A Secure Data Deduplication Scheme for Cloud Storage

Jan Stanek*, Alessandro Sorniotti†, Elli Androulaki†, and Lukas Kencl*

Abstract. As more corporate and private users outsource their data to cloud storage providers, 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, ineffective. We present a novel idea that differentiates data according to their popularity. Based on this idea, we design an 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. We show that our scheme is secure under the Symmetric External Decisional Diffie-Hellman Assumption in the random oracle model.

1 Introduction

With the rapidly increasing amounts of data produced worldwide, networked and multi-user storage systems are becoming very popular. However, concerns over data security still prevent many users from migrating data to remote storage. The conventional solution is to encrypt the data before it leaves the owner's premises. While sound from a security perspective, this approach prevents the storage provider from effectively applying storage efficiency functions, such as compression and deduplication, which would allow optimal usage of the resources and consequently lower service cost. Client-side data deduplication in particular ensures that multiple uploads of the same content only consume network bandwidth and storage space of 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). 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 is that outsourced data may require different levels of protection, depending on how *popular* it is: content shared by many users, such as a popular song or video, 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: (i) we present $\mathcal{E}_\mu$, a *novel threshold cryptosystem* (which can be of independent interest), together with a *security model* and *formal security proofs*, and (ii) we introduce a *scheme*

*Czech Technical University in Prague. `(jan.stanek|lukas.kencl)@fel.cvut.cz`.

†IBM Research - Zurich, Rüschlikon, Switzerland. `(aso|lli)@zurich.ibm.com`.

that uses $\mathcal{E}_\mu$ as a building block and enables to leverage popularity to achieve *both security and storage efficiency.* Finally, (iii) we discuss its overall security.

2 Problem Statement

Storage efficiency functions such as compression and deduplication afford storage providers 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 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 storage costs. Indeed, data deduplication is arguably one of the main reasons why the prices for cloud storage and cloud backup services have dropped so sharply.

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. It is becoming an increasingly prominent requirement due to both the number of security incidents linked to leakage of unencrypted data [1] and the tightening of sector-specific laws and regulations. Clearly, if semantically secure encryption is used, file deduplication is impossible, as no one—apart from the owner of the decryption key—can decide whether two ciphertexts correspond to the same plaintext. Trivial solutions, such as forcing users to share encryption keys or using deterministic encryption, fall short of providing acceptable levels of security.

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.

2.1 Related Work

Several deduplication schemes have been proposed by the research community [2–4] showing how deduplication allows very appealing reductions in the usage of storage resources [5, 6].

Most works do not consider security as a concern for deduplicating systems; recently however, Harnik *et al.* [7] 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 [8, 9]. 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.* [10, 11], 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.* [12] formalized convergent encryption under the name *message-locked encryption*. As expected, the security analysis presented in [12] highlights that message-locked encryption offers confidentiality for unpredictable messages only, clearly failing to achieve semantic security.

Xu *et al.* [13] present a PoW scheme allowing client-side deduplication in a bounded leakage setting. They provide a security proof in a random oracle model for their solution, but do not address the problem of low min-entropy files.

Recently, Bellare *et al.* presented DupLESS [14], a server-aided encryption for deduplicated storage. Similarly to ours, their solution uses a modified convergent encryption scheme with the aid of a secure component for key generation. While DupLESS offers the possibility to securely use server-side deduplication, our scheme targets secure client-side deduplication.

3 Overview of the Solution

Deduplication-based systems require solutions tailored to the type of data they are expected to handle [5]. We focus our analysis on scenarios where the outsourced dataset contains few instances of some data items and many instances of others. Concrete examples of such datasets include (but are not limited to) those handled by Dropbox-like backup tools and hypervisors handling linked clones of VM-images. Other scenarios where such premises do not hold, require different solutions and are out of the scope of this paper.

The main intuition behind our scheme is that there are scenarios in which data requires 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 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, may instead contain user-generated content which requires confidentiality, and would by definition not allow reclaiming a lot of space via deduplication. The same can be said about common blocks of shared VM images, mail attachments sent to several recipients, 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 [12]), and (ii) the properties of the remaining convergent encryption layer allow deduplication to happen naturally. Security is thus traded for storage efficiency as for every file that transits from unpopular to popular status, storage space can be reclaimed. Once a file reaches the popular status, space is reclaimed for the copies uploaded so far, and normal deduplication can take place for future copies. Standard security mechanisms (such as Proof of Ownership [8, 9]) can be applied to secure this step. Note that such mechanisms are not required in the case of unpopular files, given that they are protected by both encryption layers and cannot be deduplicated.

There are two further challenges in the secure design of the scheme. Firstly, without proper identity management, sybil attacks [15] 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 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 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.

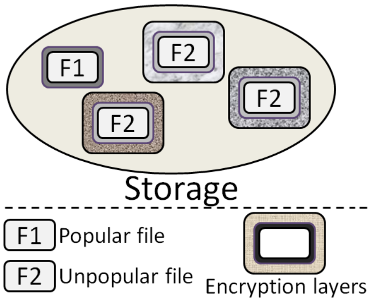

Fig. 1. The multi-layered cryptosystem used in our scheme. Unpopular files are protected using two layers, whereas for popular files, the outer layer can be removed. The inner layer is obtained through convergent encryption that generates identical ciphertext at each invocation. The outer layer (for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

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.

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, Dropbox 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.

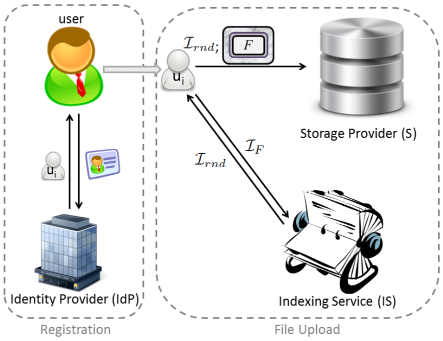

Fig. 2. Illustration of our system model. The schematic shows the main four entities and their interaction for registration and file upload process.

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: i)*Semantic security* [16] for unpopular data; ii) *Conventional convergent security* [12] 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 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. Fixing a single threshold t arguably reduces the flexibility of the scheme. While for the sake of simplicity of notation we stick to a single threshold, section 7 discusses how this restriction can be lifted.

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 cannot compromise them. While the existence of an identity provider is not uncommon and is often an essential building block of many practical deployments, we adopt the indexing service as a way to focus our analysis on the security of the content of files, and to thwart attacks to its indexes by means of the trusted nature of this separate entity. TTPs are indeed often adopted as a means of achieving security objectives all the while preserving usability [17, 18].

4 Building Blocks

Modeling Deduplication In this Section we will 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 takes place at the client side, thus avoiding the need to upload the duplicate 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 $\mathsf{DB}\left[\mathcal{I}_F\right]$ is the record mapped to the index of file F. In a simple deduplication scheme, records contain two fields, $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{data}$ and $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{users}$. 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 $\mathsf{DB}\left[\mathcal{I}_F\right]$ exists. If it does, the server appends u to $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{users}$. Otherwise, it requests u to upload the content of F, which will be assigned to $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{data}$. $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{users}$ is then initialized with u.
- Get: user u sends $\mathcal{I}_F$ to the server. The server checks whether $\mathsf{DB}\left[\mathcal{I}_F\right]$ exists and whether $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{users}$ contains u. If it does, the server responds with $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{data}$. Otherwise, it answers with an error message.

Symmetric Cryptosystems and Convergent Encryption A *symmetric cryptosystem* $\mathcal{E}$ is defined as a tuple (K, E, 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) (K, E, 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.

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 cryptosystem $\mathcal{E}_t$ 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) with the following properties:

$\mathsf{Setup}(\kappa, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk}, \mathsf{S})$: generates the public key of the system pk, the corresponding private key sk and a set $\mathsf{S} = \{(\mathsf{r_i}, \mathsf{sk_i})\}_{i=0}^{n-1}$ of n pairs of *key shares* $\mathsf{sk_i}$ of the private key with their indexes $\mathsf{r_i}$; key shares are secret, and are distributed to authorized users; indexes on the other hand need not be secret.

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

$\mathsf{DShare}(\mathsf{r_i}, \mathsf{sk_i}, m) \rightarrow (\mathsf{r_i}, \mathsf{ds_i})$: takes as input a message m and a key share $\mathsf{sk_i}$ with its index $\mathsf{r_i}$ and produces a *decryption share* $\mathsf{ds_i}$; the index is also outputted.

$\mathsf{Decrypt}(c, \mathsf{S}_t) \rightarrow (m)$: takes as input a ciphertext c, a set $\mathsf{S}_t = \{(\mathsf{r_i}, \mathsf{ds_i})\}_{i=0}^{t-1}$ of t pairs of decryption shares and indexes, and outputs the cleartext message m.

5 Our Scheme

In this Section we will formally introduce our scheme. First, we will present a novel cryptosystem of independent interest, 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

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 $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is a computable, non-degenerate bilinear pairing. We further assume that there is no efficient distortion map $\psi : \mathbb{G}_1 \rightarrow \mathbb{G}_2$, or $\psi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$. These groups are commonly referred to as SXDH groups, i.e., groups where it is known that the Symmetric Extensible Diffie Hellman Assumption(SXDH) [19] holds.

$\mathcal{E}_\mu$ 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):

$\mathsf{Setup}(\kappa, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk}, \mathsf{S})$: 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 secret $x \leftarrow_R \mathbb{Z}_q^*$ and $\{x_i\}_{i=0}^{n-1}$ be n shares of x such that any set of t shares can be used to reconstruct x through polynomial interpolation (see [20] for more details). Also, let $\bar{g}_{pub} \leftarrow \bar{g}^x$. Finally, let $H_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $H_2 : \mathbb{G}_T \rightarrow \{0,1\}^l$ for some l, be two cryptographic hash functions. Public key pk is set to $\{q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, H_1, H_2, g, \bar{g}, \bar{g}_{pub}\}$, sk to x; S is the set of n pairs $(\mathsf{r_i}, \mathsf{sk_i})$, where $\mathsf{sk_i}$ is set to x_i and $\mathsf{r_i}$ is the preimage of x_i under the aforementioned polynomial.

$\mathsf{Encrypt}(\mathsf{pk}, m) \rightarrow (c)$: let $r \leftarrow_R \mathbb{Z}_q^*$ 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_1, c_2).

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

$\mathsf{Decrypt}(c, \mathsf{S}_t) \rightarrow (m)$: parse c as (c_1, c_2) and S_t as $\{(\mathsf{r_i}, \mathsf{ds_i})\}_{i=0}^{t-1}$; compute

$$\prod_{(\mathsf{r_i},\mathsf{ds_i})\in \mathsf{S}_t} \mathsf{ds_i}^{\lambda_{0,\mathsf{r_i}}^{\mathsf{S}_t}} = \prod_{(\mathsf{r_i},\mathsf{sk_i})\in \mathsf{S}_t^{'}} H_1(m)^{\mathsf{sk_i}\lambda_{0,\mathsf{r_i}}^{\mathsf{S}_t}} = H_1(m)^{\sum\limits_{(\mathsf{r_i},\mathsf{sk_i})\in \mathsf{S}_t^{'}} \mathsf{sk_i}\lambda_{0,\mathsf{r_i}}^{\mathsf{S}_t}} = H_1(m)^{\mathsf{sk}},$$

where $\lambda_{0,\mathsf{r_i}}^{\mathsf{S}_t}$ are the Lagrangian coefficients of the polynomial with interpolation points from the set $\mathsf{S}_t^{'} = \{(\mathsf{r_i}, \mathsf{sk_i})\}_{i=0}^{t-1}$. 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, $\hat{e}\left(H_1(m)^x, \bar{g}^r\right) = \hat{e}\left(H_1(m), \bar{g}_{pub}\right)^r = \hat{e}\left(H_1(m), \bar{g}^x\right)^r$. This equality satisfies considerations on the correctness of $\mathcal{E}_\mu$.

$\mathcal{E}_\mu$ has a few noteworthy properties: i) 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; ii) 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 less than t decryption shares are collected; iii) 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. Finally, note that it is possible to generate more shares $\mathsf{sk_j}$ $(j > n)$ anytime after the execution of the Setup algorithm, to allow new users to join the system even if all the original n key-shares were already assigned.

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 [15]. The identity provider is also responsible for the execution of $\mathcal{E}_\mu.\mathsf{Setup}$, and is trusted not to leak the secret key of the system, nor to use this knowledge to violate confidentiality of unpopular data. This assumption is consistent to the trust users put on today's identity providers.

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), 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 (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 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 $\mathsf{DB}_{IS}\left[\mathcal{I}_F\right]$ with two fields, $\mathsf{DB}_{IS}\left[\mathcal{I}_F\right].\mathsf{ctr}$ and $\mathsf{DB}_{IS}\left[\mathcal{I}_F\right].\mathsf{idxes}$. 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 n users U_i, a storage provider S and the two trusted entities, the identity provider IdP and the indexing service IS. S is modeled as described in Section 4; the database record contains an extra boolean field, $\mathsf{DB}\left[\mathcal{I}_F\right].\mathsf{popular}$, 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); $\mathcal{E}_\mu$ is our convergent threshold cryptosystem. The scheme consists of the following distributed algorithms:

Init: IdP executes $\mathcal{E}_\mu.\mathsf{Setup}$, publishes the public key system pk of the system. IdP keeps key shares $\{\mathsf{sk_i}\}_{i=0}^{n-1}$ secret.

Join: whenever a user U_i wants to join the system, they contact 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 $\mathsf{sk_i}$ (generating a new $\mathsf{sk_j}$ if necessary).

Upload (Fig. 4): this algorithm is executed between a user U_i, the storage server S and the indexing service IS whenever U_i requests upload of a file F. First, U_i uses convergent encryption to create ciphertext F_c; U_i then interacts with IS to obtain an index $\mathcal{I}_{ret}$ (note that either $\mathcal{I}_{ret} = \mathcal{I}_{rnd}$ or $\mathcal{I}_{ret} = \mathcal{I}_{F_c}$) to use for the interaction with S and possibly a list of indexes used by other users when uploading the same file. Depending on IS's response, U_i proceeds with one of the following sub-algorithms:

– $\mathsf{Upload.Unpopular}$ (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 $\mathcal{E}_\mu$,

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

Fig. 3. The GetIdx algorithm.

```
$\mathsf{U}_i$:                      $K_c \leftarrow \mathcal{E}_c.\mathsf{K}(F)$
                       $F_c \leftarrow \mathcal{E}_c.\mathsf{E}(K_c, F)$
                       $\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
$\mathsf{U}_i \longrightarrow \mathsf{IS}$:   $\mathcal{I}_{F_c}$
$\mathsf{U}_i \longleftarrow \mathsf{IS}$:   $\langle \mathcal{I}_{ret}, \mathsf{I} \rangle \leftarrow \mathsf{GetIdx}(\mathcal{I}_{F_c})$
$\mathsf{U}_i$:                      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
```

Fig. 4. The Upload algorithm.

respectively. The first ciphertext allows U_i to recover 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[1]. U_i replaces F with a stub of 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 $\mathsf{DB}[\mathcal{I}_{F_c}].\mathsf{data}$. Note that, because of the nature of convergent encryption, all decrypted instances are identical, hence deduplication happens automatically. 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, $\mathcal{I}_{ret} = \mathcal{I}_{F_c}$. Here, the user is not expected to upload the content of the file as it has already been

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

Fig. 5. The Upload.Unpopular alg.

```
$\mathsf{U}_i \longrightarrow \mathsf{S}$:   $\mathsf{I}$
$\mathsf{S}$:                  $\mathsf{DS} \leftarrow \{\mathsf{ds} : \langle \mathsf{c}, \mathsf{c}_\mu, \mathsf{ds} \rangle \leftarrow$
                      $\leftarrow \mathsf{DB}[\mathcal{I}].\mathsf{data},\ \mathcal{I} \in \mathsf{I}\}$
                   foreach($\mathcal{I}_i \in \mathsf{I}$)
                      $\langle c, c_\mu, \mathsf{ds_i} \rangle \leftarrow \mathsf{DB}[\mathcal{I}_F].\mathsf{data}$
                      $F_c \leftarrow \mathcal{E}_\mu.\mathsf{Decrypt}(c_\mu, \mathsf{DS})$
                      $\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
                      $\mathsf{U}_i \leftarrow \mathsf{DB}[\mathcal{I}_i].\mathsf{users}$
                      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 $\mathsf{I}$
```

Fig. 6. The Upload.Reclaim algorithm

[1] 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, 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 ciphertext of the file and not two as described above.

declared popular. U_i replaces F with a stub containing the index $\mathcal{I}_{F_c}$ and of the key K_c.

$\mathsf{Download}$: 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.

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

Fig. 7. The Upload.Popular algorithm

6 Security Analysis

We formally analyze the security of the $\mathcal{E}_\mu$ cryptosystem and we argue informally that the security requirements of Sec. 3.2 are met by our scheme as a whole.

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_μ-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 as DS_μ-IND. Both experiments require the adversary to declare upfront the set of users to be corrupted, similarly to selective security [21].

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 *corrupt* an arbitrary number $n_\mathcal{A} < t - 1$ of pre-declared users, and obtains their secret keys through an oracle $\mathcal{O}_{\mathsf{Corrupt}}$. 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 whether to respond with a properly constructed decryption share that corresponds to message m and secret key share $\mathsf{sk_i}$ 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 experiment DS_μ-IND between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, given security parameter κ:

Setup Phase $\mathcal{C}$ executes the Setup algorithm with κ, and generates a set of user identities $\mathsf{U} = \{\mathsf{U_i}\}_{\mathsf{i=0}}^{\mathsf{n-1}}$. Further, $\mathcal{C}$ gives pk to $\mathcal{A}$ and keeps $\{\mathsf{sk_i}\}_{i=0}^{n-1}$ 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 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 a $\mathcal{O}_{\mathsf{DShare}}$-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_*}.

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

Semantic Security Informally, in IND_{μ}-CPA, the adversary is given access to all oracles as in DS_{μ}-IND. However, here, oracle $\mathcal{O}_{\mathsf{DShare}}$ responds with properly constructed decryption shares, i.e., decryption shares that correspond to the queried message and non-corrupted user identity. 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 through the IND_{μ}-CPA experiment 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 and the $\mathcal{O}_{\mathsf{Corrupt}}$ oracle as in DS_{μ}-IND. $\mathcal{A}$ is given access to 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 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, which is returned to $\mathcal{A}$.

Guessing Phase $\mathcal{A}$ 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 and semantic security are guaranteed in $\mathcal{E}_\mu$ as long as the SXDH problem is intractable [19].

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.*

Lemma 2. *Let* H_1*, and* H_2 *be random oracles. If an* IND_μ*-*CPA *adversary* $\mathcal{A}$ *has a non-negligible advantage* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_\mu\text{-}\mathsf{CPA}} := \text{Prob}[\mathsf{b}' \leftarrow \mathcal{A}(c_*) : \mathsf{b} = \mathsf{b}'] - \frac{1}{2}$ *, 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.*

Proofs for Lemma 1 and Lemma 2 are available in the appendices.

6.2 Security Analysis of the Scheme

A formal security analysis of the scheme under the UC framework [22] is not presented here due to space limitations 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 $\{\mathsf{U}_i\}_{i=1}^{n_\mathcal{A}}$. The popularity threshold p_{lim} 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 S 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 a 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 Discussion

Here we justify some of our assumptions and discuss the scheme's limitations:

Prototype Performance. To demonstrate the practicality and functionality of our proposal, we implemented a prototype of the core of the scheme as a client-server C++ application. Results show that the most overhead stems from symmetric and convergent encryption operations implemented via AES-256 and SHA-256; the execution of $\mathcal{E}_\mu.\mathsf{Encrypt}$ and $\mathcal{E}_\mu.\mathsf{Decrypt}$ forms only a fraction of the computational overhead. Additionally, while $\mathcal{E}_\mu.\mathsf{Encrypt}$ is executed for every store and retrieval operation, $\mathcal{E}_\mu.\mathsf{Decrypt}$ is used only during file state transition and user share generation is done only once per every new registered user. In conclusion, most of the computational overhead is caused by convergent and symmetric encryption to protect unpopular files, while the overhead introduced by the threshold cryptosystem is comparatively small.

Privacy. Individual privacy is often equivalent to each party being able to control the degree to which it will interact and share information with other parties and its environment. In our setting, 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 it is slightly degraded for users who own popular files. One solution for the latter case would be to incorporate anonymous and unlinkable credentials [23, 24] 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.

Dynamic Popularity Threshold. In our scheme, files are classified as popular or unpopular based on a single popularity threshold. One way of relaxing this requirement would be to create multiple instances of $\mathcal{E}_\mu$ with different values of t and issue as many keys to each user. Different users are then free to encrypt an input file using different thresholds, with the property that a file uploaded with a given threshold t_1 does not count towards popularity for the same file uploaded with a different threshold t_2 (otherwise, malicious users could easily compromise the popularity system). A label identifying the chosen threshold (which does not leak other information) must be uploaded together with the ciphertext. Furthermore, the indexing service needs to be modified to keep indexes for a given file and threshold separate from those of the same file but different thresholds. This can be achieved by modifying the GetIdx interface.

Deletion. Deletion of content is challenging in our scheme, given that the storage provider may be malicious and refuse to erase the uploaded content. Ideally, a deletion operation should remove also the uploaded decryption share and decrease the popularity of 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 $\mathsf{ds_i}$ shares of currently unpopular files stored in the storage. While this does not constitute a formal argument, it is easy to show how, if number of shares grows, this task becomes infeasible.

8 Conclusion

This work deals with the inherent tension between well established storage optimization methods and end-to-end encryption. Differently from the approach of related works, that assume all files to be equally security-sensitive, we vary the security level of a file based on how popular that file is among the users of the system. 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, so that data deduplication can be applied for the (less sensitive) popular data. Files transition from one mode to the other in a seamless way as soon as they become popular. We show that our protocols are secure under the SXDH Assumption. In the future we plan to deploy and test the proposed solution and evaluate the practicality of the notion of popularity and whether the strict popular/unpopular classification can be made more fine-grained. Also, we plan to remove the assumption of a trusted indexing service and explore different means of securing the indexes of unpopular files.

Acknowledgements

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/139/OHK3/2T/13.

References

1. Open Security Foundation: DataLossDB (`http://datalossdb.org/`).
2. Meister, D., Brinkmann, A.: Multi-level comparison of data deduplication in a backup scenario. In: SYSTOR '09, New York, NY, USA, ACM (2009) 8:1–8:12
3. Mandagere, N., Zhou, P., Smith, M.A., Uttamchandani, S.: Demystifying data deduplication. In: Middleware '08, New York, NY, USA, ACM (2008) 12–17
4. Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., Klein, S.T.: The design of a similarity based deduplication system. In: SYSTOR '09. (2009) 6:1–6:14
5. Dutch, M., Freeman, L.: Understanding data de-duplication ratios. SNIA forum (2008) `http://www.snia.org/sites/default/files/Understanding_Data_Deduplication_Ratios-20080718.pdf`.
6. Harnik, D., Margalit, O., Naor, D., Sotnikov, D., Vernik, G.: Estimation of deduplication ratios in large data sets. In: IEEE MSST '12. (april 2012) 1 –11
7. Harnik, D., Pinkas, B., Shulman-Peleg, A.: Side channels in cloud services: Deduplication in cloud storage. Security Privacy, IEEE **8**(6) (nov.-dec. 2010) 40 –47
8. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: CCS '11, New York, NY, USA, ACM (2011) 491–500
9. Di Pietro, R., Sorniotti, A.: Boosting efficiency and security in proof of ownership for deduplication. In: ASIACCS '12, New York, NY, USA, ACM (2012) 81–82
10. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, D., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: ICDCS '02, Washington, DC, USA, IEEE Computer Society (2002) 617–632
11. Storer, M.W., Greenan, K., Long, D.D., Miller, E.L.: Secure data deduplication. In: StorageSS '08, New York, NY, USA, ACM (2008) 1–10
12. Bellare, M., Keelveedhi, S., Ristenpart, T.: Message-locked encryption and secure deduplication. In: Advances in Cryptology–EUROCRYPT 2013. Springer 296–312
13. Xu, J., Chang, E.C., Zhou, J.: Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In: 8th ACM SIGSAC symposium. 195–206
14. Bellare, M., Keelveedhi, S., Ristenpart, T.: DupLESS: server-aided encryption for deduplicated storage. In: 22nd USENIX conference on Security. (2013) 179–194
15. Douceur, J.R.: The sybil attack. In: Peer-to-peer Systems. Springer (2002) 251–260
16. Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. (1984)
17. Fahl, S., Harbach, M., Muders, T., Smith, M.: Confidentiality as a service–usable security for the cloud. In: TrustCom 2012. 153–162
18. Fahl, S., Harbach, M., Muders, T., Smith, M., Sander, U.: Helping johnny 2.0 to encrypt his facebook conversations. In: SOUPS 2012. 11–28
19. Ateniese, G., Blanton, M., Kirsch, J.: Secret handshakes with dynamic and fuzzy matching. In: NDSS '07
20. Shamir, A.: How to share a secret. Commun. ACM **22**(11) (November 1979)
21. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: ACM CCS '06. 89–98
22. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: STOC '02. (2002)
23. Camenisch, J., Hohenberger, S., Lysyanskaya, A.: Balancing accountability and privacy using e-cash. In: Security and Cryptography for Networks. Springer (2006)
24. Lysyanskaya, A., Rivest, R.L., Sahai, A., Wolf, S.: Pseudonym systems. In: Selected Areas in Cryptography, Springer (2000) 184–199

Appendix A: Proof of Lemma 1

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 $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. In this setting, 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 $\hat{e}$ [19].

Challenger $\mathcal{C}$ 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 $\mathbb{G}_1$'. $\mathcal{C}$ simulates an environment in which $\mathcal{A}$ operates, using its advantage in the game DS_μ-IND to decide whether $W = g'^{ab}$. $\mathcal{C}$ interacts 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 $\mathsf{U} = \{\mathsf{U}_i\}_{i=0}^{n-1}$. 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 forwarded to $\mathcal{A}$. $\mathcal{A}$ 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. Let $\mathsf{U}_\mathcal{A} = \{\mathsf{U}_i\}_{i=0}^{n_\mathcal{A}-1}$. To generate key-shares $\{\mathsf{sk_i}\}_{i=0}^{n-1}$, $\mathcal{C}$ constructs a $t-1$-degree Lagrange polynomial $\mathrm{P}()$ with interpolation points $\mathrm{I_P} = \{(0, \mathsf{sk}) \cup \{(\mathsf{r_i}, y_i)\}_{i=0}^{t-2}\}$, where $\mathsf{r_i}, y_i \leftarrow_R \mathbb{Z}_q^*$, for $i \in [0, t-3]$, and $\mathsf{r_{t-2}} \leftarrow_R \mathbb{Z}_q^*$, $y_{t-2} \leftarrow a$. Secret key-shares are set to $\mathsf{sk_i} \leftarrow y_i, i \in [0, n-1]$. Since a is not known to $\mathcal{C}$, $\mathcal{A}$ sets the corrupted key-shares to be $\mathsf{sk_i}$ for $i \in [0, n_\mathcal{A} - 1]$.

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

$\mathcal{O}_{\mathsf{H}_1}$: to respond to $\mathcal{O}_{\mathsf{H}_1}$-queries, $\mathcal{C}$ maintains a list of tuples $\{\mathsf{H}_1, v, h_v, \rho_v, \mathrm{c}_v\}$ as explained below. We refer to this list as $\mathcal{O}_{\mathsf{H}_1}$ list, and it 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 $\{v, h_v, \rho_v, \mathrm{c}_v\}$. If so, $\mathcal{C}$ responds with $\mathsf{H}_1(v) = h_v$. Otherwise, $\mathcal{C}$ picks $\rho_v \leftarrow_R \mathbb{Z}_q^*$, 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 $\mathsf{H}_1(v) = h_v = g^{\rho_v}$ and stores $\{v, h_v, \rho_v, \mathrm{c}_v\}$; otherwise, she returns $\mathsf{H}_1(v) = h_v = B^{\rho_v}$ and stores $\{v, h_v, \rho_v, \mathrm{c}_v\}$.

$\mathcal{O}_{\mathsf{H}_2}$: 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 $\{m, h_m, \rho_m, \mathrm{c}_m\}$ be the $\mathcal{O}_{\mathsf{H}_1}$ entry in $\mathcal{C}$'s local memory for message m. Let $\mathrm{I_P}' \leftarrow \mathrm{I_P} \setminus (\mathsf{r_{t-2}}, \mathsf{sk_{t-2}})$. $\mathcal{C}$ responds with $\mathsf{ds}_{m,i} = \left(g^{\sum_{(\mathsf{r_j}, \mathsf{sk_j}) \in \mathrm{I_P}'} \mathsf{sk_j} \lambda^{\mathrm{I_P}'}_{\mathsf{r_i}, \mathsf{r_j}}} X^{\lambda^{\mathrm{I_P}'}_{\mathsf{r_i}, \mathsf{r_{t-2}}}} \right)^{\rho_m}$ 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 $\mathcal{A}$ selects the challenge message m_*. Let the corresponding entry in the $\mathcal{O}_{\mathsf{H}_1}$ list be $\{m_*, h_{m_*}, \rho_{m_*}, \mathrm{c}_{m_*}\}$. If $\mathrm{c}_{m_*} = 0$, then $\mathcal{C}$ aborts.

Guessing Phase $\mathcal{A}$ 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_*. This can only be if $W = g^{ab}$ and $\mathcal{C}$ can give a positive answer to the SXDH challenger. Clearly, if $\mathrm{c}_{m_*} = 1$ and $\mathrm{c}_m = 0$ for all other queries to $\mathcal{O}_{\mathsf{H}_1}$ such that $m \neq m_*$, 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 distinct $\mathcal{O}_{\mathsf{H}_1}$ queries. By setting $\delta \approx \frac{1}{\mathcal{Q}_{H_1}-1}$ the above probability becomes greater than $\frac{1}{e\cdot(\mathcal{Q}_{H_1}-1)}$ and the success probability of the adversary can be bounded as $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{DS}_\mu\text{-}\mathsf{IND}} \leq e \cdot (\mathcal{Q}_{H_1}-1) \cdot \mathsf{Adv}^{\mathcal{C}}_{\mathsf{SXDH}}$.

Appendix B: Proof of Lemma 2

Challenger $\mathcal{C}$ 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 described before. $\mathcal{C}$ interacts with $\mathcal{A}$ within an IND_μ-CPA game:

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 of corrupted users $\mathsf{U}_\mathcal{A}$ as in DS_μ-IND. Let $\mathsf{U}_\mathcal{A} = \{\mathsf{U}_i\}_{i=0}^{n_\mathcal{A}-1}$. To generate key-shares $\{\mathsf{sk_i}\}_{i=0}^{n-1}$, $\mathcal{C}$ picks a $t-1$ degree Lagrange polynomial $\mathrm{P}()$ assuming interpolation points $\mathrm{I_P} = \left\{(0,a)\ \cup\ \{(\mathsf{r_i}, y_i)\}_{i=0}^{t-2}\right\}$, where $\mathsf{r_i}, y_i \leftarrow_R \mathbb{Z}_q^*$. She then sets the key-shares to $\mathsf{sk_i} \leftarrow y_i, i \in [0, n-1]$ and assigns $\mathsf{sk_i}$ for $i \in [0, n_\mathcal{A}-1]$ to corrupted users.

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

$\mathcal{O}_{\mathsf{H}_1}$, $\mathcal{O}_{\mathsf{H}_2}$, $\mathcal{O}_{\mathsf{Corrupt}}$: $\mathcal{C}$ responds to these queries as in DS_μ-IND.

$\mathcal{O}_{\mathsf{DShare}}$: $\mathcal{C}$ 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 $\{m, h_m, \rho_m, \mathrm{c}_m\}$ 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, $\mathcal{C}$ aborts. If the limit of $t - n_\mathcal{A} - 1$ queries has not been reached, $\mathcal{C}$ responds with a random $\mathsf{ds}_{m,i} \in \mathbb{G}_1$ and keeps a record for it. From Lemma 1, this step is legitimate as long as less than t decryption shares are available for m. Let $\mathrm{I_P}' \leftarrow \mathrm{I_P} \setminus (0,a)$. If $\mathrm{c}_m = 0$, $\mathcal{C}$ responds with $\mathsf{ds}_{m,i} = \left(g^{\sum_{(\mathsf{r_j},\mathsf{sk_j}) \in \mathrm{I_P}'} \mathsf{sk_j} \lambda^{\mathrm{I_P}'}_{\mathsf{r_i},\mathsf{r_j}}} A^{\lambda^{\mathrm{I_P}'}_{\mathsf{r_i},0}} \right)^{\mathsf{r_m}}$.

Challenge Phase $\mathcal{A}$ submits m_* to $\mathcal{C}$. $\mathcal{A}$ 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 $\{m_*, h_{m_*}, \rho_{m_*}, \mathrm{c}_{m_*}\}$. If $\mathrm{c}_{m_*} = 0$, $\mathcal{C}$ aborts. Otherwise, $\mathcal{C}$ computes $e_* \leftarrow W^{\rho_{m_*}}$, sets $c_* \leftarrow \langle m_* \oplus \mathsf{H}_2(e_*), \bar{C}\rangle$ and returns c_* to $\mathcal{A}$.

Guessing Ph. $\mathcal{A}$ outputs the guess b' for b. $\mathcal{C}$ provides b' 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, $W^{\rho_{m_*}} = \hat{e}\,(g, \bar{g})^{abc\rho_{m_*}} = \hat{e}\left((B^{\rho_{m_*}})^a, \bar{g}^c\right) = \hat{e}\left(H_1(m_*)^{\mathsf{sk}}, \bar{C}\right)$. Clearly, 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 $\delta \approx \frac{1}{\mathcal{Q}_{H_1}-1}$, the above probability becomes greater than $\frac{1}{e\cdot(\mathcal{Q}_{H_1}-1)}$, and the success probability of the adversary $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_\mu\text{-}\mathsf{CPA}}$ is bounded as $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_\mu\text{-}\mathsf{CPA}} \leq e \cdot (\mathcal{Q}_{H_1}-1) \cdot \mathsf{Adv}^{\mathcal{C}}_{\mathsf{SXDH}}$.