RZ 3852 (# ZUR1308-022) 09/05/2013
Computer Science 26 pages

Research Report

A Secure Data Deduplication Scheme for Cloud Storage

Jan Stanek‡, Alessandro Sorniotti*, Elli Androulaki*, Lukas Kencl‡

‡Czech Technical University in Prague

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

A Secure Data Deduplication Scheme for Cloud Storage

Jan Stanek[*] Alessandro Sorniotti[†] Elli Androulaki[†] Lukas Kencl[*]

August 12, 2013

Abstract

Nowadays, more and more corporate and private users outsource their data to cloud storage providers. At the same time, recent data breach incidents make end-to-end encryption an increasingly prominent requirement. Unfortunately, semantically secure encryption schemes render various cost-effective storage optimization techniques, such as data deduplication, completely ineffective. In this paper, we present a novel encryption scheme that guarantees semantic security for unpopular data and provides weaker security and better storage and bandwidth benefits for popular data. This way, data deduplication can be effective for popular data, whilst semantically secure encryption protects unpopular content, preventing its deduplication. Transitions from one mode to the other take place seamlessly at the storage server side if and only if a file becomes popular. We show that our scheme is secure under the Symmetric External Decisional Diffie-Hellman Assumption in the random oracle model, and evaluate its performance with benchmarks and simulations.

1 Introduction

With the rapidly increasing amounts of data produced worldwide, networked and multi-user storage systems are becoming very popular, thanks to their accessibility and moderate cost. However, one obstacle still prevents many users from migrating data to remote storage – data security. The conventional means to address concerns over the loss of governance for outsourced data is to encrypt it before it leaves the premises of its owner. While sound from a security perspective, this approach prevents the storage provider from applying any space- or bandwidth-saving functions. The effectiveness of storage efficiency functions, such as compression and deduplication, is an objective for both storage provider and customer: indeed, high compression and deduplication ratios allow optimal usage of the resources of the storage provider, and consequently, lower cost for its users. In particular, we focus our attention on deduplication: data deduplication ensures that multiple uploads for the same content only consume the network bandwidth and the storage space for a single upload. Deduplication is actively used by a number of cloud backup providers (e.g. Bitcasa) as well as various cloud services (e.g. Dropbox). It is arguably one of the main reasons why prices for cloud storage have dropped so sharply. Unfortunately, encrypted data is pseudorandom and thus cannot be deduplicated: as a consequence, current schemes have to entirely sacrifice either security or storage efficiency.

In this paper, we present a scheme that permits a more fine-grained trade-off. The intuition behind the solution is that outsourced data may require different degrees of protection,

[*]Czech Technical University in Prague. (jan.stanek|lukas.kencl)@fel.cvut.cz.
[†]IBM Research - Zurich, Rüschlikon, Switzerland. (aso|lli)@zurich.ibm.com.

depending on how *popular* it is: content that is shared by many users, such as a popular song, movie or install package, arguably requires less protection than a personal document, the copy of a payslip or the draft of an unsubmitted scientific paper.

Around this intuition we build the following contributions:

- we present $\mathcal{E}_\mu$, a novel threshold cryptosystem (which can be of independent interest), together with a security model and formal security proofs;
- we introduce a scheme that uses $\mathcal{E}_\mu$ as a building block and enables to leverage popularity as a means to achieve both security and storage efficiency and discuss its overall security;
- we present performance evaluation, both of its computational overhead using a real implementation, and of its ability to reduce storage-space using a simulation, demonstrating the practicality of the scheme;

The rest of the paper is structured as follows: in Section 2 we state the problem and review the state-of-the-art. Section 3 contains a high-level overview of our scheme, as well as system and security model. Preliminary building blocks are described in Section 4, while Section 5 contains a detailed description of the scheme. Section 6 discusses the security of the cryptosystem and of the scheme as a whole; in Section 7 the scheme performance is evaluated. Possible extensions and system limitations are discussed in Section 8, while Section 9 contains our concluding remarks.

2 Problem Statement & Related Works

2.1 Data Security or Storage Efficiency?

Storage efficiency functions such as compression and deduplication afford storage providers a better utilization of their storage backends and the ability to serve more customers with the same infrastructure. Data deduplication is the process by which a storage provider only stores a single copy of a file that is owned by several of its users. There are four different deduplication strategies, depending on whether deduplication happens at the client side (i.e. before the upload) or at the server side, and whether deduplication happens at a block level or at a file level. Deduplication is most rewarding when it is triggered at the client side, as it also saves upload bandwidth. For these reasons, deduplication is a critical enabler for a number of popular and successful storage services (e.g. Dropbox, Memopal) that offer cheap, remote storage to the broad public by performing client-side deduplication, thus saving both the network bandwidth and the storage costs associated with processing the same content multiple times. Indeed, data deduplication is arguably one of the main reasons why the prices for cloud storage and cloud backup services are dropping so sharply compared to a few years ago.

Unfortunately, deduplication loses its effectiveness in conjunction with end-to-end encryption. End-to-end encryption in a storage system is the process by which data is encrypted at its source prior to ingress into the storage system, and is always only present as ciphertext within. End-to-end encryption is becoming an increasingly prominent requirement because of both the increasing number of security incidents linked to leakage of unencrypted data [3] and the tightening of sector-specific laws and regulations. Also, companies like VMware and Microsoft[1] will

[1]Microsoft is a trademark of Microsoft Corporation in the United States, other countries, or both. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies.

be providing VM disk encryption in their hypervisors, making end-to-end encryption a strong reality in cloud systems. Clearly, if semantically secure encryption is used, file deduplication cannot take place, as no one—apart from the owner of the decryption key—can decide whether two ciphertexts correspond to the same plaintext file. Trivial solutions, such as forcing users to share encryption keys and/or using deterministic encryption, fall short of providing acceptable levels of security, since users can formulate guess-ciphertexts and use deduplication as an oracle that reveals whether some other user has uploaded the corresponding plaintext.

As a consequence, storage systems are expected to undergo major restructuring to maintain the current disk/customer ratio in the presence of end-to-end encryption. The design of storage efficiency functions in general and of deduplication functions in particular that do not lose their effectiveness in presence of end-to-end security is therefore still an open problem.

2.2 State-of-the-art

Several deduplication schemes have been proposed by the research community [25, 24, 7] showing how deduplication allows very appealing reductions in the usage of storage resources [16, 20]. Deduplication is widely adopted in practice for instance by services such as Bitcasa [1], Ciphertite [2] and Flud [4].

Most works do not consider security as a concern for deduplicating systems; recently however, Harnik *et al.* [21] have presented a number of attacks that can lead to data leakage in storage systems in which client-side deduplication is in place. To thwart such attacks, the concept of proof of ownership has been introduced [19, 13]. None of these works, however, can provide real end-user confidentiality in presence of a malicious or honest-but-curious cloud provider.

Convergent encryption is a cryptographic primitive introduced by Douceur*et al.* [15, 28], attempting to combine data confidentiality with the possibility of data deduplication. Convergent encryption of a message consists of encrypting the plaintext using a deterministic (symmetric) encryption scheme with a key which is deterministically derived solely from the plaintext. Clearly, when two users independently attempt to encrypt the same file, they will generate the same ciphertext which can be easily deduplicated. Unfortunately, convergent encryption does not provide semantic security as it is vulnerable to content-guessing attacks. Later, Bellare *et al.* [10] formalized convergent encryption under the name *message-locked encryption*. As expected, the security analysis presented in [10] highlights that message-locked encryption offers confidentiality for unpredictable messages only, clearly failing to achieve semantic security.

Xu *et al.* [29] present a PoW scheme that allows client-side deduplication in a bounded leakage setting. They provide a security proof in a random oracle model for their solution, but their work does not address the problem of low min-entropy files.

3 Overview, System and Security Models

The main intuition behind our scheme is that, in a deduplicated storage system, data may require different degrees of protection that depend on how *popular* a datum is. Let us start with an example: imagine that a storage system is used by multiple users to perform full backups of their hard drives. The files that undergo backup can be divided into those *uploaded by many users* and those *uploaded by one or very few users only.* Files falling in the former category (e.g. system binaries) will benefit strongly from deduplication because of their popularity and may not be particularly sensitive from a confidentiality standpoint. Files falling in the latter category, on the other hand, may contain user-generated content which requires confidentiality,

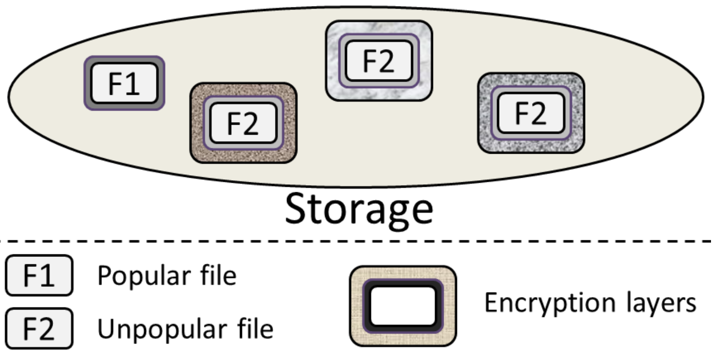


Figure 1: The multi-layered cryptosystem used in our scheme. Unpopular files are protected using two layers, whereas for popular files, the outer layer can be removed. The inner layer is obtained through convergent encryption that generates identical ciphertext at each invocation. The outer layer (for unpopular files) is obtained through a semantically secure cryptosystem.

and would by definition not allow a lot of space to be reclaimed through deduplication. The same reasoning can be applied to the common blocks of a VM image used by multiple VMs that adopt a copy-on-write sharing approach, to mail attachments sent to a large number of recipient, to reused code snippets, etc.

This intuition can be implemented cryptographically using a *multi-layered* cryptosystem. All files are initially declared unpopular and are encrypted with two layers, as illustrated in Figure 1: the inner layer is applied using a *convergent* cryptosystem, whereas the outer layer is applied using a semantically secure *threshold* cryptosystem. Uploaders of an unpopular file attach a *decryption share* to the ciphertext. In this way, when sufficient *distinct* copies of an unpopular file have been uploaded, the threshold layer can be removed. This step has two consequences: (i) the security notion for the now popular file is downgraded from semantic to standard convergent (see [10]), and (ii) the properties of the remaining convergent encryption layer allow deduplication to happen naturally. It is easy to see that security is traded for storage efficiency as for every file that transits from unpopular to popular status, storage space can be reclaimed.

There are two further challenges in the secure design of the scheme. First of all, if no proper identity management is in place, sybil attacks [14] could be mounted by spawning sufficient sybil accounts to force a file to become popular: in this way, the semantically secure encryption layer could be forced off and more information could be inferred on the content of the file, whose only remaining protection is the weaker convergent layer. While this is acceptable for popular files (provided of course that storage efficiency is an objective), it is not for unpopular files whose content – we postulate – has to enjoy stronger protection. The second issue relates to the need of every deduplicating system to group together uploads of the same content. In client-side deduplicating systems, this is usually accomplished through an *index* computed deterministically from the content of the file so that all uploading users can compute the same. However, by its very nature, this index leaks information about the content of the file and violates semantic security for unpopular files.

For the reasons listed above, we extend the conventional user-storage provider setting with two additional trusted entities: (i) an identity provider, that deploys a strict user identity control and prevents users from mounting sybil attacks, and (ii) an indexing service that provides a secure indirection for unpopular files.

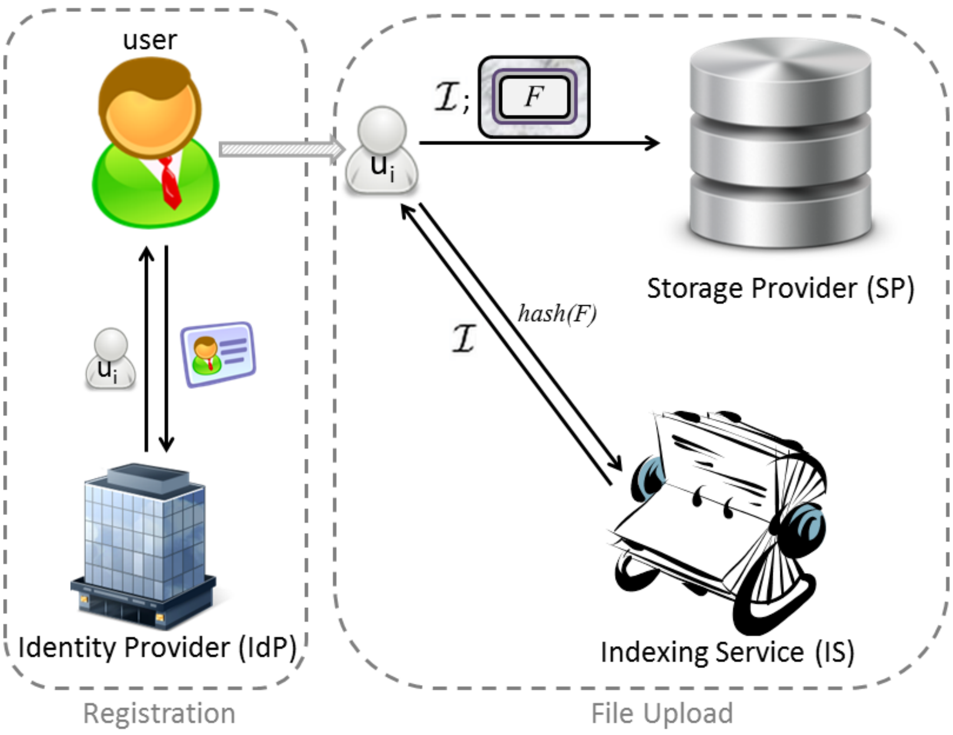


Figure 2: Illustration of our system model. The schematic shows the main four entities and their interaction for registration and file upload process.

3.1 System Model

Our system consists of *users*, a *storage provider* and two trusted entities, the *identity provider*, and the *indexing service*, as shown in Figure 2.

The storage provider (S) offers basic storage services and can be instantiated by any storage provider (e.g. Bitcasa [1], Flud [4] etc.). Users (U_i) own files and wish to make use of the storage provider to ensure persistent storage of their content. Users are identified via credentials issued by an identity provider IdP when a user first joins the system.

A file is identified within S via a unique file identifier ($\mathcal{I}$), which is issued by the indexing service IS when the file is uploaded to S. The indexing service also maintains a record of how many distinct users have uploaded a file.

3.2 Security Model

The objective of our scheme is confidentiality of user content. Specifically, we achieve two different security notions, depending on the nature of each datum, as follows:

- *Semantic security* [17] for unpopular data;
- *Conventional convergent security* [10] for popular data.

Note that integrity and data origin authentication exceed the scope of this work.

In our model, the storage provider is trusted to reliably store data on behalf of users and make it available to any user upon request. Nevertheless, S is interested in compromising the confidentiality of user content. We assume that the storage provider controls $n_{\mathcal{A}}$ users: this captures the two scenarios of a set of malicious users colluding with the storage provider and the storage provider attempting to spawn system users. We also assume that the goal of a malicious user is only limited to breaking the confidentiality of content uploaded by honest users.

Let us now formally define popularity. We introduce a system-wide popularity limit, p_{lim}, which represents the smallest number of *distinct*, *legitimate* users that need to upload a given file F for that file to be declared popular. Note that p_{lim} does not account for for malicious uploads. Based on p_{lim} and $n_{\mathcal{A}}$, we can then introduce the threshold t for our system, which is set to be $t \geq p_{lim} + n_{\mathcal{A}}$. Setting the global system threshold to t ensures that the adversary cannot use its control over $n_{\mathcal{A}}$ users to subvert the popularity mechanism and force a non popular file of its choice to become popular. A file shall therefore be declared *popular* once more than t uploads for it have taken place. Note that this accounts for $n_{\mathcal{A}}$ possibly malicious uploads.

The indexing service and the identity provider are assumed to be completely trusted and to abide by the protocol specifications. In particular, it is assumed that these entities will not collude with the adversary, and that the adversary can not compromise them. We also assume that communication between these entities and the user is properly secured using any known secure communication protocol (e.g. TLS/SSL).

4 Preliminaries

This section describes the building blocks for our scheme.

4.1 Modelling Deduplication

In this Section we shall describe the interactions between a storage provider (S) that uses deduplication and a set of users (U) who store content on the server. We consider client-side deduplication, i.e., the form of deduplication that "happens" at the client side, thus avoiding the need to upload the file and saving network bandwidth. For simplicity, we assume that deduplication happens at the file level. To identify files and detect duplicates, the scheme uses an indexing function $\mathcal{I}$: $\{0,1\}^* \rightarrow \{0,1\}^*$; we will refer to $\mathcal{I}_F$ as the index for a given file F. The storage provider's backend can be modeled as an associative array DB mapping indexes produced by $\mathcal{I}$ to records of arbitrary length: for example $\mathsf{DB}[\mathcal{I}_F]$ is the record mapped to the index of file F. In a simple deduplication scheme, records contain two fields, $\mathsf{DB}[\mathcal{I}_F].\mathsf{data}$ and $\mathsf{DB}[\mathcal{I}_F].\mathsf{users}$. The first contains the content of file F, whereas the second is a list that tracks the users that have so far uploaded F. The storage provider and users interact using the following algorithms:

- Put: user u sends $\mathcal{I}_F$ to S. The latter checks whether $\mathsf{DB}[\mathcal{I}_F]$ exists. If it does, the server appends u to $\mathsf{DB}[\mathcal{I}_F].\mathsf{users}$. Otherwise, it requests u to upload the content of F, which will be assigned to $\mathsf{DB}[\mathcal{I}_F].\mathsf{data}$. $\mathsf{DB}[\mathcal{I}_F].\mathsf{users}$ is initialized with u.

- Get: user u sends $\mathcal{I}_F$ to the server. The server checks whether $\mathsf{DB}[\mathcal{I}_F]$ exists and whether $\mathsf{DB}[\mathcal{I}_F].\mathsf{users}$ contains u. If it does, the server responds with $\mathsf{DB}[\mathcal{I}_F].\mathsf{data}$. Otherwise, it answers with an error message.

4.2 Symmetric Cryptosystems and Convergent Encryption

A *symmetric cryptosystem* $\mathcal{E}$ is defined as a tuple $(\mathsf{K}, \mathsf{E}, \mathsf{D})$ of probabilistic polynomial-time algorithms (assuming a security parameter κ). K takes κ as input and is used to generate a random secret key k, which is then used by E to encrypt a message m and generate a ciphertext c, and by D to decrypt the ciphertext and produce the original message.

A *convergent encryption scheme* $\mathcal{E}_c$, also known as message-locked encryption scheme, is defined as a tuple of three polynomial-time algorithms (assuming a security parameter κ) $(\mathsf{K}, \mathsf{E}, \mathsf{D})$. The two main differences with respect to $\mathcal{E}$ is that i) these algorithms are not probabilistic and ii) that keys generated by K are a deterministic function of the cleartext message m; we then refer to keys generated by $\mathcal{E}_c.\mathsf{K}$ as k_m. As a consequence of the deterministic nature of these algorithms, multiple invocations of K and E (on input of a given message m) produce identical keys and ciphertexts, respectively, as output.

4.3 Threshold Cryptosystems

Threshold cryptosystems offer the ability to share the power of performing certain cryptographic operations (e.g. generating a signature, decrypting a message, computing a shared secret) among n authorized users, such that any t of them can do it efficiently. Moreover, according to the security properties of threshold cryptosystems it is computationally infeasible to perform these operations with fewer than t (authorized) users. In our scheme we use *threshold public-key cryptosystem.* A threshold public-key cryptosystems $\mathcal{E}_t$ is defined as a tuple $(\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{DShare}, \mathsf{Decrypt})$, consisting of four probabilistic polynomial-time algorithms (in terms of a security parameter κ) with the following properties:

$\mathsf{Setup}(\kappa, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk_1}, \ldots, \mathsf{sk_n})$: generates the public key of the system pk and n *shares* $\mathsf{sk_i}$ of the private key, which are secretly provided to the authorized users.

$\mathsf{Encrypt}(\mathsf{pk}, m) \rightarrow (c)$: takes as input a message m and produces its encrypted version c under the public key pk.

$\mathsf{DShare}(\mathsf{sk_i}, m) \rightarrow (\mathsf{ds_i})$: takes as input a message m and a key share $\mathsf{sk_i}$ and produces a *decryption share* $\mathsf{ds_i}$.

$\mathsf{Decrypt}(c, \mathsf{ds_1}, \ldots, \mathsf{ds_t}) \rightarrow (m)$: takes as input a ciphertext c and a set of t decryption shares and outputs the cleartext message m.

5 Our scheme

In this Section we shall formally introduce our scheme. First, we will present a novel cryptosystem whose threshold and convergent nature make it a suitable building block for our scheme. We will then describe the role of our trusted third parties and finally we will detail the algorithms that compose the scheme.

5.1 $\mathcal{E}_\mu$: a Convergent Threshold Cryptosystem

This Section contains a formal description of the first contribution of this paper, namely, $\mathcal{E}_\mu$, a novel threshold cryptosystem, that will constitute a fundamental building block in the implementation of our scheme. Nonetheless, this also constitutes a contribution of independent interest which can be applied in other scenarios.

In the remainder of this paper we will make use of pairing groups $\mathbb{G}_1, g, \mathbb{G}_2, \bar{g}, \mathbb{G}_T, \hat{e}$, where $\mathbb{G}_1 = \langle g \rangle$, $\mathbb{G}_2 = \langle \bar{g} \rangle$ are of prime order q, where the bitsize of q is determined by the security parameter κ, and $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a computable, non-degenerate bilinear pairing. We further assume that there is no efficient distortion map $\psi : \mathbb{G}_1 \to \mathbb{G}_2$, or $\psi : \mathbb{G}_2 \to \mathbb{G}_1$. These groups are commonly referred to as SXDH groups, i.e., groups where it is known that the Symmetric Extensible Diffie Hellman Assumption [8] holds. Security of $\mathcal{E}_\mu$ is based on this assumption (see Section 6).

$\mathcal{E}_\mu$ is defined as a tuple (Setup, $\mathsf{Encrypt}$, DShare, $\mathsf{Decrypt}$), consisting of four probabilistic polynomial-time algorithms (in terms of a security parameter κ) implemented as follows:

$\mathsf{Setup}(\kappa, n, t) \to (\mathsf{pk}, \mathsf{sk_1}, \ldots, \mathsf{sk_n})$: at first, q, $\mathbb{G}_1$, g, $\mathbb{G}_2$, $\bar{g}$, $\mathbb{G}_T$ and $\hat{e}$ are generated as described above. Also, let x be a random element of $\mathbb{Z}_q^*$ and $\{x_i\}_{i=0}^n$ be shares of x such that any set of t shares can be used to reconstruct x through polynomial interpolation (see [27] for more details). Also, let $\bar{g}_{pub} \leftarrow \bar{g}^x$. Finally, let $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \mathbb{G}_T \to \{0,1\}^l$ for some l, be two cryptographic hash functions. Then, the public key pk is set to be $\{q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, H_1, H_2, g, \bar{g}, \bar{g}_{pub}\}$ and the i-th decryption share $\mathsf{sk_i}$ to be x_i.

$\mathsf{Encrypt}(\mathsf{pk}, m) \to (c)$: let r be chosen randomly from $\mathbb{Z}_q^*$ and let $E \leftarrow \hat{e}\left(H_1(m), \bar{g}_{pub}\right)^r$. Next, set $c_1 \leftarrow H_2(E) \oplus m$ and $c_2 \leftarrow \bar{g}^r$. Finally, output the ciphertext c as $c \leftarrow (c_1, c_2)$.

$\mathsf{DShare}(\mathsf{sk_i}, m) \to (\mathsf{ds_i})$: let $\mathsf{ds_i} \leftarrow H_1(m)^{\mathsf{sk_i}}$.

$\mathsf{Decrypt}(c, \mathsf{ds_1}, \ldots, \mathsf{ds_t}) \to (m)$: first parse c as (c_1, c_2); then compute

$$\prod_{\mathsf{ds_i} \in S} \mathsf{ds_i}^{\lambda_{0,i}^S} = H_1(m)^{\sum\limits_{i \in S} x_i \lambda_{0,i}^S} = H_1(m)^x,$$

where S is the set of decryption shares $\{\mathsf{ds_1}, \ldots, \mathsf{ds_t}\}$ and $\lambda_{0,i}^S$ are Lagrangian coefficients for the given set S. Then compute $\hat{E}$ as $\hat{e}\left(H_1(m)^x, c_2\right)$ and output $c_1 \oplus H_2(\hat{E})$.

Note that decryption is possible because, by the properties of bilinear pairings

$$\hat{e}\left(H_1(m)^x, \bar{g}^r\right) = \hat{e}\left(H_1(m), \bar{g}_{pub}\right)^r = \hat{e}\left(H_1(m), \bar{g}^x\right)^r$$

The above equality satisfies considerations on the correctness of $\mathcal{E}_\mu$. In Section 6 we also prove that $\mathcal{E}_\mu$ is a semantically secure cryptosystem.

$\mathcal{E}_\mu$ has a few interesting properties that are instrumental to achieving a secure and efficient implementation of our scheme:

- The decryption algorithm is non-interactive, meaning that it does not require the live participation of the entities that executed the $\mathcal{E}_\mu.\mathsf{DShare}$ algorithm.
- It mimics convergent encryption in that the decryption shares are deterministically dependent on the plaintext message; however, in contrast to plain convergent encryption, the cryptosystem provides semantic security as long as fewer than t decryption shares are collected.
- The cryptosystem can be reused for an arbitrary number of messages, i.e., the $\mathcal{E}_\mu.\mathsf{Setup}$ algorithm should only be executed once.

5.2 The Role of Trusted Third Parties

Our scheme uses two trusted components, namely, an *identity provider* (IdP) and an *indexing service* (IS). The main role of the IdP is to thwart sybil attacks by ensuring that users can sign in only once: we treat this as an orthogonal problem for which many effective solutions have been outlined [14]. The identity provider is also responsible for the execution of $\mathcal{E}_\mu$.Setup and for the distribution of the public key and a share of the private key to each user of the system. Execution of $\mathcal{E}_\mu$.Setup grants the identity provider knowledge of the private key of the system: we assume the IdP to be trusted not to leak it and not to use this knowledge to violate confidentiality of unpopular data. The assumption is a legitimate one as today's identity providers are entrusted by their users to comply with similar rules.

The main role of the second trusted third party, i.e., the indexing service, is to avoid leaking information about unpopular files to the storage provider through the index used to coalesce multiple uploads of the same file coming from different users (see Section 4.1), without which reclaiming space and saving network bandwidth through deduplication would be infeasible. The leakage is related to the requirement of finding a common indexing function that can be evaluated independently by different users whose only shared piece of information is the content of the file itself. As a result, the indexing function is usually a deterministic (albeit, often one-way) function of the file's content, which is leaked to the cloud provider. We introduce the indexing service to tackle this problem before deduplication takes place, i.e., when the file is still unpopular.

Recall from Section 4.1 that the indexing function $\mathcal{I}$ produces indexes $\mathcal{I}_F$ for every file F. This function can be implemented using cryptographic hash functions, but we avoid the usual notation with H to prevent it from being confused with the other hash functions used in $\mathcal{E}_\mu$. Informally, the indexing service receives requests from users about $\mathcal{I}_F$ and keeps count of the number of requests received for it from different users. As long as this number is below the popularity threshold, IS answers with a bitstring of the same length as the output of $\mathcal{I}$; this bitstring is obtained by invoking a PRF (with a random seed σ) on a concatenation of $\mathcal{I}_F$ and the identity of the requesting user. The domain of $\mathcal{I}$ and of the PRF is large enough to ensure that collisions happen with negligible probability. IS also keeps track of all such indexes. Whenever the popularity threshold is reached for a given file F, the indexing service reveals the set of indexes that were generated for it. More formally, the IS maintains an associative array $\mathsf{DB}_{IS}[\mathcal{I}_F]$ with two fields, $\mathsf{DB}_{IS}[\mathcal{I}_F]$.ctr and $\mathsf{DB}_{IS}[\mathcal{I}_F]$.idxes. The first is a counter initialized to zero, the second is an initially empty list. IS implements the **GetIdx** algorithm in Figure 3.

An important consequence of the choice of how $\mathcal{I}_{rnd}$ is computed is that repeated queries by the same user on the same target file will neither shift a given file's popularity nor reveal anything but a single index.

5.3 The Scheme

We are now ready to formally introduce our scheme, detailing the interactions between a set of users $\{\mathsf{U}_i\}_{i=0}^n$, a storage provider S and the two trusted entities, the identity provider IdP and the indexing service IS. S is modelled as described in Section 4.1; the database record contains an extra boolean field, $\mathsf{DB}[\mathcal{I}_F]$.popular, initialized to false for every new record.

Recall that $\mathcal{E}$ and $\mathcal{E}_c$ are a symmetric cryptosystem and a convergent symmetric cryptosystem, respectively (see Section 4.2); $\mathcal{E}_\mu$ is our convergent threshold cryptosystem. The scheme consists of the following distributed algorithms:

Init: IdP executes $\mathcal{E}_\mu$.Setup, publishes the public key system pk of the system. IdP keeps key

```
$\mathsf{U}_i$:                      $\mathcal{I}_F \leftarrow \mathcal{I}(F)$
$\mathsf{U}_i \longrightarrow \mathsf{IS}$:   $\mathcal{I}_F$
$\mathsf{IS}$:                       $\mathsf{I} \leftarrow \emptyset$
                          **if** $(\mathsf{DB}_{IS}[\mathcal{I}_F].\mathsf{ctr} > t)$
                             **return** $\mathcal{I}_F, \mathsf{I}$
                          $\mathcal{I}_{rnd} \leftarrow \mathtt{PRF}_\sigma(\mathsf{U}_i || \mathcal{I}_F)$
                          **if** $(\mathcal{I}_{rnd} \notin \mathsf{DB}_{IS}[\mathcal{I}_F].\mathsf{idxes})$
                             increment $\mathsf{DB}_{IS}[\mathcal{I}_F].\mathsf{ctr}$
                             add $\mathcal{I}_{rnd}$ to $\mathsf{DB}_{IS}[\mathcal{I}_F].\mathsf{idxes}$
                          **if** $(\mathsf{DB}_{IS}[\mathcal{I}_F].\mathsf{ctr} = t)$
                             $\mathsf{I} \leftarrow \mathsf{DB}_{IS}[\mathcal{I}_F].\mathsf{idxes}$
                          **return** $\mathcal{I}_{rnd}, \mathsf{I}$
```

Figure 3: The GetIdx algorithm.

```
$\mathsf{U}_i$:                      $K_c \leftarrow \mathcal{E}_c.\mathsf{K}(F)$;  $F_c \leftarrow \mathcal{E}_c.\mathsf{E}(K_c, F)$
                          $\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
$\mathsf{U}_i \longrightarrow \mathsf{IS}$:   $\mathcal{I}_{F_c}$
$\mathsf{U}_i \longleftarrow \mathsf{IS}$:   $\langle \mathsf{I}, \mathcal{I}_{ret} \rangle \leftarrow \mathsf{GetIdx}(\mathcal{I}_{F_c})$
$\mathsf{U}_i$:                      **if**$(\mathcal{I}_{ret} = \mathcal{I}_{F_c})$ execute Upload.Popular
                          **else if**$(\mathsf{I} = \emptyset)$ execute Upload.Unpopular
                          **else**
                             execute Upload.Unpopular
                             execute Upload.Reclaim
```

Figure 4: The Upload algorithm.

shares $\{\mathsf{sk}_i\}_{i=1}^{i=n}$ secret.

Join: whenever a user U_i wants to join the system, she contacts IdP. IdP verifies U_i's identity; upon successful verification, it issues the credentials U_i will need to authenticate to S and a secret key share sk_i.

Upload (Fig. 4): this algorithm describes the interactions taking place between a user U_i, the storage server S and the indexing service IS whenever U_i requests upload of a file F. At first, U_i uses convergent encryption to create ciphertext F_c; U_i then interacts with IS to obtain an index $\mathcal{I}_{ret}$ to use for the interaction with S and a (possibly empty) list of indexes used by other users when uploading the same file. Based on what IS returns, U_i proceeds with the execution of different sub-algorithms, detailed below:

– **Upload.Unpopular** (Fig. 5): this algorithm captures the interaction between U_i and S if F is not (yet) popular. In this case, $\mathcal{I}_{ret}$ is a random index. The user uploads a blob containing two ciphertexts, obtained with $\mathcal{E}$ and $\mathcal{E}_\mu$, respectively. The first ciphertext allows U_i to retrieve and decrypt the file if it never becomes popular. The second gives S the ability to remove the threshold encryption layer and perform deduplication if the file becomes popular[2]. U_i deletes

[2]We have chosen to formalize this approach for the sake of readability. In practice, one would adopt a solution in which the file is encrypted only once with K; this key – and not the entire file – is in turn encrypted with a slightly modified version of $\mathcal{E}_\mu$ that allows $H_1(F_c)$ to be used as the H_1-hash for computing ciphertext and decryption shares for K. This approach would require uploading and storing a single encrypted copy of the file and not two as described above. We adopt this approach in our prototype implementation in Section 7

F, replacing it with a stub containing the two indexes, $\mathcal{I}_{ret}$, $\mathcal{I}_{F_c}$, and the two keys K and K_c.

– **Upload.Reclaim** (Fig. 6): this algorithm is executed exactly once for every popular file whenever U_i's upload of F reaches the popularity threshold. The user sends to S the list of indexes I received from IS. S collects the decryption shares from each uploaded blob. It is then able to decrypt each uploaded instance of c_μ and can trigger the execution of Put, to store the outcome of the decryption as $\mathsf{DB}\left[\mathcal{I}_{F_c}\right].\mathsf{data}$. Note that, because of the nature of convergent encryption, all decrypted instances are identical, hence deduplication happens automatically[3]. Finally, S can remove all previously uploaded record entries, thus effectively reclaiming the space that was previously used.

– **Upload.Popular** (Fig. 7): this algorithm captures the interaction between U_i and S if F is already popular; note that in this case, $\mathcal{I}_{ret} = \mathcal{I}_{F_c}$. In this case, the user is not expected to upload the content of the file as it has already been declared popular. U_i deletes F, replacing it with a stub containing the index $\mathcal{I}_{F_c}$ and of the key K_c.

Dowload: whenever user U_i wants to retrieve a previously uploaded file, it reads the tuple used to replace the content of F during the execution of the Upload algorithm. It first attempts to issue a Get request on S, supplying $\mathcal{I}_{ret}$ as index. If the operation succeeds, it proceeds to decrypt the received content with $\mathcal{E}.\mathsf{D}$, using key K, and returns the output of the decryption. Otherwise, it issues a second Get request, supplying $\mathcal{I}_{F_c}$ as index; then it invokes $\mathcal{E}_c.\mathsf{D}$ on the received content, using K_c as decryption key, and outputs the decrypted plaintext.

6 Security Analysis

In this section we formally analyze the security of the $\mathcal{E}_\mu$ cryptosystem. Subsequently, we argue, albeit only informally, that the security requirements outlined in Section 3.1 are met by our scheme as a whole. A formal analysis based on the UC framework [12] is left for future work.

6.1 Security Analysis of $\mathcal{E}_\mu$

In this section we will define and analyze semantic security for $\mathcal{E}_\mu$. The security definition we adopt makes use of a straightforward adaptation of the IND-CPA experiment, henceforth referred to as IND$_\mu$-CPA. Additionally, we introduce the concept of *unlinkability of decryption shares* and prove that $\mathcal{E}_\mu$ provides this property: informally, this property assures that an adversary cannot link together decryption shares as having been generated for the same message, as long as less than t of them are available. We will refer to the experiment used for the proof of this property

```
$\mathsf{U}_i$:              $K \leftarrow \mathcal{E}.\mathsf{K}()$;   $c \leftarrow \mathcal{E}.\mathsf{E}(K, F)$
                  $c_\mu \leftarrow \mathcal{E}_\mu.\mathsf{Encrypt}(\mathsf{pk}, F_c)$
                  $\mathsf{ds_i} \leftarrow \mathcal{E}_\mu.\mathsf{DShare}(\mathsf{sk_i}, F_c)$
                  $F' \leftarrow \langle c, c_\mu, \mathsf{ds_i} \rangle$
$\mathsf{U}_i \longrightarrow \mathsf{S}$:  $\mathcal{I}_{ret}, F'$
$\mathsf{S}$:               if$(\neg \mathsf{DB}\left[\mathcal{I}_{ret}\right].\mathsf{popular})$ execute $\mathsf{Put}(\mathcal{I}_{F_c}, \mathsf{U}_i, F')$
                  else signal an error and exit
$\mathsf{U}_i$:              $F \leftarrow \langle K, K_c, \mathcal{I}_{ret}, \mathcal{I}_{F_c} \rangle$
```

Figure 5: The Upload.Unpopular algorithm.

[3] S could perform an additional check and raise an error if not all decryptions of c_μ are pairwise identical.

```
$\mathsf{U}_i \longrightarrow \mathsf{S}$:  $\mathsf{I}$
$\mathsf{S}$:        $\mathsf{DS} \leftarrow \{\mathsf{ds} : \langle \mathsf{c}, \mathsf{c}_\mu, \mathsf{ds}\rangle \leftarrow \mathsf{DB}[\mathcal{I}].\mathsf{data},\ \mathcal{I} \in \mathsf{I}\}$
             foreach($\mathcal{I}_i \in \mathsf{I}$)
                parse $\mathsf{DB}[\mathcal{I}_F].\mathsf{data}$ as $\langle c, c_\mu, \mathsf{ds_i}\rangle$
                $F_c \leftarrow \mathcal{E}_\mu.\mathsf{Decrypt}(c_\mu, \mathsf{DS})$
                $\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
                $\mathsf{U}_i \leftarrow \mathsf{DB}[\mathcal{I}_i].\mathsf{users}$
                execute $\mathsf{Put}(\mathcal{I}_{F_c}, \mathsf{U}_i, F_c)$
             $\mathsf{DB}[\mathcal{I}_{F_c}].\mathsf{popular} \leftarrow \mathsf{true}$
             delete all records indexed by $\mathsf{I}$
```

Figure 6: The Upload.Reclaim algorithm

as DS_μ-IND. Both security experiments require the adversary to declare upfront the set of users it wishes to corrupt before it can observe any output produced by the scheme and adaptively choose in what order to invoke the corrupt oracle over those. Hence, our security notion is not the strongest possible, similarly to "selective security" for attribute-based encryption [26, 18].

6.1.1 Unlinkability of Decryption Shares

Informally, in DS_μ-IND, the adversary is given access to two hash function oracles $\mathcal{O}_{\mathsf{H}_1}$, and $\mathcal{O}_{\mathsf{H}_2}$; the adversary can also *corrupt* an arbitrary number $n_\mathcal{A} < t - 1$ of pre-declared users, and obtains their secret keys through the $\mathcal{O}_{\mathsf{Corrupt}}$ oracle. Finally, the adversary can access a *decryption share* oracle $\mathcal{O}_{\mathsf{DShare}}$, submitting a message m of her choice and a non-corrupted user identity U_i; for each message that appears to $\mathcal{O}_{\mathsf{DShare}}$-queries, the challenger chooses at random (based on a fair coin flip) whether to respond with a properly constucted decryption share that corresponds to message m and secret key share sk_i as defined in $\mathcal{E}_\mu$, or with a random bitstring of the same length (e.g., when $\mathsf{b}_{m_*} = 0$). At the end of the game, the adversary declares a message m_*, for which up to $t - n_\mathcal{A} - 1$ decryption share queries for distinct user identities have been submitted. The adversary outputs a bit b'_{m_*} and wins the game if $\mathsf{b}'_{m_*} = \mathsf{b}_{m_*}$. $\mathcal{E}_\mu$ is said to satisfy unlinkability of decryption shares, if no polynomial time adversary can win the game with a non-negligible advantage.

Formally, unlinkability of decryption shares is defined using the following experiment (DS_μ-IND) between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, given a security parameter κ:

- **Setup Phase** $\mathcal{C}$ executes the Setup algorithm with κ, and generates a set of user identities $\mathsf{U} = \{\mathsf{U_i}\}_{\mathsf{i=1}}^{\mathsf{i=n}}$. Further, $\mathcal{C}$ gives pk to $\mathcal{A}$ and keeps $\{\mathsf{sk}_i\}_{i=1}^{i=n}$ secret. At this point, $\mathcal{A}$ declares the list $\mathsf{U}_\mathcal{A}$ of $|\mathsf{U}_\mathcal{A}| = n_\mathcal{A} < t - 1$ identities of users that will later on be subject to $\mathcal{O}_{\mathsf{Corrupt}}$ calls.

- **Access to Oracles** Throughout the game, the adversary can invoke oracles for the hash

```
$\mathsf{U}_i \longrightarrow \mathsf{S}$:  $\mathcal{I}_{F_c}$
$\mathsf{S}$:        if($\mathsf{DB}[\mathcal{I}_{F_c}].\mathsf{popular}$) execute $\mathsf{Put}(\mathcal{I}_{F_c}, \mathsf{U}_i)$
             else signal an error and exit
$\mathsf{U}_i$:       $F \leftarrow \langle K_c, \mathcal{I}_{F_c}\rangle$
```

Figure 7: The Upload.Popular algorithm

functions H_1 and H_2. Additionally, the adversary can invoke the corrupt oracle $\mathcal{O}_{\mathsf{Corrupt}}$ and receive the secret key share that corresponds to any user $\mathsf{U}_i \in \mathsf{U}_{\mathcal{A}}$. Finally, $\mathcal{A}$ can invoke the decryption share oracle $\mathcal{O}_{\mathsf{DShare}}$ to request a decryption share that corresponds to a specific message, say m, and the key share of a non-corrupted user, say $\mathsf{U}_i \notin \mathsf{U}_{\mathcal{A}}$. More specifically, for each message m that appears in $\mathcal{O}_{\mathsf{DShare}}$-queries, the challenger chooses at random (based on a fair coin flip b_m) whether to respond to $\mathcal{O}_{\mathsf{DShare}}$-queries for m with decryption shares constructed as defined by the protocol, or with random bitstrings of the same length. Let $\mathsf{ds}_{i,m}$ denote the response of $\mathcal{O}_{\mathsf{DShare}}$-query for m and U_i. $\mathsf{b}_m = 1$ correspond to the case, where responses in $\mathcal{O}_{\mathsf{DShare}}$-queries for m are properly constructed decryption shares.

- **Challenge Phase** $\mathcal{A}$ chooses a target message m_*. The adversary is limited in the choice of the challenge message as follows: m_* must not have been the subject of more than $t - n_{\mathcal{A}} - 1$ $\mathcal{O}_{\mathsf{DShare}}$ queries for distinct user identities. At the challenge time, if the limit of $t - n_{\mathcal{A}} - 1$ has not been reached, the adversary is allowed to request for more decryption shares for as long as the aforementioned condition holds. Recall that $\mathcal{C}$ responds to challenge $\mathcal{O}_{\mathsf{DShare}}$-queries based on b_{m_*}.

- **Guess** $\mathcal{A}$ outputs b'_{m_*}, that represents her guess for b_{m_*}. The adversary wins the game, if $\mathsf{b}_{m_*} = \mathsf{b}'_{m_*}$.

6.1.2 Semantic Security

Informally, in IND_{μ}-CPA, the adversary is given access to the two hash function oracles $\mathcal{O}_{\mathsf{H}_1}$, and $\mathcal{O}_{\mathsf{H}_2}$. Also, the adversary can *corrupt* an arbitrary number $n_{\mathcal{A}} < t-1$ of pre-declared users, and obtain their secret keys through the $\mathcal{O}_{\mathsf{Corrupt}}$ oracle. Finally, the adversary can access a *decryption share* oracle $\mathcal{O}_{\mathsf{DShare}}$; here the adversary submits a message m of her choice and a user identity U_i to receive a decryption share $\mathsf{ds}_{m,i}$ that corresponds to m and the secret key share sk_i of (non-corrupted) user U_i. At the end of the game, the adversary outputs a message m_*; the challenger flips a fair coin b, and based on its outcome, it returns to $\mathcal{A}$ the encryption of either m_* or of another random bitstring of the same length. The adversary outputs a bit b' and wins the game if $\mathsf{b}' = \mathsf{b}$. $\mathcal{E}_{\mu}$ is said to be semantically secure if no polynomial time adversary can win the game with a non-negligible advantage.

Formally, the security of $\mathcal{E}_{\mu}$ is defined using the following experiment (IND_{μ}-CPA) between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, given a security parameter κ:

- **Setup Phase** is the same as in DS_{μ}-IND.

- **Access to Oracles** Throughout the game, the adversary can invoke oracles for the hash functions H_1 and H_2. Additionally, the adversary can invoke the corrupt oracle $\mathcal{O}_{\mathsf{Corrupt}}$ and receive the secret key share that corresponds to each user $\mathsf{U}_i \in \mathsf{U}_{\mathcal{A}}$. Finally, $\mathcal{A}$ can invoke the decryption share oracle $\mathcal{O}_{\mathsf{DShare}}$ and receive a decryption share $\mathsf{ds}_{m,i}$ that corresponds to m and the key share of U_i.

- **Challenge Phase** $\mathcal{A}$ picks the challenge message m_* and sends it to $\mathcal{C}$; the adversary is limited in her choice of the challenge message as follows: the sum of *distinct* user identities supplied to $\mathcal{O}_{\mathsf{DShare}}$ together with the challenge message cannot be greater than $t - n_{\mathcal{A}} - 1$. $\mathcal{C}$ chooses at random (based on a coin flip b) whether to return the encryption of m_* ($\mathsf{b} = 1$), or of another random string of the same length ($\mathsf{b} = 0$); let c_* be the resulting ciphertext; this is returned to $\mathcal{A}$.

- **Guess** $\mathcal{A}$ outputs b', that represents her guess for b.The adversary wins the game, if $\mathsf{b} = \mathsf{b}'$.

The following lemmas show that unlinkability of decryption shares is guaranteed in $\mathcal{E}_\mu$, and that $\mathcal{E}_\mu$ is a semantically secure cryptosystem, as long as the Symmetric External Diffie-Hellman (SXDH) problem is intractable. In short, SXDH assumes two groups of prime order q, $\mathbb{G}_1$, and $\mathbb{G}_2$, such that there is not an efficiently computable distortion map between the two; a bilinear group $\mathbb{G}_T$, and an efficient, non-degenerate bilinear map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. In this setting, the SXDH assumption states that the Decisional Diffie-Hellman (DDH) holds in both $\mathbb{G}_1$, and $\mathbb{G}_2$, and that the bilinear decisional Diffie-Hellman (BDDH) holds given the existence of the bilinear map $\hat{e}$ [8].

Lemma 1. *Let H_1, and H_2 be random oracles. If a DS_μ-IND adversary $\mathcal{A}$ has a non-negligible advantage*

$$\mathsf{Adv}^{\mathcal{A}}_{\mathsf{DS}_\mu\text{-}\mathsf{IND}} := \Pr[\mathsf{b}'_{m_*} \leftarrow \mathcal{A}(m_*, \mathsf{ds}_{*,m_*}) : \mathsf{b}'_{m_*} = \mathsf{b}_{m_*}] - \frac{1}{2}$$

then a probabilistic, polynomial time algorithm $\mathcal{C}$ can create an environment where it uses $\mathcal{A}$'s advantage to solve any given instance of the SXDH problem.

Proof. We define the challenger $\mathcal{C}$ as follows. $\mathcal{C}$ is given an SXDH context $\mathbb{G}'_1, \mathbb{G}'_2, \mathbb{G}'_T, \hat{e}'$ and an instance of the DDH problem $\langle \mathbb{G}'_1, g', A = (g')^a, B = (g')^b, W \rangle$ in $\mathbb{G}_1$'. The algorithm $\mathcal{C}$ simulates an environment in which $\mathcal{A}$ operates, using its advantage in the game DS_μ-IND to decide whether $W = g'^{ab}$. Algorithm $\mathcal{C}$ works by interacting with $\mathcal{A}$ in the DS_μ-IND game as follows:

- **Setup Phase** $\mathcal{C}$ sets $\mathbb{G}_1 \leftarrow \mathbb{G}'_1$, $\mathbb{G}_2 \leftarrow \mathbb{G}'_2$, $\mathbb{G}_T \leftarrow \mathbb{G}'_T$, $\hat{e} = \hat{e}'$, $g \leftarrow g'$; picks a random generator $\bar{g}$ of $\mathbb{G}_2$ and sets $\bar{g}_{pub} = (\bar{g})^{\mathsf{sk}}$, where $\mathsf{sk} \leftarrow_R \mathbb{Z}^*_q$. $\mathcal{C}$ also generates the set of user identities $\mathsf{U} = \{\mathsf{U}_i\}^{i=n}_{i=1}$. The public key $\mathsf{pk} = \{q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T\, \hat{e}, \mathcal{O}_{\mathsf{H}_1}, \mathcal{O}_{\mathsf{H}_2}, \bar{g}, \bar{g}_{pub}\}$ and U are then forwarded to the adversary. At this point, $\mathcal{A}$ declares the list $\mathsf{U}_{\mathcal{A}}$ of $n_{\mathcal{A}} < t - 1$ user identities that will later on be subject to $\mathcal{O}_{\mathsf{Corrupt}}$ calls. Without loss of generality, let us assume that $\mathsf{U}_{\mathcal{A}} = \{\mathsf{U}_i\}^{n_{\mathcal{A}}}_{i=1}$.

  To generate key-shares $\{\mathsf{sk}_i\}^{i=n}_{i=1}$, $\mathcal{C}$ constructs a $t-1$-degree Lagrange polynomial $\mathrm{P}()$ assuming interpolation points

  $$\mathrm{I}_{\mathrm{P}} = \{(i, y_i)\}^{i=t-1}_{i=0},$$

  where $y_0 \leftarrow \mathsf{sk}$, $y_i \leftarrow_R \mathbb{Z}^*_q$, for $i \in [1, t-2]$, and $y_{t-1} \leftarrow a$. Notice that a is not known to $\mathcal{C}$. She then sets the key-shares for corrupted users to be $\mathsf{sk}_i \leftarrow y_i$ for $i \in [1, n_{\mathcal{A}}]$.

- **Access to Oracles** $\mathcal{C}$ simulates oracles $\mathcal{O}_{\mathsf{H}_1}$, $\mathcal{O}_{\mathsf{H}_2}$, $\mathcal{O}_{\mathsf{Corrupt}}$ and $\mathcal{O}_{\mathsf{DShare}}$ as follows:

  - $\mathcal{O}_{\mathsf{H}_1}$: to respond to $\mathcal{O}_{\mathsf{H}_1}$ queries $\mathcal{C}$ maintains a list of tuples $\{\mathsf{H}_1, v, h_v, r_v, \mathrm{c}_v\}$ as explained below. We refer to this list as $\mathcal{O}_{\mathsf{H}_1}$ list, and is initially empty. When $\mathcal{A}$ submits an $\mathcal{O}_{\mathsf{H}_1}$ query for v, $\mathcal{C}$ checks if v already appears in the $\mathcal{O}_{\mathsf{H}_1}$ list in a tuple $\{v, h_v, r_v, \mathrm{c}_v\}$. If so, $\mathcal{C}$ responds with $\mathsf{H}_1(v) = h_v$. Otherwise, $\mathcal{C}$ picks a random $r_v \in \mathbb{Z}^*_q$, and flips a coin c_v; c_v flips to $'0'$ with probability δ for some δ to be determined later. If c_v equals $'0'$, $\mathcal{C}$ responds $\mathsf{H}_1(v) = h_v = g^{r_v}$ and stores $\{v, h_v, r_v, \mathrm{c}_v\}$; otherwise, she returns $\mathsf{H}_1(v) = h_v = B^{r_v}$ and stores $\{v, h_v, r_v, \mathrm{c}_v\}$.
  - $\mathcal{O}_{\mathsf{H}_2}$: The challenger $\mathcal{C}$ responds to a newly submitted $\mathcal{O}_{\mathsf{H}_2}$ query for v with a randomly chosen $h_v \in \mathbb{G}_T$. To be consistent in her $\mathcal{O}_{\mathsf{H}_2}$ responses, $\mathcal{C}$ maintains the history of her responses in her local memory.

- $\mathcal{O}_{\mathsf{Corrupt}}$: $\mathcal{C}$ responds to a $\mathcal{O}_{\mathsf{Corrupt}}$ query involving user $\mathsf{U}_i \in \mathsf{U}_{\mathcal{A}}$, by returning the coordinate y_i chosen in the Setup phase.
- $\mathcal{O}_{\mathsf{DShare}}$: simulation of $\mathcal{O}_{\mathsf{DShare}}$ is performed as follows. As before, $\mathcal{C}$ keeps track of the submitted $\mathcal{O}_{\mathsf{DShare}}$ queries in her local memory. Let $\langle m, \mathsf{U}_i \rangle$ be a decryption query submitted for message m and user identity U_i. If there is no entry in H_1-list for m, then $\mathcal{C}$ runs the $\mathcal{O}_{\mathsf{H}_1}$ algorithm for m. Let $\{m, h_m, r_m, \mathrm{c}_m\}$ be the $\mathcal{O}_{\mathsf{H}_1}$ entry in $\mathcal{C}$'s local memory for message m. Let $\mathrm{I_P}' \leftarrow \mathrm{I_P} \setminus (t-1, y_{t-1})$. $\mathcal{C}$ responds with

$$\mathsf{ds}_{m,i} = \left(g^{\sum\limits_{(j,y_j)\in \mathrm{I_P}'} y_j \lambda_{i,j}^{\mathrm{I_P}'}} X^{\lambda_{i,t-1}^{\mathrm{I_P}'}} \right)^{r_m}$$

  where $X \leftarrow A$ iff $\mathrm{c}_m = 0$, and $X \leftarrow W$ iff $\mathrm{c}_m = 1$. In both cases, $\mathcal{C}$ keeps a record of her response in her local memory.

- **Challenge Phase** In the challenge phase, $\mathcal{A}$ selects the challenge message m_*. Let the corresponding entry in the $\mathcal{O}_{\mathsf{H}_1}$ list be $\{m_*, h_{m_*}, r_{m_*}, \mathrm{c}_{m_*}\}$. If $\mathrm{c}_{m_*} = 0$, then $\mathcal{C}$ aborts. As discussed before, from now on $\mathcal{A}$ can perform $\mathcal{O}_{\mathsf{DShare}}$-queries for m_* so as the submitted $\mathcal{O}_{\mathsf{DShare}}$-queries for m_* from the beginning of the game do not correspond to more than $t - n_{\mathcal{A}} - 1$ distinct user identities.
- **Guess** $\mathcal{A}$ outputs one bit b'_{m_*} representing the guess for b_{m_*}. $\mathcal{C}$ responds positively to the DDH challenger if $\mathsf{b}'_{m_*} = 0$, and negatively otherwise.

It is easy to see, that if $\mathcal{A}$'s answer is $'0'$, it means that the $\mathcal{O}_{\mathsf{DShare}}$ responses for m_* constitute properly structured decryption shares for m_*. However this can only be if $W = g^{ab}$ and $\mathcal{C}$ can give a positive answer to the SXDH challenger.

A detailed analysis shows that if $\mathrm{c}_{m_*} = 1$ and $\mathrm{c}_m = 0$ for all other queries to $\mathcal{O}_{\mathsf{H}_1}$ such that $m \neq m_*$, then the execution environment is indistinguishable from the actual game DS_μ-IND. This happens with probability

$$\Pr[\mathrm{c}_{m_*} = 1 \ \wedge \ (\forall m \neq m_* : \mathrm{c}_m = 0)] = \delta(1-\delta)^{\mathcal{Q}_{H_1}-1}$$

where $\mathcal{Q}_{H_1}$ is the number of different $\mathcal{O}_{\mathsf{H}_1}$ queries. By setting $\delta \approx \frac{1}{\mathcal{Q}_{H_1}-1}$ we know that the above probability is greater than $\frac{1}{e\cdot(\mathcal{Q}_{H_1}-1)}$. In conclusion, we can bound the probability of success of the adversary $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{DS}_\mu\text{-}\mathsf{IND}}$ as $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{DS}_\mu\text{-}\mathsf{IND}} \leq e \cdot (\mathcal{Q}_{H_1} - 1) \cdot \mathsf{Adv}^{\mathcal{C}}_{\mathsf{SXDH}}$. □

Lemma 2. *Let* H_1*, and* H_2 *be random oracles. If an* IND_μ*-CPA adversary* $\mathcal{A}$ *has a non-negligible advantage*

$$\mathsf{Adv}^{\mathcal{A}}_{\mathit{IND}_\mu\text{-}\mathit{CPA}} := \mathrm{Prob}[\mathsf{b}' \leftarrow \mathcal{A}(c_*) : \mathsf{b} = \mathsf{b}'] - \frac{1}{2}$$

then a probabilistic, polynomial time algorithm $\mathcal{C}$ *can create an environment where it uses* $\mathcal{A}$*'s advantage to solve any given instance of the SXDH problem.*

Proof. We define the challenger $\mathcal{C}$ as follows. $\mathcal{C}$ is given an instance $\langle q', \mathbb{G}'_1, \mathbb{G}'_2, \mathbb{G}'_T, \hat{e}', g', \bar{g}', A = (g')^a, B = (g')^b, C = (g')^c, \bar{A} = (\bar{g}')^a, \bar{B} = (\bar{g}')^b, \bar{C} = (\bar{g}')^c, W\rangle$ of the SXDH problem and wishes to use $\mathcal{A}$ to decide if $W = \hat{e}\,(g', \bar{g}')^{abc}$. The algorithm $\mathcal{C}$ simulates an environment in which $\mathcal{A}$ operates, using its advantage in the game IND_μ-CPA to help compute the solution to the BDDH problem as detailed before. Algorithm $\mathcal{C}$ works by interacting with $\mathcal{A}$ in an IND_μ-CPA game as follows:

- **Setup Phase** $\mathcal{C}$ sets $q \leftarrow q'$, $\mathbb{G}_1 \leftarrow \mathbb{G}_1'$, $\mathbb{G}_2 \leftarrow \mathbb{G}_2'$, $\mathbb{G}_T \leftarrow \mathbb{G}_T'$, $\hat{e} = \hat{e}'$, $g \leftarrow g'$, $\bar{g} \leftarrow \bar{g}'$, $\bar{g}_{pub} = \bar{A}$. Notice that the secret key $\mathsf{sk} = a$ is not known to $\mathcal{C}$. $\mathcal{C}$ also generates the list of user identities U. $\mathcal{C}$ sends $\mathsf{pk} = \{q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T\, \hat{e}, \mathcal{O}_{\mathsf{H}_1}, \mathcal{O}_{\mathsf{H}_2}, \bar{g}, \bar{g}_{pub}\}$ to $\mathcal{A}$. At this point, $\mathcal{A}$ declares the list $\mathsf{U}_{\mathcal{A}}$ of $n_{\mathcal{A}} < t-1$ identities of users that will later on be subject to $\mathcal{O}_{\mathsf{Corrupt}}$ calls. Without loss of generality, let us assume that $\mathsf{U}_{\mathcal{A}} = \{\mathsf{U}_i\}_{i=1}^{n_{\mathcal{A}}}$. To generate key-shares $\{\mathsf{sk}_i\}_{i=1}^{n}$, $\mathcal{C}$ picks a $t-1$ degree Lagrange polynomial $\mathrm{P}()$ assuming interpolation points

$$\mathrm{I_P} = \left\{(0, a) \;\cup\; \{(i, y_i)\}_{i=1}^{i=t-1}\right\},$$

where $y_i \leftarrow_R \mathbb{Z}_q^*$. She then sets the key-shares for corrupted users to be $\mathsf{sk}_i \leftarrow y_i$ for $i \in [1, n_{\mathcal{A}}]$.

- **Access to Oracles** $\mathcal{C}$ simulates oracles $\mathcal{O}_{\mathsf{H}_1}$, $\mathcal{O}_{\mathsf{H}_2}$, $\mathcal{O}_{\mathsf{Corrupt}}$ and $\mathcal{O}_{\mathsf{DShare}}$ as follows:

  - $\mathcal{O}_{\mathsf{H}_1}$, $\mathcal{O}_{\mathsf{H}_2}$, $\mathcal{O}_{\mathsf{Corrupt}}$: $\mathcal{C}$ responds to these queries as described in the DS_{μ}-IND experiment.
  - $\mathcal{O}_{\mathsf{DShare}}$: simulation of $\mathcal{O}_{\mathsf{DShare}}$ is performed as follows. $\mathcal{C}$ keeps track of the submitted $\mathcal{O}_{\mathsf{DShare}}$ queries in her local memory. Let $\langle m, \mathsf{U}_i\rangle$ be a decryption query submitted for message m and user identity U_i. If there is no entry in H_1-list for m, then $\mathcal{C}$ runs the $\mathcal{O}_{\mathsf{H}_1}$ algorithm for m. Let $\{m, h_m, r_m, \mathrm{c}_m\}$ be the $\mathcal{O}_{\mathsf{H}_1}$ entry in $\mathcal{C}$'s local memory for message m. If $\mathrm{c}_m = 1$, and $\mathcal{A}$ has already submitted $t - n_{\mathcal{A}} - 1$ queries for m, $\mathcal{C}$ reports failure and terminates. If the limit of $t - n_{\mathcal{A}} - 1$ queries has not yet been reached, $\mathcal{C}$ responds with a random $\mathsf{ds}_{m,i} \in \mathbb{G}_1$ and keeps a record for it. This step is legitimate as Lemma 1 establishes that the adversary is not able to distinguish properly formed decryption shares as long as less than t are available. Let $\mathrm{I_P}' \leftarrow \mathrm{I_P} \setminus (0, a)$. If $\mathrm{c}_m = 0$, $\mathcal{C}$ responds with

$$\mathsf{ds}_{m,i} = \left(g^{\sum\limits_{(j,y_j)\in \mathrm{I_P}'} y_j \lambda_{i,j}^{\mathrm{I_P}'}} A^{\lambda_{i,0}^{\mathrm{I_P}'}} \right)^{r_m}$$

- **Challenge Phase** in the challenge phase, $\mathcal{A}$ submits the challenge message m_* to $\mathcal{C}$. Recall that $\mathcal{A}$ has not submitted $\mathcal{O}_{\mathsf{DShare}}$ queries for the challenge message with more than $t - n_{\mathcal{A}} - 1$ distinct user identities. Next, $\mathcal{C}$ runs the algorithm for responding to $\mathcal{O}_{\mathsf{H}_1}$ queries for m_* to recover the entry from the $\mathcal{O}_{\mathsf{H}_1}$ list. Let the entry be $\{m_*, h_{m_*}, r_{m_*}, \mathrm{c}_{m_*}\}$. If $\mathrm{c}_{m_*} = 0$, $\mathcal{C}$ reports failure and aborts. Otherwise, $\mathcal{C}$ computes $e_* \leftarrow W^{r_{m_*}}$, sets $c_* \leftarrow \langle m_* \oplus \mathsf{H}_2(e_*), \bar{C}\rangle$ and returns c_* to the $\mathcal{A}$.

- **Guess** $\mathcal{A}$ outputs one bit b' representing the guess for b. $\mathcal{C}$ provides the same answer for its SXDH challenge.

If $\mathcal{A}$'s answer is $\mathsf{b}' = 1$, it means that she has recognized the ciphertext c_* as the encryption of m_*; $\mathcal{C}$ can then give the positive answer to her SXDH challenge. Indeed

$$W^{r_{m_*}} = \hat{e}\,(g, \bar{g})^{abcr_{m_*}} = \hat{e}\,((B^{r_{m_*}})^a, \bar{g}^c) = \hat{e}\left(H_1(m_*)^{\mathsf{sk}}, \bar{C}\right)$$

A detailed analysis shows that if $\mathrm{c}_{m_*} = 1$ and $\mathrm{c}_m = 0$ for all other queries to $\mathcal{O}_{\mathsf{H}_1}$ such that $m \neq m_*$, then the execution environment is indistinguishable from the actual game IND_{μ}-CPA. This happens with probability

$$\Pr[\mathrm{c}_{m_*} = 1 \;\wedge\; (\forall m \neq m_* : \mathrm{c}_m = 0)] = \delta(1-\delta)^{\mathcal{Q}_{H_1}-1}$$

where $\mathcal{Q}_{H_1}$ is the number of different $\mathcal{O}_{\mathsf{H}_1}$ queries. By setting $\delta \approx \frac{1}{\mathcal{Q}_{H_1}-1}$ we know that the above probability is greater than $\frac{1}{e\cdot(\mathcal{Q}_{H_1}-1)}$. In conclusion, we can bound the probability of success of the adversary $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_\mu\text{-}\mathsf{CPA}}$ as $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_\mu\text{-}\mathsf{CPA}} \leq e \cdot (\mathcal{Q}_{H_1} - 1) \cdot \mathsf{Adv}^{\mathcal{C}}_{\mathsf{SXDH}}$. □

6.2 Security Analysis of the Scheme

In this Section we will study the security of the scheme as a whole. A formal analysis under the UC framework is not presented here and is left for future work. We instead present informal arguments, supported by the proofs shown in the previous Section and the assumptions on our trusted third parties, showing how the security requirements highlighted in Section 3 are met.

Let us briefly recall that the adversary in our scheme is represented by a set of users colluding with the cloud storage provider. The objective of the adversary is to violate the confidentiality of data uploaded by legitimate users: in particular, the objective for unpopular data is semantic security, whereas it is conventional convergent security for popular data. We assume that the adversary controls a set of $n_{\mathcal{A}}$ users $\{\mathsf{U}_i\}_{i=1}^{n_{\mathcal{A}}}$. Let us also recall the popularity threshold p_{lim} which represents the smallest number of distinct, legitimate users that are required to upload a given file F for that file to be declared popular. We finally recall that the threshold t of $\mathcal{E}_\mu$– also used by the indexing service – is set to be $t \geq p_{lim} + n_{\mathcal{A}}$. This implies that the adversary cannot use its control over $n_{\mathcal{A}}$ users to subvert the popularity mechanism and force a non-popular file of its choice to become popular. This fact stems from the security of $\mathcal{E}_\mu$ and from the way the indexing service is implemented. As a consequence, transition of a file between unpopular and popular is governed by legitimate users.

The adversary can access two conduits to retrieve information on user data: i) the indexing service (IS) and ii) the records stored by the storage provider in DB.

The indexing service cannot be used by the attacker to retrieve any useful information on popular files; indeed the adversary already possesses $\mathcal{I}_{F_c}$ for all popular files and consequently, queries to IS on input the index $\mathcal{I}_{F_c}$ do not reveal any additional information other than the notion that the file is popular. As for unpopular files, the adversary can only retrieve indexes computed using a PRF with a random secret seed. Nothing can be inferred from those, as guaranteed by the security of the PRF. Note also that repeated queries of a single user on a given file always only yield the same index and do not influence popularity.

Let us now consider what the adversary can learn from the content of the storage backend, modeled by DB. The indexing keys are either random strings (for unpopular files) or the output of a deterministic, one-way function $\mathcal{I}$ on the convergent ciphertext (for popular files). In the first case, it is trivial to show how nothing can be learned. In the latter case, the adversary may formulate a guess F' for the content of a given file, compute $\mathcal{I}_{F'}$ and compare it with the index. However this process does not yield any additional information that can help break standard convergent security: indeed the same can be done on the convergent ciphertext. As for the data content of DB, it is always in either of two forms: $\langle c, c_\mu, \mathsf{ds_i} \rangle$ for unpopular files and F_c for popular files. It is easy to see that in both cases, the length of the plaintext is leaked but we argue this does not constitute a security breach. The case of popular file is very simple to analyze given that security claims stem directly from the security of convergent encryption. As for unpopular files, c is the ciphertext produced by a semantically secure cryptosystem and by definition does not leak any information about the corresponding ciphertext. c_μ and $\mathsf{ds_i}$ represent the ciphertext and the decryption share produced by $\mathcal{E}_\mu$, respectively. Assuming that t is set as described above, the adversary cannot be in possession of t decryption shares. Consequently, Lemma 2 guarantees that no information on the corresponding plaintext can be learned.

7 Performance Evaluation

In this Section we evaluate the performance of our scheme with respect to the computation overhead at file encryption and decryption time, and with respect to storage optimization. Subsequently we compare the performance of our scheme with the performance of other systems that preserve data confidentiality.

7.1 Prototype implementation

Our prototype consists of a client program, that performs the file encryption and decryption, and a server program that sets up the system parameters and performs the appropriate crypro-graphic operations when a file becomes popular. We implemented both sides in C, and used the libraries provided by [22, 9, 5, 6] for pairings, and other cryptographic operations. Both programs were tested on an Intel Xeon X3323 machine with 4 CPU cores 2.5 GHz, and 8GB of RAM running CentOS 5.4. Furthermore, for our experiments we used SHA-256 for hashing and AES with a 256-bit key as symmetric cipher, and type F bilinear pairing (for details see the manual of the PBC Library [22]). In the following we evaluate the computational overhead imposed by our scheme at system setup, file encryption (encryption) and file access (decryption) time, as well as in file transition from unpopular to popular.

In Section 5 we have presented a sub-optimal version of the upload protocol in order not to sacrifice the elegance in the descrion of the $\mathcal{E}_\mu$ scheme. For our experiments however, we have implemented a straightforward modification of the scheme that avoids the requirement of uploading two distinct ciphertexts. In particular, the file F is encrypted a first time with convergent encryption; the resulting ciphertext is encrypted a second time using standard symmetric encryption with a 256-bit key K. Finally, K is encrypted using a slightly modified version of $\mathcal{E}_\mu$ that allows to specify the H_1-hash to be used for computing ciphertext and decryption shares for K.

Setup. We evaluate the time required to setup $\mathcal{E}_\mu$, that includes the generation of the system parameters, and user keys. Note that this operation is performed by the storage provider only once. In our experiments we fix the number of users (e.g. user shares generated during the setup phase) to $n = 1000$. Figure 8 shows the impact of the popularity threshold t to the system setup time for a varying security parameter κ. As expected, for all values of κ, the time required for the system setup increases as the popularity threshold increases. This is mainly because a higher popularity threshold signifies the computation of a higher degree polynomial used for shares generation and this polynomial is evaluated for each user share using the Horner scheme. Nevertheless, the setup performance is in the order of tens of seconds, which can be considered insignificant for an operation which is executed only once.

File Upload. We fix the file size to 10MB and measure the time required for a file to be encrypted according to our scheme. Figure 9 shows the running time of file encryption using our scheme compared to the running times of conventional symmetric encryption and convergent encryption. The Figure shows the running time broken down into components: component 1 represents the running time of convergent key derivation ($\mathcal{E}_c.\mathsf{K}$), component 2 represents the running time of convergent encryption ($\mathcal{E}_c.\mathsf{E}$), component 3 represents the running time of symmetric encryption ($\mathcal{E}.\mathsf{E}$) and finally, component 4 represents the running time required for

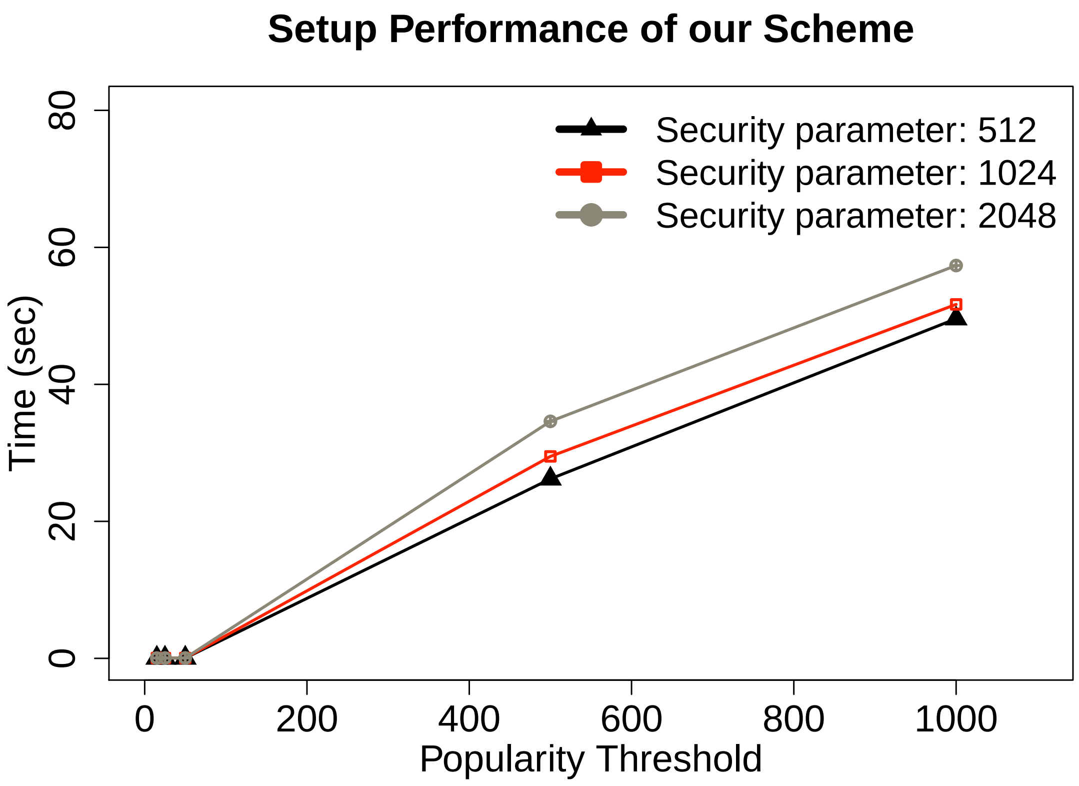


Figure 8: System Setup Performance for various popularity thresholds and values of the security parameter κ.

encrypting a symmetric key using $\mathcal{E}_\mu$ (that is, invoking $\mathcal{E}_\mu.\mathsf{Encrypt}$). The running time for $\mathcal{E}.\mathsf{K}$ is not represented as it is negligible.

It is easy to see that file encryption in our scheme takes roughly twice as much time as convergent encryption or symmetric encryption. We do not plot similar figures where the input file size varies, given that i) the running times of $\mathcal{E}_c.\mathsf{K}$, $\mathcal{E}_c.\mathsf{E}$ and $\mathcal{E}.\mathsf{E}$ are strictly linear in the input file size and ii) the running time of $\mathcal{E}_\mu.\mathsf{Encrypt}$ is independent from the input file size.

The communication overhead imposed at file upload can be broken down to the messages exchanged in (i) the user-IS communication session, where the former obtains a file index for her file, and (ii) the actual upload of the file ciphertext to the storage provider. The overall communication overhead at user-IS interaction accounts for 276B, and can be therefore considered negligible. At the upload of the file ciphertext, a user uploads the symmetric ciphertext which has roughly the same size as F, and the $\mathcal{E}_\mu$ ciphertext of K, which is 64B.

File Access. We measure the time required for a user to access a file that has been encrypted using our scheme. File access includes the communication overhead to download the file, and the computation overhead to decrypt it. Decryption of a popular file is just a classic convergent decryption (i.e., operation $\mathcal{E}_c.\mathsf{D}$). As to unpopular files, decryption consists of two symmetric decryption operations, $\mathcal{E}_c.\mathsf{D}$ and $\mathcal{E}.\mathsf{D}$.

The communication overhead at file decryption is roughly the same as the one of the file

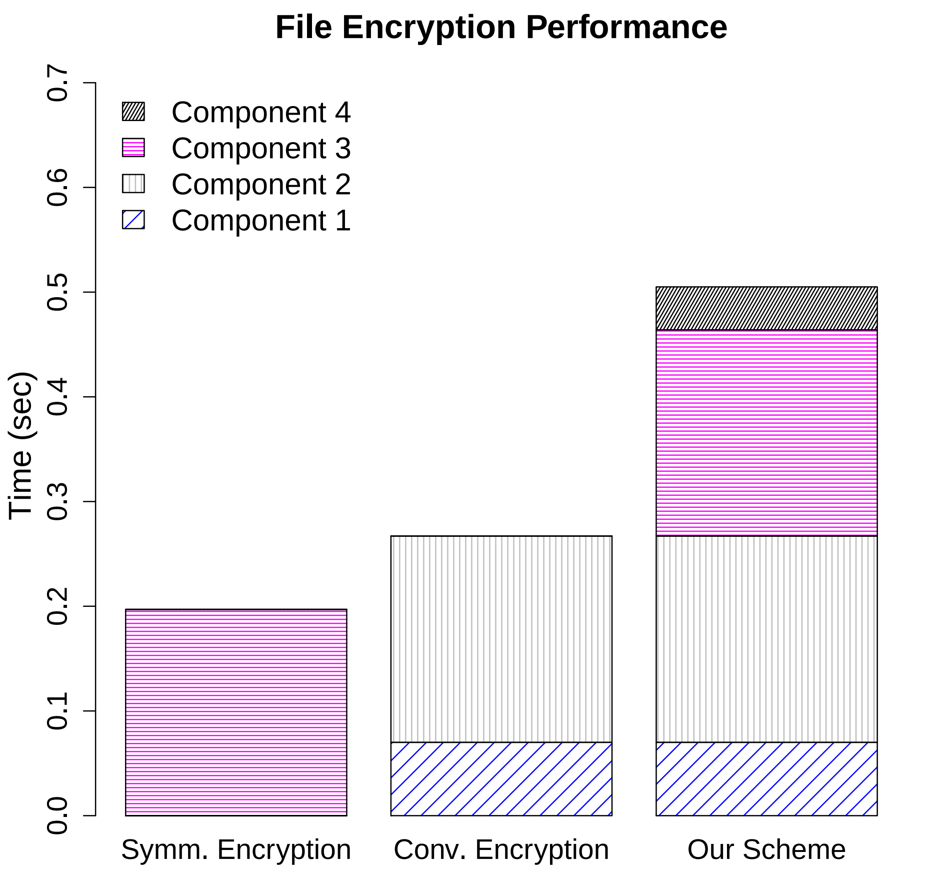


Figure 9: File Encryption Performance for the cases where: (i) the file is encrypted using a symmetric encryption scheme, (ii) the file is encrypted using a convergent encryption scheme (ii), and The file is encrypted using our scheme.

upload operation.

File Transition. We measure the time required for a file that has just reached the popularity threshold to transit form the unpopular state to the popular one. This operation takes place at the storage provider side only once per file. More specifically, at this time the storage provider, who has aggregated sufficient decryption shares, runs $\mathcal{E}_\mu$.Decrypt to recover K and $\mathcal{E}$.D to recover the convergent ciphertext of F. Note that there are t copies of F in storage, the storage provider will have to repeat this procedure for all t uploaded copies of F. As the running times for the decryption for both $\mathcal{E}$ and $\mathcal{E}_c$ are exactly the same as encryption, we only present the running times of $\mathcal{E}_\mu$.Decrypt, when the popularity threshold varies from 15 to 1000 in Table 1. It is easy to see that even when the popularity threshold is set to 1000, $\mathcal{E}_\mu$.Decrypt of K does not take more than fractions of a second.

Overall, we conclude that the overhead introduced by the threshold cryptosystem $\mathcal{E}_\mu$ is relatively small compared to the resource consumption of symmetric and convergent encryption operations. However, as our scheme combines both symmetric encryption and convergent encryption, the time needed for the encryption and decryption operations on an unpopular file is approximately the same as the sum of the time required for each of these operations when performed individually.

7.2 Analysis of Storage Efficiency

In this Section, we analyze, using a simulation, the ability of our scheme to reclaim space through deduplication. Recall that our scheme introduces a global threshold t. As t increases, storage efficiency is negatively impacted; indeed, files can be deduplicated only once at least t copies are uploaded. Before then, up to $t-1$ copies of the same file need to be stored.

In the simulation we have chosen values of t in the set $\{15, 25, 50\}$. The deduplication ratio ranges from 1:1 to 64:1, with power-of-two increments. The initial value corresponds to the ratio below which deduplication does not produce any gain, whereas the final value represents a very optimistic scenario that has been shown to be practically achievable in a backup setting [25]. The input dataset for our simulation is a set of 10^5 files of unitary size (since file size plays no role as far as efficiency is concerned). Estimating the popularity distribution is not a straightforward task as, in practice, it depends heavily on the input dataset and it can be highly variable. We have chosen to use the deduplication ratio as the basis for the mean value of file popularity and use some of the standard distributions – Pareto, Uniform and Exponential– to demonstrate the influence of this aspect. For example, Figure 10 shows the simulated distribution of file popularity for deduplication ratio 1:16 and threshold $t = 25$.

The results are shown in Figure 11. We observe that if the popularity of files fits Pareto distribution, then our space savings compared with savings of classic deduplication are only slightly worse, yet still quite good for any practical choice of the t value. If the popularity fits Uniform or Exponential distribution, the savings highly depend on the chosen value of t. While this simulation offers answers to the question of what influences the space savings efficiency of the proposed scheme and how, finding out how to derive a file popularity distribution is still an open problem and a target for our future work.

8 Discussion

In this section we justify some of our assumptions and discuss the limitations and future perspective of our protocols.

8.1 Privacy

Individual privacy is often equivalent to each party being able to control the degree to which that party will interact and share information with other parties and its environment. In the setting we are considering, user privacy is closely connected to user data confidentiality: it should not be possible to link a particular file plaintext to a particular individual with better probability than choosing that individual and file plaintext at random. Clearly, within our protocols, user privacy is provided completely for users who own only unpopular files, while privacy is

Table 1: Threshold Decryption Time (in milliseconds)

p_{lim}	15	25	50	500	1000
$\mathcal{E}_\mu$.Decrypt	35.43	37.20	43.43	195.78	364.61

slightly degraded for users who own popular files. One solution for the latter case would be to incorporate anonymous and unlinkable credentials [11, 23] every time user authentication is required. This way, a user who uploads a file to the storage provider will not have her identity linked to the file ciphertext. On the contrary, the file owner will be registered as one of the *certified users* of the system.

8.2 Flexibility on the Popularity Threshold

We have assumed that files are classified as popular or unpopular based on a single popularity threshold, which is decided by the identity provider. Naturally, the question arises whether the threshold could be set dynamically at file encryption time. One way to achieve such flexibility would be that the identity provider issues many keys for each user, each corresponding to a different popularity threshold. At file encryption, a user who has picked a popularity threshold t should upload shares that corresponds to t together with an indicator of which threshold has been used. This step is required to allow the storage provider to perform the steps required when reclaiming when the threshold is reached.

8.3 Deletion

Achieving deletion of content in our scheme is challenging, given that the storage provider may be malicious and refuse to erase the uploaded content. The case of unpopular files is of particular interest, given that ideally, a deletion operation should remove also the uploaded decryption share and decrease the popularity for that file by one. A malicious storage provider would undoubtedly refuse to perform this step. However, the indexing service, which is a trusted entity, would perform the deletion step honestly by removing the random index generated for the file and decreasing the popularity. This alone however does not guarantee any security. Indeed, we may be faced with the scenario in which the popularity threshold has not yet been reached (that is, the storage provider has not been given the set of indexes), and yet more than t decryption shares exist at unknown locations. The property of unlinkability of decryption shares described in Lemma 1 however guarantees that the adversary has no better strategy than trying all the $\binom{N}{t}$ possible set of t decryption shares among a total of N. While this does not constitute a formal argument, it is easy to show how, if N grows, this task becomes infeasible.

9 Conclusion

In this paper, we dealt with the inherent tension between well established storage optimization methods and end-to-end encryption. As opposed to related work that assumed that all files are equally security sensitive, we varied the security provisions of a file based on how popular that file is among the users of the system. We presented a novel encryption scheme that guarantees semantic security for unpopular data and provides weaker security and better storage and bandwidth benefits for popular data, so that data deduplication can be applied for the (less sensitive) popular data. In our system, encryption takes place at the client side, whereas decryption is client-independent. File transitions from one mode to the other take place seamlessly at the storage server side if and only if a file becomes popular. We showed that our protocols are secure under the Symmetric External Decisional Diffie Hellman Assumption, and that they scale well with large numbers of files and users.

References

[1] Bitcasa Infinite Storage. `https://www.bitcasa.com/`.

[2] Ciphertite Secure Backup. `https://www.cyphertite.com/`.

[3] DataLossDB. e-archive. `http://datalossdb.org/`.

[4] flud backup. `http://flud.org/`.

[5] The gnu multiple precision arithmetic library. `http://gmplib.org/`.

[6] Openssl cryptography and ssl/tls toolkit. `http://www.openssl.org/`.

[7] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009. ACM.

[8] Giuseppe Ateniese, Marina Blanton, and Jonathan Kirsch. Secret handshakes with dynamic and fuzzy matching. In *Network and Distributed System Security Symposuim*. The Internet Society, 2007.

[9] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Efficient implementation of pairing-based cryptosystems. *Journal of Cryptology*, 17(4):321–334, 2004.

[10] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. Cryptology ePrint Archive, Report 2012/631, 2012. `http://eprint.iacr.org/`.

[11] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Balancing accountability and privacy using e-cash. In *Security and Cryptography for Networks*, pages 141–155. Springer, 2006.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/`.

[13] Roberto Di Pietro and Alessandro Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 81–82, New York, NY, USA, 2012. ACM.

[14] John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

[15] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 617–, Washington, DC, USA, 2002. IEEE Computer Society.

[16] M. Dutch and L. Freeman. Understanding data de-duplication ratios. SNIA forum, 2008. `http://www.snia.org/sites/default/files/Understanding_Data_Deduplication_Ratios-20080718.pdf`.

[17] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 1984.

[18] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security*, pages 89–98, 2006.

[19] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 491–500, New York, NY, USA, 2011. ACM.

[20] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik. Estimation of deduplication ratios in large data sets. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1 –11, april 2012.

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

[22] Ben Lynn. The pairing-based cryptography library. `http://crypto.stanford.edu/pbc/`.

[23] Anna Lysyanskaya, Ronald L Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In *Selected Areas in Cryptography*, pages 184–199. Springer, 2000.

[24] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, pages 12–17, New York, NY, USA, 2008. ACM.

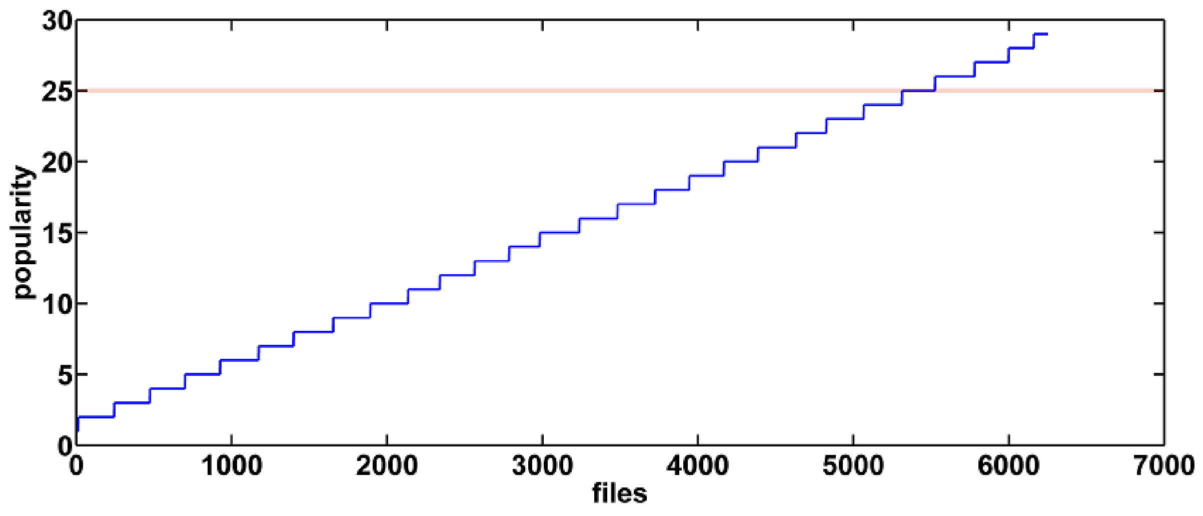
[25] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[26] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Proc. EUROCRYPT*, volume 3494, pages 457–473. Springer, 2005.

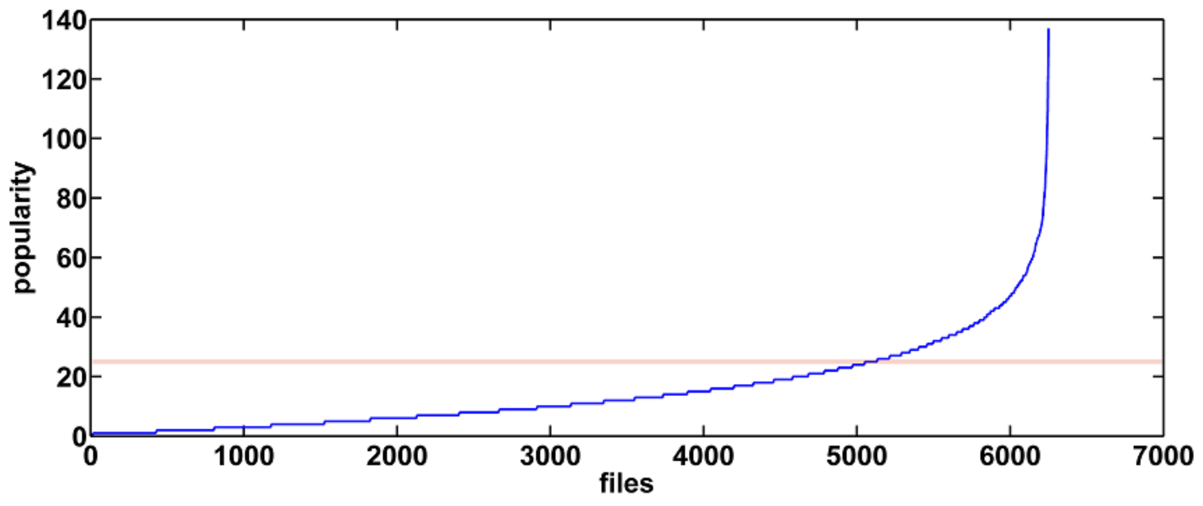
[27] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[28] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, pages 1–10, New York, NY, USA, 2008. ACM.

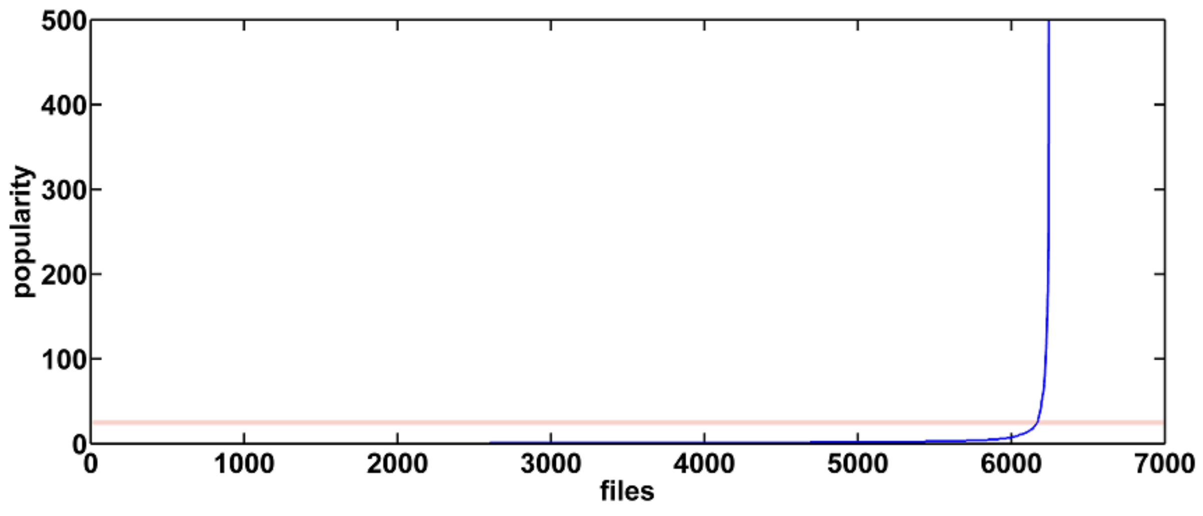
[29] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Leakage-resilient client-side deduplication of encrypted data in cloud storage. Cryptology ePrint Archive, Report 2011/538, 2011. `http://eprint.iacr.org/`.



(a) Uniform

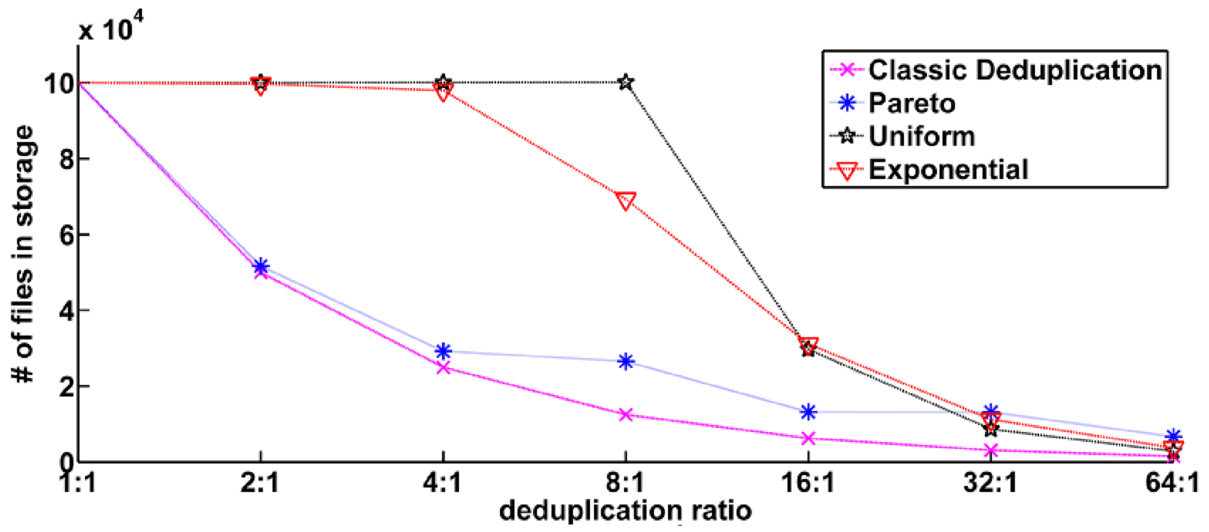


(b) Exponential

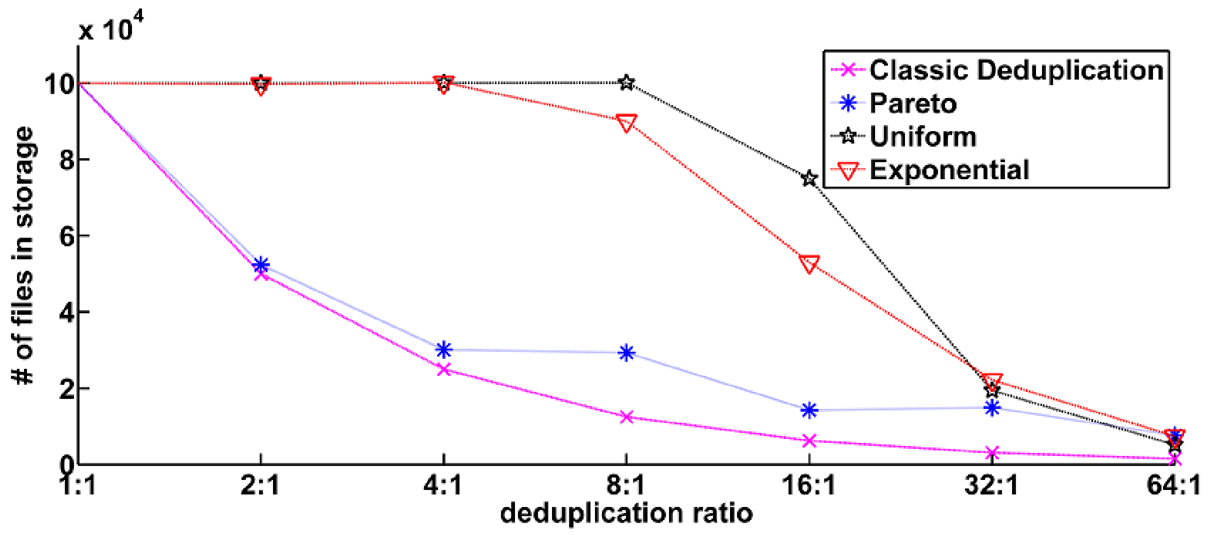


(c) Pareto

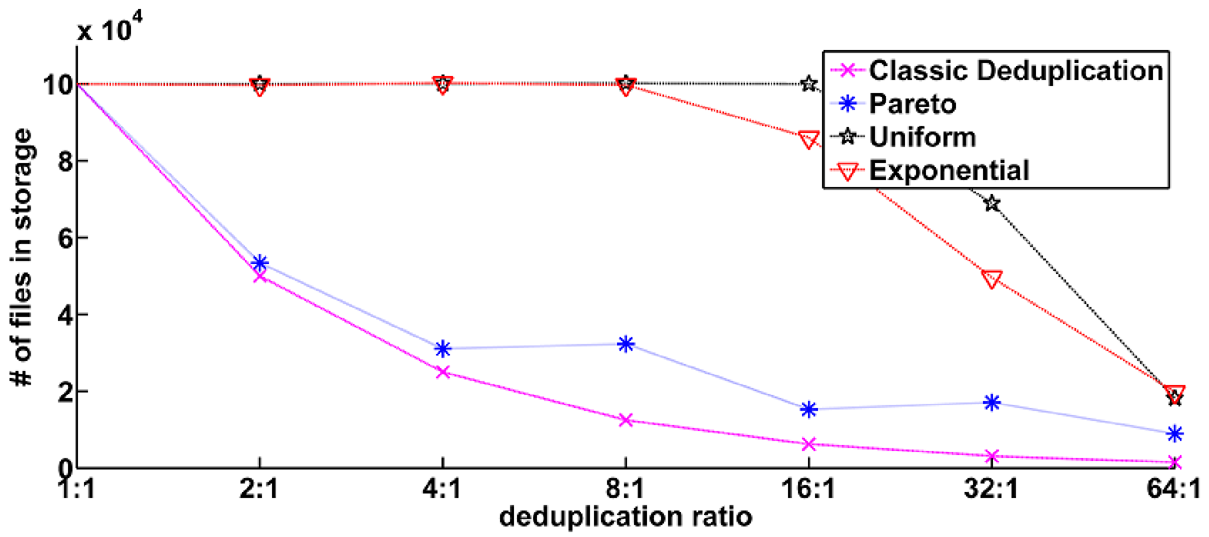
Figure 10: Plot of file popularity simulated for deduplication ratio 16:1; the horizontal line shows the global threshold $t = 25$, under which, files are not deduplicated.



(a) t=15



(b) t=25



(c) t=50

Figure 11: Demonstrating how t influences the space savings introduced by different popularity distributions.