means that the gateway has plain access to the keys, and the value COB for data means that the data is convergently encrypted, but vulnerable to confirmation of block attack. As an example of an asset, the metadata manager (MM) column of row 2 shows that when additional encryption (ae) of data is performed at the gateway, the symmetric encryption key of the gateway is required at the metadata manager in order to access plain data. In Table 1, each row presents the value added by that specific security feature alone. Combinations are later presented as security levels of the prototype.

Applying a secret in each client during convergent encryption was discarded as an option because it is similar to gateway performed additional encryption, but makes all of the clients holders of the shared secret, making it easier for adversaries to acquire than from a single external component.

Key chaining was also discarded, because generating a secret key per file and applying symmetric encryption to the block keys imposes lower or at most equal cost in terms of key management while providing better performance and stronger security.

Secret key per file encryption of block keys that was described in chapter 8.4.2 is also not present in the table, as the security provided is equal to secret key per user encryption. The difference between the two is that generating a secret key per file does not prevent file sharing, but requires management of multiple keys at the client as opposed to one personal key.

Block signing with a private key is excluded from this table because it is an access control mechanism and provides no data security.

- = plain access

S = personal secret key of the client

GW = gateway symmetric encryption key

MM = metadata manager symmetric encryption key

COB = convergently encrypted data, vulnerable to dictionary COF and COB

Feature	Cryptographic Strength					Key Management Cost		
	Gateway		MM		Cloud	Client	Gateway	MM
	keys	data	keys	data	data			
convergent encryption	-	COB	-	COB	COB	-	-	key per block
ae of data	-	-	-	GW	GW	-	one key	-
secret key encryption	S	-	S	-	-	one key	-	-
post-MM encryption	-	-	-	-	MM	-	-	one key

Table 1: Security features and their added cryptographic strength

From the possible features, three levels of security were constructed:

9.1. Minimum security level

Minimum security level is for normal users and businesses that do not store any critical information in the system and that do not want to deal with any unnecessary key management. For minimum security, the gateway symmetric encryption was considered sufficient. This way the data is secured against the potentially curious cloud storage provider unless the gateway is compromised.

9.2. Medium security level

Medium security level is for standard businesses that value security of their critical documents, and accept sufficient security without any extra layers and key management. For medium security, additional encryption of data and together with post-metadata manager encryption was selected. The improvement to the minimum level security is that in the event of gateway key being compromised, a potentially curious cloud storage provider does not achieve access to plain data. The cloud provider achieves plain data access only by acquiring both the gateway key and the metadata manager symmetric encryption key. Security against the metadata manager is the same as in minimum security. In other words, it has plain access to data if provided with the gateway key.

9.3. Maximum security level

Maximum security is for government, military or banking systems that require maximum security and are able to deal with the key management issues that it brings. The level includes convergent encryption, additional encryption of data, secret key encryption of keys at the client and post-mm additional data encryption. The secret key encryption of the block keys protects them against dictionary attacks by both the metadata manager and the cloud. The secret is considered to be a per user as opposed to per file, as the key management cost is lower, and maximum security customers are less likely to be interested in file sharing.

9.4. Security level summary

The security levels, added value by component and key management costs are presented in Table 2:

Minimum: no convergent encryption, only additional encryption of data

Medium: convergent encryption, additional encryption of data

Maximum: all medium, secret key encryption of block keys and post-MM data encryption

Security Level	Cryptographic Strength					Key Management Cost		
	Gateway		MM		Cloud	Client	Gateway	MM
	keys	data	keys	data	data			
Minimum	no keys	-	no keys	GW	GW	-	one key	-
Medium	no keys	-	no keys	GW	GW + MM	-	one key	one key
Maximum	S	COB	S	GW + COB	GW + MM + COB	one key + key per shared file	one key	one key + key per block

Table 2: Prototype security levels and their key management costs

10. File sharing in block-level deduplication system

The definition of file sharing in a deduplication system is that two or more users can read and modify a file unlimited times after only one of them has initially uploaded and shared the file. The file sharing ability is considered lost upon modification if any of the collaborators become unable to decrypt it.

Block key encryption and signature verification features impose restrictions on file sharing between users. This chapter first explains how file sharing works without any of the features and then analyzes the restrictions by feature and compares their severity. Finally, access revocation possibilities are analyzed.

10.1. File sharing without block key encryption

To understand how the file sharing works, imagine a situation where Alice wants to share a file with Bob. They both use the same deduplication storage system and belong to the same deduplication group. Alice starts by dividing the file with appropriate chunking algorithm and computing the block keys for the blocks, as presented in equation (3) and encrypting the data. Then, she uploads her keys and the ciphertexts into the system. Now, after retrieving the keys and the data from the system, Bob is able to decrypt the data with the keys, as shown in equation (8). When Bob contributes to the work and modifies the file, he computes new block keys, encrypts the modified file and uploads it into the system. Then, the next time Alice downloads the file and the keys, she can use the keys directly to decrypt the modified version of the file.

10.2. Secret key per user block key encryption

Now imagine a different scenario, where Alice and Bob generate personal secret keys K_A and K_B to encrypt their files. Like in the previous scenario Alice wants to share her file with Bob. Hence, Alice starts by applying convergent encryption to the blocks. Then she encrypts the keys with her secret key K_A:

$$kct_n = E_{K_A}(K_n)) \tag{10}$$

Then, Alice uploads the key ciphertexts (*kct)* and the block ciphertexts *(ct)* to the system. Afterwards, in order to contribute to the file, Bob downloads the keys and blocks. The operation that Bob is required to perform in order to decrypt the keys is:

$$K_n = D_{K_A}(kct_n)) \tag{11}$$

However, as Bob does not know Alice's secret key K_A he is unable to recover the keys. Hence, file sharing is not possible when utilizing secret key per user block key encryption feature.

10.3. Secret key per file block key encryption

In this scenario, Alice generates a secret key K_f for the file. Then she divides the file, applies convergent encryption and encrypts the block keys like in

equation (11), but using K_f instead of K_A. Then she uploads the key and block ciphertexts to the system and Bob retrieves them. As before, Bob is unable to decrypt the keys, as he does not know K_f. However, in this case the key K_f is not personal for Alice, it is generated only for this file, and Alice can tell Bob the key. After sharing the key K_f the file sharing works equally to the scenario where block key encryption was disabled. To conclude, file sharing is possible in secret key per file block key encryption, after the secret key of the file has been distributed to all collaborators.

10.4. Block key chaining

This time, Alice and Bob are utilizing block key chaining. Alice applies convergent encryption and encrypts all keys except the first like in equation (6). Then she uploads the key and block ciphertexts and Bob retrieves them. Then Bob tries to decrypt the key chain, like in equation (7), but fails because he does not know the first block key K_1, that is stored locally in Alice's computer. Similarly to secret key per file scenario, he is able to decrypt the chain if and then the file, if Alice provides him with K_1. However, if Bob now contributes to the file and modifies blocks including the first one, the entire chain is affected. Thus, when he uploads his contributions and Alice tries to retrieve the file, she fails, as the locally stored K_1 is not valid anymore. As a conclusion, file sharing works, but only as long as the first block key K_1 is distributed to all collaborators after every modification of the first block.

10.5. Signature checking at the gateway

Signatures depend on both the data and the private key of each user. Thus, no user is able to append any collaborator's signatures into the blocks. As a result, signature checking prevents file sharing completely.

10.6. Access revocation in file sharing

As previously concluded, file sharing in a system utilizing key chaining works only as long as the first block key is distributed to all collaborators after each modification. Thus, access of a collaborator can be revoked by modifying the first block and not sending him or her the first block key. However, as the block keys in convergent encryption are data bound, the user whose access was revoked can still decrypt all unmodified blocks if he or she memorized

the keys while the access was granted. Access revocation is complete only when all the blocks have been modified and re-encrypted after changing the secret key.

When utilizing secret per file key encryption, the access of a collaborator can be revoked only by re-encrypting all the block keys with a different secret key. Afterwards, the new secret key has to be distributed to the existing collaborators. Similarly to key chaining, the access revocation is complete only after modifying and re-encrypting every block in the file.

10.7. File sharing summary

Using signature checks at the gateway or secret key per user block key encryption prevents file sharing. For key chaining, file sharing requires secret distribution after each modification of the first block of the file. For secret key per file block key encryption, file sharing only requires initial secret distribution. In terms of access revocation, modification and re-encryption of each block in the file is required.

11. Prototype architecture and implementation

This chapter first gives an overview of the prototype architecture and then presents the implementation choices that were made by component.

11.1. Overview

The prototype is a block-level deduplication system that consists three components: the client, the gateway and the metadata manager. The prototype can be configured to three different security levels, that were presented in chapter 9. The implementation language is Python.

11.2. Client

The client is a Windows synchronization folder, like the Dropbox desktop application [22]. The folder replicates all files in the folder into the system in the background. Furthermore, all changes to the files are reflected to the remote side after 2 seconds of inactivity. The remote side also keeps state of

the file structure, so that upon download the files are placed in the correct folders. To improve user experience, all files in the synchronization folder are decorated with an overlay icon. The icon reflects the state of the file. For example, when a new file is added to the folder, it is decorated with a blue synchronization icon. After the background upload of the new file is complete, the icon is changed to a green ok sign.

The actual operation that occurs upon file modification is that the entire file is uploaded, but most of the blocks are deduplicated, resulting in only light traffic to the metadata managers upload link. The alternative would be to store a state or a hash for each block in a local database to determine which blocks are altered in the change.

Storing a local plaintext copy of the users files is a security risk in the sense that anyone that potentially has access to his or her computer, can directly access the files. On the other hand, not storing a local copy requires the system to download and decrypt the files upon access. The usability was considered a higher priority. As a result, the files are stored in the synchronization folder in plain text.

11.3. Gateway

The gateway is implemented as a plug-in for squid [23] proxy. The gateway stores one secret key and applies symmetric encryption to all the blocks, before sending them to the metadata manager.

11.4. Metadata manager

The metadata manager receives the blocks and the corresponding keys from the gateway and uploads non-duplicate blocks into the cloud storage. Internally, the metadata manager consists of three separate entities. To communicate with the gateway, the metadata manager runs a Tornado web server [24]. The core that performs deduplication is based on Redis [26]. To communicate with different cloud storage providers, the metadata manager has a Libcloud [25] based interface.

The structure of the redis database is presented in Table 3:

Symbol	Dataset index	Key structure	Value structure
F	0	F:user_id:file_id	{'name': fname}
K	0	K:file_id	[key1, key2,...]
FB	0	FB:file_id	[b_id0,b_id1,...]
B	0	B:block_id	{'storage_container': p.container.name 'storage-object': pointer.name 'counter': number_of_owners}
U	1	U:uid	{'password': pwd}
U	1	U:uid:session_key	session_key with expiration

Table 3: metadata structure in redis

For each file, the filename, the block keys and block identifiers are stored. The filename contains the path information so that the file tree can be constructed at the client side. The block identifiers are 256-bit hashes of the blocks and are stored to know which blocks belong to the file. For each block, the storage information is stored so that the block can be retrieved from the cloud. The deduplication applies if a new block has an identifier that already exists in the system. In that case the number of owners is increased for the block. Correspondingly, when a user deletes a block, the counter is decreased, and the block is only removed from the cloud if the counter value is 0 afterwards.

12. Prototype access control

The previously introduced security features provide confidentiality by cryptographic design. In other words, they deny any third parties the ability to decrypt the files in the system.

This chapter presents how access control can be enforced by authentication and authorization features, like credential based authentication. As opposed to data confidentiality by encryption, access control mechanisms restrict the ability of third parties to retrieve the data or the keys. The only access control mechanism presented in DupLESS is the certification based authentication into the key server. Also ClouDedup proposes a singular authentication

method, which is appending signatures to each block and checking them at the gateway to verify ownership.

Considering the prototype, the goal was to build sufficient access control to allow disabling security layers that impose performance or feature restrictions, like the secret key per user encryption of block keys that prevents file sharing. As an example, if the system does not allow any other user than the owner to retrieve a file, then it is acceptable that a third party has the ability to decrypt it. This kind of access control is crucial for the prototype system if security level is minimum or medium, because every user of the system has the ability to decrypt any file they can retrieve. In maximum security every user only possesses the ability to decrypt his or her own files, because of the secret key per user encryption of block keys. However, even in the maximum-security level, the other users would be able to perform dictionary COF and COB attacks against the data of other people's files, if they had the ability to download it. This is because the additional encryption of the data is decrypted by the gateway during the download, so the data downloaded by any other user than the owner is vulnerable to COB attacks.

12.1. Client access control

The synchronization folder client allows only a single user. It is not possible for multiple users to operate on the same desktop client. The access control is implemented by symmetrically encrypting the user's secret key with the hash his or her password as a key. The hash is computed by scrypt [2], that provides good resistance to brute forcing.

12.2. Gateway access control

The prototype provides no built-in authentication for the gateway. However, if the component is located in a company's premises, it can be configured to accept requests only from the local area network. Furthermore, squid already provides all the most common authentication interfaces. Thus, the company can configure the authentication to reflect their policies by simply editing the squid configuration.

12.3. Metadata manager access control

The metadata manager has a simple credential authentication. Upon user registration, the hash of the new user's password is stored among the

metadata. When a user logs in to his or her client, an authentication request is made by submitting the username and the scrypt hash of the password. As a response, the user receives a session key. The metadata manager keeps track of active session keys to validate any file operation requests.

13. Prototype credential management and rotation

This chapter presents solutions considering management and rotation of secret keys and credentials of users.

13.1. Secret key management

The easiest approach to client key management is not to take responsibility of the keys, and let the clients back up their keys themselves. Actually, people in IT-companies usually have many other keys and passwords to backup too, like hard drive encryption keys, passwords to Internet sites and various other resources and services. Hence, backing up one more key is not necessarily a major issue. Furthermore, big companies have policies for key management, so they might prefer dealing with it themselves. Anyway, if the company insists not to take care of the keys for the system, then they must be stored at a server. One might do some kind of clever trick, like for example encrypting the keys with the users password or the hash of it, and storing them in the metadata manager. However, further caution is recommended. A good principle is that the key storage should not be able to gain access to the keys. In the example case the metadata manager would be able to try bruteforcing the passwords in order to retrieve the keys. Also, the password hash might be used as an authentication method, and if the secret keys are also encrypted with the password hash, then the metadata manager is only required to decrypt the keys with the password hashes that it already knows.

The key management implementation depends on the company policies, like whether or not they should have cleartext backups. Thus, the first version of the prototype leaves key management entirely to the companies.

13.2. Credential and secret key rotation

The prototype allows change of password by the client. In order to change his or her password, the user selects a new password, and sends a request with

the old and the new password hashes to the metadata manager. Upon success, the secret key is re-encrypted with the new password and rewritten at disk. The administrator of the metadata manager can force a user to change password. If password change is forced, the user is prompted for a new password as a response to an authentication request.

Considering the secret key rotation, the first version of the prototype leaves this to the users. In order to change their secret key, they are required to remove their files from the synchronization folder, generate a new key and put the files back. The performance cost of lacking a specific key rotation service is that the users need to re-upload also the data, instead of just uploading the re-encrypted block keys. The upload of data is not necessary in theory, as the secret key rotation affects only the encryption of the block keys.

14. Prototype file sharing support

14.1. Versioning

In terms of file sharing support, Local and remote version counters were added to each file. On the remote side, the counter is initialized to one when a file is uploaded the first time. After this, the remote side accepts only modifications that have version one greater than the current acknowledged version. On the local side, the file version is also initially set to one, and then incremented by one upon every successful modification. This way, the system guarantees that consistency is preserved in the remote side. For example, if Alice and Bob are sharing a file with version 5, and then Alice updates the file, the remote version is incremented to 6. Now, if Bob modifies his version of the file, the modification arrives at the remote side with version number 6, which is not exactly one greater than 6. As a result, the remote system detects the inconsistency, rejects the modification, and sends Bob the updated version of the file. Then, depending on Bob's privileges in the system, he might be provided with options to accept or override Alice's changes.

14.2. Summary

The first version of the prototype does not support file sharing between different users, but allows file sharing between two devices of the same user. The minimum and medium security levels allow cross-user file sharing directly, but the maximum-security level requires configuration. In order to allow cross-user file sharing with maximum level of security, a new secret key

has to be generated for each shared file and distributed to the collaborators. None of the prototype security levels prevent cross-user file sharing. The feature is likely to be added in the next version.

15. Metadata manager upload buffering

This chapter presents a solution that was added to reduce the client-experienced latency of upload requests when the metadata manager's upload link to the cloud is the throughput bottleneck. The decrease in latency is achieved by storing the received blocks in a local buffer and sending a response immediately after the blocks have been received, as opposed to responding after the blocks have finished uploading to the cloud. Sending the confirmation before the blocks have finished uploading leaves a possibility of losing the blocks if the machine is lost. Different approaches to buffer implementation are featured, along with corresponding disaster recovery strategies. Finally, a known security weakness of the buffer solution is presented.

15.1. Advantages and disadvantages of upload buffering

The upload link from metadata manager to the cloud is the throughput bottleneck of the system, so if blocks are buffered and uploaded in the background, it is possible to respond to the client faster than if one would have to wait until every block has been uploaded. As an additional bonus, such operation would make the upload response time depend on the data buffering time instead of the actual upload time, which would remove the client's ability to detect deduplication by timing upload requests. Furthermore, the buffer could be utilized during a short period of cloud unavailability. This would result in higher availability, but only until the buffer becomes full. However, Buffering blocks introduces a replication problem in case the buffer is a local database, or a memory problem, if the blocks are kept in memory.

15.2. Different approaches to buffer implementation

Considering the prototype, the most straightforward solution is to buffer the blocks in Redis. It is automatically disaster-safe, as redis already utilizes master-slave replication. However, storing the blocks in memory, especially

in multiple replicas, is expensive. Depending on the size of the buffer, it would require potentially gigabytes of memory in every replica.

An alternative solution is to store the blocks in a local database. This way, the blocks are stored on disk instead of memory, which guarantees them to survive crashes. Obviously, such solution requires the disk to be faster than the metadata manager's upload link. Otherwise the disk becomes the bottleneck of the system. Besides providing the ability to maintain the blocks upon a crash, the local database solution introduces a disaster recovery problem: if upload confirmation has been sent to the user and the entire machine is destroyed, the blocks in the local database are lost, but the user assumes that his or her file is safely stored in the system.

15.3. Disaster recovery

The most straightforward way of addressing the disaster recovery problem is to replicate the local database to a slave. Despite the fact that the cost of this operation is actually the same than the cost of uploading the data directly to the cloud, the solution is viable if the master has a faster connection to the slave than to the cloud. Obviously, uploading the database to the slave is not a good option if it is not much more efficient than uploading the blocks directly to the cloud.

In the prototype, the metadata manager uses a local database. To address the replication problem, the response could be delayed until the database is replicated to a slave in the same network. As a result, the system would be disaster-safe, but the response time would be extended by the replication time. Instead, the database is not replicated. The justification for this is that it is very unlikely to lose the buffered files, as the synchronization folder keeps a local copy of the files. In the event of metadata manager crash or complete destruction, the slave takes over. The new master is unaware of the pending files in the old master's local buffer only until synchronization is made with the clients. The synchronization will re-upload any files that are present in the clients folder but not present in the remote side. To clarify, the system is using the clients local copy as a backup during a buffered upload. Thus, the files are only lost if the client's computer and the metadata manager master machine are both simultaneously destroyed. The risk is considered very unlikely to happen, but moderately severe. As a final result, the reductions in internal network load and user-visible response latency were considered worth taking the risk. Thus, the buffer is not replicated. As a result, the system imposes only very low latency to the users as long as the buffer is not full. Also, the solution allows the maximum buffer size to be very large, as it is only bound

by available disk space. In addition to that, the solution makes the system resistant to timing-based deduplication detection at the clients, denying them the possibility to perform COF attacks.

15.4. Known security weakness of the buffer

In a buffered system, a file upload latency is very low as long as the buffer is not full. This attribute can be exploited with certain accuracy. In order to do that, an adversary user is required to measure the time a packet spends in the buffer. To guarantee that the time is as constant as possible, the adversary must generate exclusively data that either is never or that is always deduplicated. The easier and more efficient way is to measure with data that is guaranteed to be deduplicated. For an example, uploading a dataset twice guarantees that it is deduplicated on the second time.

Once the adversary knows how long a packet stays in the buffer, he or she can detect duplicates by measuring the difference between response time and the time spent in the buffer. If the buffer is FIFO, which is very likely as it provides the highest efficiency, the adversary can also be sure that no other user can affect his measurement by sending any intermediate deduplicating or non-deduplicating data packets. Depending on the size of the buffer and the variance of the upload rate of the metadata manager upload link, the measurement may be fairly accurate. Protection against timing-based detection of deduplication in buffered systems can be improved by adding random variance to the upload rate of the metadata manager. Such addition would randomize the time the packet spends in the buffer. But even then, the adversary might succeed by repeating the experiments multiple times and measuring the average. In the prototype, this attack is a known weakness, and is to be fixed in the future.

16. Performance

16.1. Storage space saved by deduplication

Gaining access to datasets of real people's files is hard. The deduplication ability of the prototype was measured using a POP email dataset of Eurecom. The dataset contained a total of 9.35 GB emails of 1077 different users. The deduplicability was measured by fixed-sized and variable-sized chunking with the block size and the average variable block size configured to 8 KB. The space saved by deduplication was 27.1 % for variable-sized chunking and

0.6 % for fixed-sized chunking. The results indicate that, at least for this data set, the deduplication indeed saves substantial space and that variable sized chunking is more efficient in deduplication than fixed-sized chunking. Chunking by rabin-fingerprint was also concluded more efficient in deduplication than fixed-sized chunking in the paper *A study of practical deduplication* [4].

16.2. Throughput benchmarks

For the prototype system, the highest priority was upload throughput. The objective in throughput optimization is that no other component than the user's or the metadata manager's upload link ever limits the throughput. For an example, the client side performance would be unnecessarily limited, if the client's data encryption rate would be lower than the upload rate limit. On the server side, the performance would not be optimal, for example, if the metadata manager's data processing or redis-interaction speed would be slower than the maximum upload speed. On the client side, it is safe to assume that unless the client has a slow CPU and an exceptional upload bandwidth, because the symmetric encryption (AES) and rabin-fingerprinting are very efficient [10][7].

The theoretical throughput bottleneck for upload is the metadata managers upload link capacity into the cloud. To confirm this, the upload speed was measured when the cloud interface is uploading the blocks to the cloud and when the line is commented, so that it skips the actual data upload. The result was that the upload link is not only the bottleneck, it also severely handicaps the upload performance. Indeed, the results showed that Libcloud imposes a large overhead to the upload time of each file. In order to determine if parallelization would improve the performance, experiments were performed with simultaneous file upload requests to Amazon S3 eu west object storage [27].

16.2.1. Libcloud upload performance

The efficiency of making parallel upload requests with Libcloud was measured by uploading a 1 MB file with different number of parallel requests. In the tests, the file was uploaded when using a number of threads from 1 to 10. From each test run, the average block upload time and the total upload time for the file were calculated. The tests were executed 10 times. The final results were the average of those two values in the ten different test rounds. Doubled standard deviation of the different round averages was used as error margin for both values. The machine used for the tests was an Amazon M3-large instance [28] with 2 CPUs, 7,5 GB of RAM and a Gbps upload link.

Chart 1 presents the efficiency of libcloud when uploading a 1 MB file, divided into 99 blocks.

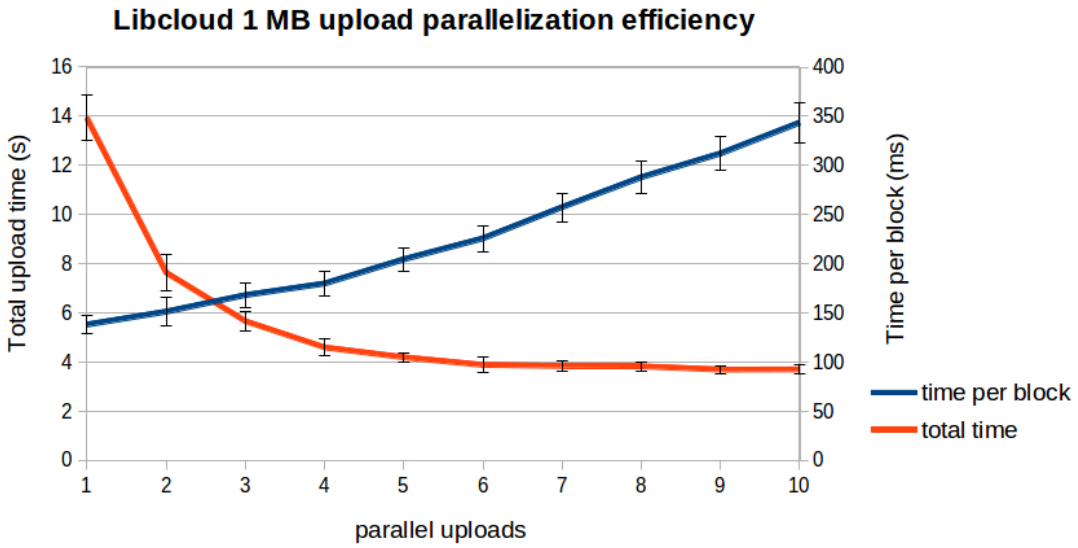

Chart 1: upload of 1 MB by libcloud with different amount of parallelization

The results indicate that increasing the number of parallel block upload requests also increases the time that each of the requests takes to complete. As a conclusion, increasing the number of threads after the first 5 does not improve performance significantly. To confirm this result, the same test was performed with 100 parallel threads. The total time measured was 3.53 s with an error margin of 0.23 s. Thus, the results indicate that even if all the blocks are uploaded in parallel, the upload rate will be limited to 2.27 Mbps, that in this case is about 0.2 % of the available bandwidth.

In order to fix the bottleneck, two steps can be taken. To trace whether the cause is Libcloud or amazon, an alternative interface can be built to communicate directly with the Amazon API. Both are very efficient when uploading one large file, but when uploading many very small blocks as different files the performance seems to be significantly lower. Implementing a new performance-optimized cloud connector for the prototype would allow testing the Amazon API and other APIs directly, and measure their performance. If the performance of the object storage APIs would be bad even with a performance optimized connector, then the only way left to improve the performance of the prototype would be to implement also a cloud storage.

16.2.2. Throughput dependency on file size

In theory, the file size should not affect the throughput of the system. To confirm this, upload, download and deduplication times were measured for 1, 5, 10, 20 and 50 MB files. Deduplication time is the time to upload a file that

already exists in the system and is not actually uploaded to the cloud. For the test execution, the client machine was a amazon S3-medium instance. The metadata manager machine was the same as in the upload parallelization tests. The tests were executed 10 times, and average upload, download and deduplication times were measured. The error margins, counted as doubled standard deviations of the values during the 10 rounds of tests, are not presented, as they were all less than a second. In terms of parallelization the system was configured to 100 simultaneous block uploads and downloads. The results are presented in Chart 2.

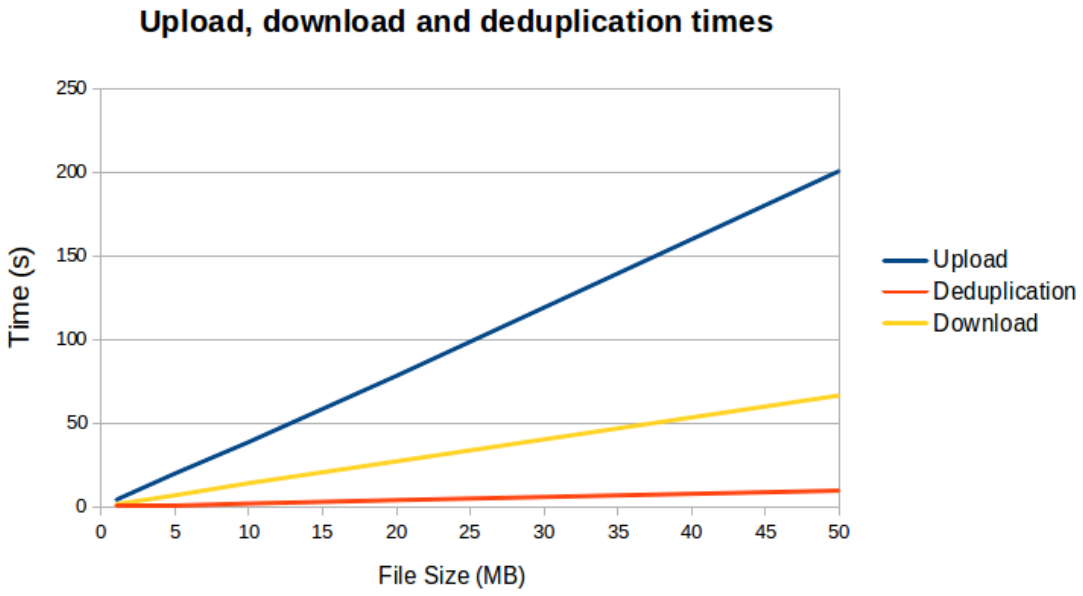

Chart 2: upload, download and deduplication time dependency on file size

The results confirm that the upload, download and deduplication times increase linearly as file size increases. In other words, the transfer rate is independent of the file size. For the 50 MB file, the upload, download and deduplication rates were 2.0 Mbps, 43.2 Mbps and 6.2 Mbps correspondingly. These values define the system performance with sufficient accuracy.

16.3. Upload buffer size impact on response time

As described in the implementation of the prototype, buffering can decrease the client visible latency. When a client uploads a file, a positive acknowledgement can be returned to the client as soon as the last block of the file is in the buffer. If there is not enough space in the buffer immediately for the entire file, the client has to wait until enough existing data is uploaded from the buffer to the cloud. To confirm that the response time decreases as a

function of the proportion of the file that immediately fits to the buffer, tests were performed. The upload response time of a 1 MB file was measured with different buffer sizes. The buffer size is presented as the percentage of the file that fits into the buffer. For client and the metadata manager, the same machines were used as in the throughput benchmark tests. Parallelization was configured to 10 threads, corresponding to about 10 % of the file being simultaneously uploaded, as the total number of blocks in the file was 99. The tests were executed 10 times and final values presented are the averages of the response times in different test rounds. The test margins are doubled standard deviations between the values of different rounds. Chart 3 presents the results.

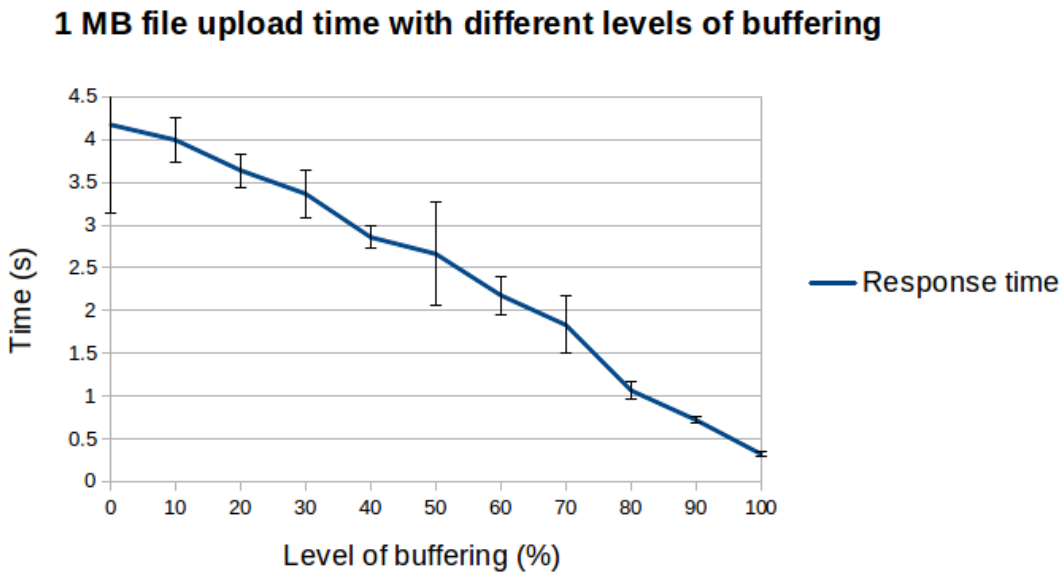

Chart 3: Response time observed at client for 1 MB file upload with different buffering levels

The results indicate that the response time indeed decreases linearly as the level of space in the buffer increases. The average response time for 0 % buffering was 4.2 s and for 100 % buffering 0.3 s. The times correspond to the upload rates of 1.9 Mbps without buffering and 24.7 Mbps with complete buffering. The round trip time between the client and the metadata manager was ignored in the rate calculations, as it was less than 20 ms.

16.4. Network overhead

The network overhead caused by the system was measured by monitoring the transferred bytes between the client and the metadata manager, and dividing that amount by the actual data that was transferred. In this case the actual

data means the encrypted data blocks, including padding for symmetric encryption, but excluding encoding overhead and all the metadata. Most of the network overhead is caused by base64 encoding of the data. As base64 encodes every 3 characters as 4, and occasionally applies one or two characters of padding, the overhead for a block is between 33,3 and 33,4 %. The overhead is slightly increased by the transferred filename and the block keys, which are also base64-encoded. The impact of filename is small, as it is transferred only once per file. The encrypted block keys, however, impose an effect of 64 bytes per block. As the average block size for rabin fingerprint file splitting is configured to 8 KB, the average overhead caused by the block keys is 0,8 %. The network overhead between the client and the metadata manager was measured for three files, 10 KB, 1 MB and 5 MB. The results are presented in Chart 4.

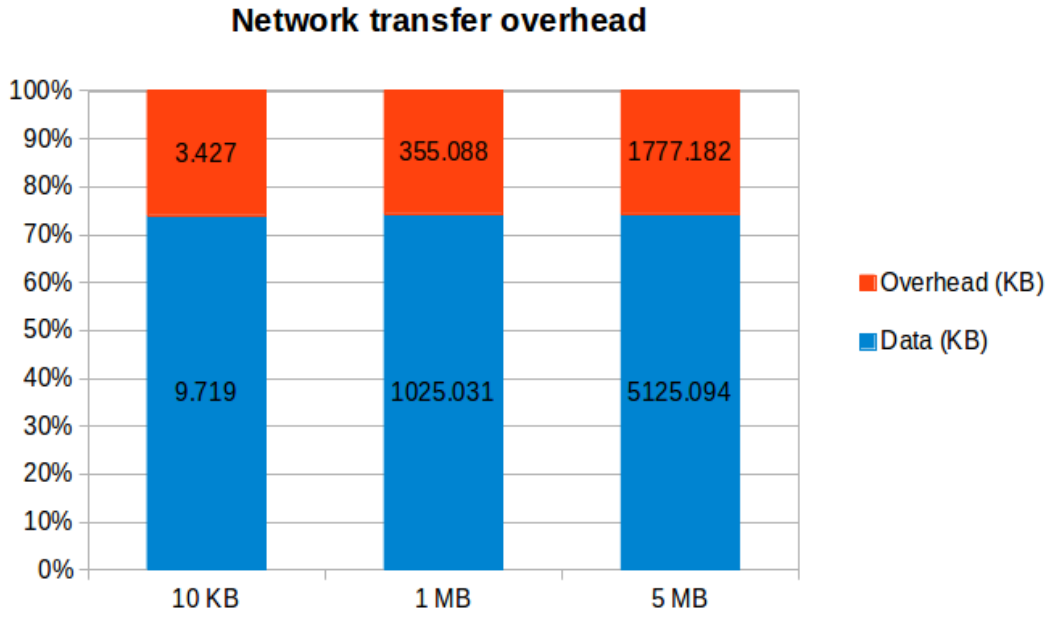

Chart 4: Network overhead between the client and the metadata manager

The overhead proportion of the non-encoded data was between 33.4 % and 33.6 % for all the files. This confirms that unless the file is extremely small, the amount of overhead is proportional to the file size.

16.5. Metadata memory usage

The metadata memory usage was measured by monitoring the memory usage of redis, with different amounts of data in the system. The results are presented in Chart 5.

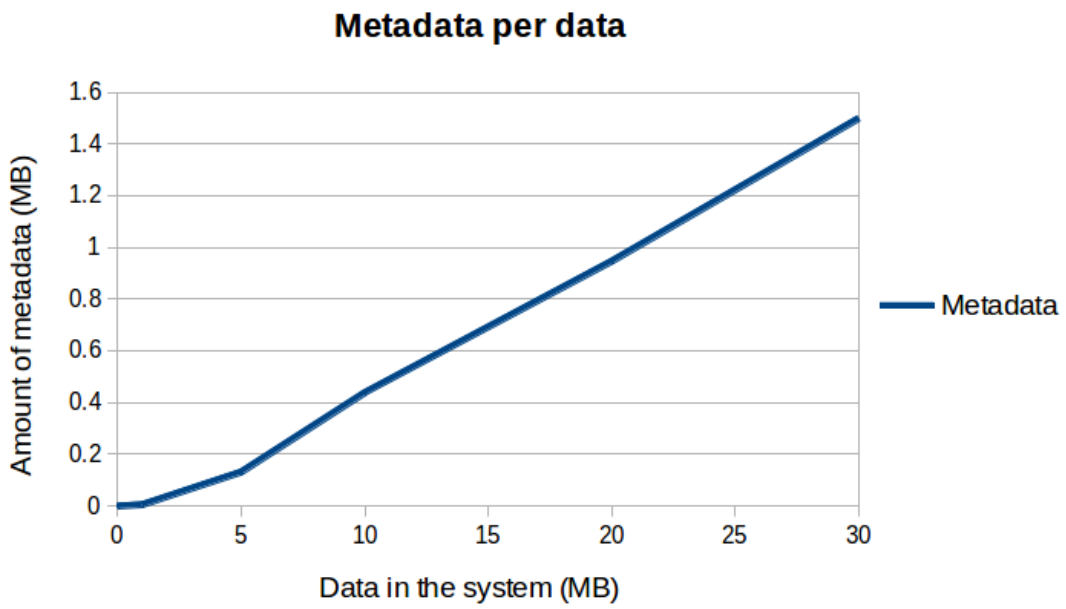

Chart 5: metadata size as a function of actual data stored in the system

The results indicate that the memory required for metadata is about 5 % of the data stored in the system.

16.6. Solutions to reduce metadata overhead

As the metadata manager requires a lot more memory than the keys require in theory, it is possible that the memory usage can be further optimized. The first optimization would be to optimize the symmetric encryption padding of the client. In the current implementation, the client utilizes the same symmetric encryption function for variable and fixed sized arguments. As a result, everything is padded, as a standard and safe procedure. For the keys and block IDs it means that even though they are already of the correct length, they are anyway appended with 32 bytes of padding. As a result, all the keys and block ids have 100 % (32 bytes) of padding overhead. The optimization could be easily applied by utilizing special non-padding encryption and decryption for these units.

16.7. Memory optimal metadata structure

As shown in Table 3, the current implementation stores the metadata as partially redundant key-value entries in Redis. In this context, partially redundant means that some pieces of information appear multiple times either as a key or as a value. A fully optimized solution would contain each

unique piece of information only once, either as a key, as a value or as a part of a value. However, as a block can be in multiple file, a division between per-block and per-file information is necessary. A memory optimized metadata structure is presented in Table 4.

Key	Value
F:file_id	'user_id': user_id, 'filename': filename, 'block_keys': [K_0, K_1,...],
B_block_id	'storage_info': block_storage_information, 'owner_counter': number_of_owners

Table 4: A memory optimized metadata structure

However, Redis supports hash maps as values but does not allow complex types like the block key list inside any hash map that is a value. To work around that the complex values would need to be serialized and stored as a string, which would increase access time. On the other hand, some entries are only searched when a client is downloading a file. As the download occurs rarely, the priority of the access time is low for certain entries, like the block keys.

16.8. Storing metadata in the cloud

As an advanced alternative, the memory consumption at the metadata manager could be reduced by encrypting all the metadata that is not crucial for deduplication and sending it to the cloud. As an example, the block keys are data bound, so they do not change before the data changes. On the other hand, the data changes only when the user updates his or her file. As a conclusion, the keys only change upon an update, and therefore could be pushed to the cloud. Also in terms of security, the solution would be approximately as strong as the current one, considering that the keys would be safe from the cloud unless the metadata manager is compromised. In order to store the keys in the cloud, each file entry would require containing the necessary storage information to retrieve them. However, the storage info is of constant size, as opposed by a key per block that depends on the number of blocks in the file. Also, if the post-mm additional encryption is already utilized, this operation has no additional key management cost. Otherwise the cost is to store one symmetric encryption key at the metadata manager.

17. Discussion

This thesis studied the state of the art of block-level deduplication system security features, their provided security and their impact on key management and file sharing. An important discovery was made: the key chaining feature, presented in Cloudedup [5], should in fact never be used. It was shown that, considering the encryption of convergent encryption block keys, applying symmetric encryption with a randomly generated key for each file is a strictly better alternative to key chaining. The provided level of security is stronger, the restrictions imposed on file sharing are less severe and the features are equal in terms of performance and key management cost.

The problem of selecting security features to provide a desired level of security was adressed and as a result, three sets of features were presented as security levels. The introduced levels are sets that provide a certain strength against an attack at different components of the deduplication system, and result in minimal key management costs and minimum restrictions on cross-user file sharing. In fact, the first two levels preserve block-level, cross-user deduplicability and cross-user file sharing support while requiring only very light key management and providing a very strong level of security against attacks at the metadata manager or at the cloud storage. They are only vulnerable against an adversary that has access to the gateway component, which is in practice always located at the company's premises and was, for instance, considered semi-trusted in Cloudedup [5]. In the maximum-security level, the adversary has to gain access also to the client's secret key for the target file. As a result, a key management cost of one key plus one key per share file is imposed on each client. Also, the secret key encryption of block keys that is applied at the client demands that in order to share a file between users the encryption key of the file must be initially shared among them.

The amount of metadata that was generated in practice was measured to be about 5 % of the original data. As a result, only data that have an average deduplication ratio greater than 5 % should be stored in the system, otherwise space will be lost instead of saved. Also, it should be taken into consideration that the metadata needs to be replicated. With one metadata replica, the average deduplication rate for the data stored should be greater than 10 %. However, the deduplication rate was tested to be over 27 % for a large set of real user email data. Furthermore, related studies have measured deduplication ratios as high as 68 % for file systems [4], 80 % for virtual images [16] and 92,7 % for backups [11]. On the other hand, the viability threshold can be reduced from 10 % by decreasing the amount of metadata.

18. Related work and future improvements

This chapter presents features in related studies that could be applied to improve the prototype but were outside the scope of this thesis.

18.1. Metadata reduction

A recent study *Frequency based chunking for data de-duplication* introduced frequency based chunking and measured that in order to achieve the same duplicate elimination ratio than that of conventional content defined chunking, the number of chunks can be from 2.5 to 4.0 times less [12]. This would mean equal decrease to the amount of metadata, as it depends on the number of chunks.

Another method to reduce the amount metadata is decreasing the amount of space required for the keys, as proposed in a recent paper *Secure deduplication with efficient and reliable convergent key management* [13]. In the paper, the storage of keys is proposed to be performed using Ramp secret sharing scheme [14][15]. It is mentioned in *Efficient sharing (broadcasting) of multiple secrets* that "the scheme allows a group of *n* users to share multiple secrets". This means that each block key is stored only once, instead of storing an encrypted key for each user, like in the key management scheme of Cloudedup. Furthermore, in the latter scheme, it is not enough to store a key for each user; it is also necessary to replicate the key storage, which is not required by the former scheme.

18.2. Improving scalability

A recent study *Building a High-performance Deduplication System* [17] explores the difficulties in achieving high deduplication efficiency, scalability and throughput at the same time. It is shown, that only two of these can be simultaneously achieved without compromises. In this thesis, the system is optimized to high deduplication effectiveness and throughput, but lacks scalability in the sense that Redis stores the index of existing blocks in memory. In *Building a High-performance Deduplication System* a design is presented that provides good deduplication effectiveness, scalability and throughput for a single node deduplication system, like the one in this thesis. The scalability issue of the index has also been studied in *Sparse Indexing: LargeScale, Inline Deduplication Using Sampling and Locality* [18]. The study presents sampling and locality based indexing that provides good scalability in terms of RAM usage, but allows some loss of deduplicability. The idea to use locality in indexing was first presented in *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System* [19]. More recently, different

techniques and their effect on deduplication in single- and multi-node systems has been studied in general level in *A survey of indexing techniques for scalable record linkage and deduplication* [20]. The study shows that indexing is a non-trivial problem and the right solution depends on the use case.

Scalability can also be improved by storing the index on an optimized disk, like in the study *ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory* [21]. The average lookup time of the presented flash drive solution was observed to be from 7 to 60 times faster than that of a solution on a non-optimized normal disk and 2 to 4 times faster than that of a solution on a non-optimized flash disk.

19. Conclusions

The fundamentals of deduplication and convergent encryption were explained. An economical motivator for constructing a deduplication product was given. The state of the art security features were analyzed in depth and three levels of security were constructed for the prototype.

After in depth analysis of state of the art security features, secret per file encryption of block keys was introduced as an alternative to key chaining and per user secret key encryption. The feature was observed to allow file sharing with initial secret distribution, as opposed by secret distribution after each modification of the first block, as required by key chaining. Furthermore, the feature was observed to provide better security than key chaining with the same key management cost.

A block-level deduplication system prototype was implemented and the performance was measured. The primary goals of the project were accomplished in the sense that while maintaining a high level of security the prototype provides substantial space savings with cross-user block-level deduplication. A severe throughput bottleneck was identified, and a buffering solution was introduced to reduce the bottleneck impact on end-users.

File sharing support was provided for different devices of the same user. Solutions were given to add cross-user file sharing in the future.

20. References

1. Bellare, M., Keelveedhi, S., & Ristenpart, T. DupLESS: server-aided encryption for deduplicated storage. In *Proceedings of the 22nd USENIX conference on Security* (pp. 179-194). USENIX Association.
2. Eröcal, B. SCrypt: Using Symbolic Computation to Bridge the Gap Between Algebra and Cryptography. In *First International Conference on Symbolic Computation and Cryptography* (p. 61).
3. Harnik, D., Pinkas, B. ja Shulman-Peleg, A., 2010. Side channels in cloud services: Deduplication in cloud storage. *Security & Privacy, IEEE,* **8**(6), (pp. 40-47).
4. Meyer, D.T. & Bolosky, W.J., 2012. A study of practical deduplication. *ACM Transactions on Storage (TOS),* **7**(4), (pp. 14).
5. Puzio, P., Molva, R., Onen, M., & Loureiro, S. ClouDedup: secure deduplication with encrypted data for cloud storage. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on* Vol. 1, (pp. 363-370). IEEE.
6. Rabin, M.O., 1981. *Fingerprinting by random polynomials* (pp. 15-18). Center for Research in Computing Techn., Aiken Computation Laboratory, Univ.
7. Storer, M. W., Greenan, K., Long, D. D. & Miller, E. L. Secure data deduplication. *Proceedings of the 4th ACM international workshop on Storage security and survivability* (pp. 1-10). ACM.
8. Douceur, J. R., Adya, A., Bolosky, W. J., Simon, P., & Theimer, M. Reclaiming space from duplicate files in a serverless distributed file system. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (pp. 617-624). IEEE.
9. Bellare, M., Keelveedhi, S. & Ristenpart, T. Message-locked encryption and secure deduplication. *Advances in Cryptology–EUROCRYPT 2013* (pp. 296-312). Springer Berlin Heiderberg.
10. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C. & Ferguson, N. Performance comparison of the AES submissions.
11. Bhagwat, D., Eshghi, K., Long, D. D., & Lillibridge, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on* (pp. 1-9). IEEE.
12. Lu, Guanlin, Yu Jin, & David HC Du. Frequency based chunking for data de-duplication. *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on* (pp. 287-296). IEEE.
13. Li, J., Chen, X., Li, M., Li, J., Lee, P. P., & Lou, W. Secure deduplication with efficient and reliable convergent key management. In *Parallel and Distributed Systems, IEEE Transactions on,* 25(6), (pp. 1615-1625).

14. Blakley, G. R., & Meadows, C. Security of ramp schemes. In *Advances in Cryptology* (pp. 242-268). Springer Berlin Heidelberg.
15. Harn, L. Efficient sharing (broadcasting) of multiple secrets. *IEE Proceedings-Computers and Digital Techniques*, 142(3), (pp. 237-240).
16. Jin, K., & Miller, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (p. 7). ACM.
17. Guo, F., & Efstathopoulos, P. Building a High-performance Deduplication System. In *USENIX Annual Technical Conference*.
18. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezis, G., & Camble, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Fast* Vol. 9, (pp. 111-123).
19. Zhu, B., Li, K., & Patterson, R. H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Fast* Vol. 8, (pp. 1-14).
20. Christen, P. A survey of indexing techniques for scalable record linkage and deduplication. *Knowledge and Data Engineering, IEEE Transactions on*, **24**(9), (pp. 1537-1555).
21. Debnath, B. K., Sengupta, S., & Li, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *USENIX annual technical conference*.
22. https://www.dropbox.com/
23. http://www.squid-cache.org/Intro/
24. http://www.tornadoweb.org/en/stable/
25. https://libcloud.apache.org/
26. http://redis.io/
27. http://aws.amazon.com/s3/
28. http://aws.amazon.com/ec2/instance-types/