Enhanced Secure Thresholded Data Deduplication Scheme for Cloud Storage

Jan Stanek, *Member, IEEE* and Lukas Kencl, *Member, IEEE*

Abstract—As more corporate and private users outsource their data to cloud storage, recent data breach incidents make end-to-end encryption increasingly desirable. Unfortunately, semantically secure encryption renders various cost-effective storage optimization techniques, such as data deduplication, ineffective. On this ground Stanek et al. [1] introduced the concept of "data popularity" arguing that data known/owned by many users do not require as strong protection as unpopular data; based on this, Stanek et al. presented an encryption scheme, where the initially semantically secure ciphertext of a file is transparently downgraded to a convergent ciphertext that allows for deduplication as soon as the file becomes popular. In this paper we propose an enhanced version of the original scheme. Focusing on practicality, we modify the original scheme to improve its efficiency and emphasize clear functionality. We analyze the efficiency based on popularity properties of real datasets and provide a detailed performance evaluation, including comparison to alternative schemes in real-like settings. Importantly, the new scheme moves the handling of sensitive decryption shares and popularity state information out of the cloud storage, allowing for improved security notion, simpler security proofs and easier adoption. We show that the new scheme is secure under the Symmetric External Diffie-Hellman assumption in the random oracle model.

Index Terms—Security, data protection, deduplication, convergent encryption, cloud storage, popularity

1 Introduction

With the rapidly increasing amounts of data produced worldwide, shared-network multi-user cloud 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 the security perspective, this approach prevents effective application of storage efficiency functions such as compression and deduplication, which would allow storage providers to better utilize their storage back-ends and serve more customers with the same infrastructure.

Data deduplication is a process by which a storage provider only stores a single copy of a file (or of its part) owned by multiple 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 it happens at a block level or at a file level. Client-side data deduplication is more beneficial than server-side since it ensures that multiple uploads of the same content only consume network bandwidth and storage space of a single upload. While security was not part of the early deduplication designs its need soon became imminent with users requiring protection for their data. Convergent encryption, in which the encryption key is derived from the plaintext, seemed like a simple and secure solution allowing deduplication. Unfortunately, it was proven insecure [2]. Moreover, a general impossibility result holds stating that classic semantic security is not achievable for schemes implementing plain convergent encryption [3].

In this paper, we present a scheme that permits a more fine-grained security-to-efficiency trade-off. The intuition is that outsourced data may require different levels of protection, depending on how *popular* the datum 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. This differentiation based on popularity also partly alleviates the user's need to manually sort data as common (deduplicable) and potentially sensitive (non-deduplicable) as every file is first treated as potentially sensitive and thus not deduplicated. Deduplication occurs only when the file becomes popular (i.e., is shared by many users). Note that strictly confidential files that must never be deduplicated should be explicitly marked as non-deduplicable (e.g., by the user or policy) and handled separately.

Around this intuition we build the following contributions: i) we present $\mathcal{E}_\mu$, enhanced threshold cryptosystem that leverages popularity and allows *fine-grained trade-off between security and storage efficiency* and exploit it for the construction of a deduplication scheme, ii) we analyze performance of the proposed deduplication scheme, and iii) we provide analysis of scheme deduplication efficiency based on popularity properties of real datasets. Finally, iv) we discuss the overall security of the proposed scheme and provide ideas for future improvement.

This article differs from our previous paper on this topic [1] in several aspects. We focus more on the practical application of the proposed scheme, providing an entirely new evaluation of the scheme performance together with efficiency analysis that enables potential adopters to easily

• *The authors are with the Faculty of Electrical Engineering, Czech Technical University in Prague, 166 36 Prague 6, Czech Republic. E-mail: {jan.stanek, lukas.kencl}@fel.cvut.cz.*

Manuscript received 19 May 2015; revised 27 July 2016; accepted 16 Aug. 2016. Date of publication 26 Aug. 2016; date of current version 6 July 2018. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2016.2603501

predict the cost and efficiency of scheme deployment in their environments. Scheme internals and their description were modified to better reflect the individual processing steps and to achieve better performance, sparing unnecessary complexity where possible. Specifically, handling of the decryption shares was significantly modified to prevent the attacker from guessing whether two ciphertexts *produced by the same user* correspond to the same plaintext with a non-negligible advantage. This modification also enabled notable simplification of the security analysis. Additionally, removal of the file popularity information from the storage provider database enables the storage provider to handle both popular and unpopular files in the same way.

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 its security. In Section 7 scheme performance is evaluated from the resource consumption point of view and scheme efficiency is evaluated based on dataset parameters. Possible extensions and system limitations are discussed in Section 8, while Section 9 contains our concluding remarks.

2 Related Work

Several deduplication schemes have been proposed by the research community [4], [5], [6] showing how deduplication allows very appealing reductions in the usage of storage resources [7], [8]. Most works did not consider security as a concern for deduplicating systems; recently however, Harnik et al. [2] 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 (PoW) has been introduced [9], [10]. While preventing some attacks, PoW cannot 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. [11] and by Storer et al. [12], 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. [13] formalized convergent encryption under the name *message-locked encryption* (MLE) and proved that it offers confidentiality for unpredictable messages only. As a follow-up work, Bellare et al. [14] presented an enhanced version called interactive MLE providing privacy even for correlated messages depending on the public parameters.

Xu et al. [15] present a PoW scheme allowing client-side deduplication in a bounded leakage setting. They provide security proof in a random oracle model for their solution, but do not address the problem of low min-entropy files.

Bellare et al. present DupLESS [3], 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.

Armknecht et al. present ClearBox [16], a gateway-aided encryption for deduplication storage with built-in PoW and transparent deduplication pattern attestation. Our solution lacks the built-in PoW but offers additional flexibility for potentially sensitive files—if their eventual deduplication is acceptable, the user does not have to treat them differently and they are still strongly protected while unpopular.

Lou et al. [17] try to solve the problem of management of many convergent keys that arises when deduplicating many files or their parts. They offer a viable solution but do not tackle the issue of insecure convergent encryption.

Meye et al. [18] present a two-phase data deduplication scheme trying to solve some of the known convergent encryption issues. While they offer partial solutions including proofs, the proposed methods are often quite inefficient with respect to practical usability.

Duan [19] presents a solution similar to that of DupLESS [3] but instead of a one-component key server he suggests using a threshold scheme for pseudo-convergent key generation. Usage of a threshold component is similar to our proposal, albeit it is being used for a different purpose.

Puzio et al. [20] present ClouDedup, a solution based on convergent encryption strengthened by additional encryption provided by trusted metadata manager. The proposed solution is technically similar to ours, but the metadata manager is quite complex and also responsible for the actual data transfer, which we try to avoid.

Liu et al. [21] present a solution based on password authenticated key exchange protocol to alleviate the need for a trusted third party. While innovative, the solution incurs a notable processing and bandwith overhead and requires users to stay online for most of the time.

Zhao et al. [22] present a scalable deduplication file system. This work demonstrates the trend of introducing deduplication into cloud deployments to achieve better resource utilization. Security is not considered in the system.

3 Overview of the Solution

Deduplication-based systems require solutions tailored to the type of data they are expected to handle [7]. We design our scheme for scenarios where the outsourced dataset contains *few instances of some data items* and *many instances of others*. Concrete examples of such datasets are those generated by backup tools, hypervisors handling linked clones of VM-images, etc.

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

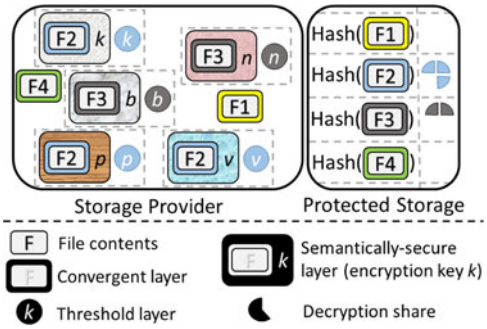

Fig. 1. The multi-layered cryptosystem used in our scheme. Unpopular files (F2 and F3) are protected using two layers, semantically-secure and convergent, whereas popular files (F1 and F4) have the upper layer already removed and are deduplicated.

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. Former category contains, e.g., common blocks of shared VM images, mail attachments sent to many recipients or reused code snippets.

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 Fig. 1: the inner layer is applied using *convergent* encryption, whereas the outer layer is applied using a semantically secure cryptosystem. Uploaders of an unpopular file provide not only the ciphertext but also a *decryption share* usable to reconstruct the key for the upper encryption layer once enough shares are collected. In this way, when sufficient *distinct* copies of an unpopular file have been uploaded, the upper 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 [13], 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 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 PoW [9], [10]) can be applied to secure this step.

There are two further challenges in the secure design of the scheme. First, without proper identity management, Sybil attacks [23] 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 uploads of the same content. In client-side deduplicating systems, this is usually accomplished through an *index* (also called *tag* or *locator*) computed deterministically from the content of the file so that all uploading users compute the same. The client then provides only the index to the storage provider who checks whether he already stores data associated to the same index. If so, data upload is unnecessary. However, the index leaks

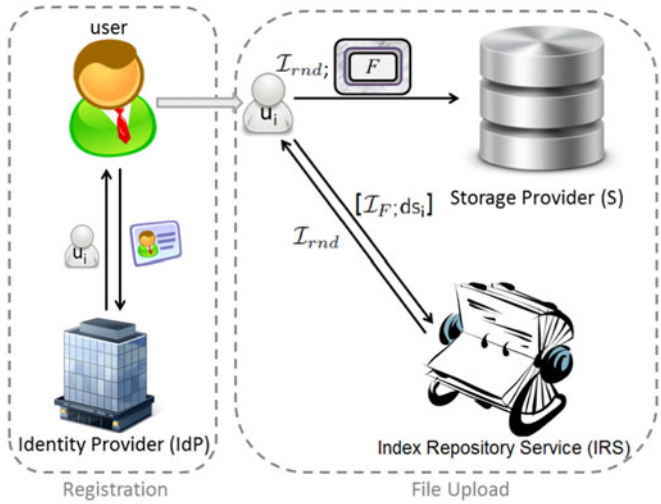

Fig. 2. Illustration of our system model. The schematic shows the main four entities and their interaction for registration and file upload process.

information about the content of the file and therefore violates semantic security which is unacceptable for unpopular files. Deduplication systems using plain convergent encryption compute the index from the convergent ciphertext, thus the same weakness holds.

For these reasons, 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 hinders Sybil attacks, and ii) an index repository service that provides secure indexation for unpopular files.

3.1 System Model

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

The storage provider (S) offers basic storage services and can be instantiated by any storage provider (e.g., Bitcasa, Dropbox, etc.). Users $\mathsf{U}_i \in \mathcal{U}$ 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.

File F is identified within S via a unique file identifier $\mathcal{I}_F$ of bitsize κ ($|\mathcal{I}_F| = \kappa$) computed by a collision-resistant indexing function $\mathcal{I}$: $\{0,1\}^* \rightarrow \{0,1\}^\kappa$. $\mathcal{I}_F$ is issued by the index repository service IRS during the file upload process. The IRS also maintains a record of how many distinct users have uploaded a file and *newly*, compared to [1], stores also the associated decryption shares.

3.2 Security Model

The objective of our scheme is the confidentiality of user content. Specifically, two different security notions, depending on the nature of each datum, are achieved: i) *Semantic security* [24] for unpopular data; ii) *Conventional convergent security* [13] for popular data. Note that integrity and data origin authentication exceed the scope of this work.

In our model, the storage provider S is honest but curious (HBC)—he is trusted to reliably store data on behalf of users and make it available to any user upon request but might be interested in compromising the confidentiality of user content or controlled by the adversary. We also assume

$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathcal{I}_F, \mathsf{U}_i$
S: **if**($\mathsf{DB_S}[\mathcal{I}_F] \neq \emptyset$)
  $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} \leftarrow \mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} \cup \{\mathsf{U}_i\}$
else
$\mathsf{U}_i \longleftarrow \mathsf{S}$: provide file contents
$\mathsf{U}_i \longrightarrow \mathsf{S}$: F
S: $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{data} \leftarrow F$; $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} \leftarrow \{\mathsf{U}_i\}$

Fig. 3. The Put$(\mathcal{I}_F, \mathsf{U}_i, F)$ algorithm.

that the adversary can control up to $n_{\mathcal{A}}$ users and that the goal of the adversary 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 $t \geq p_{\text{lim}} + n_{\mathcal{A}}$. 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 an unpopular file of its choice to become popular. A file shall therefore be declared *popular* when at least 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 8 discusses how this restriction can be lifted.

The IRS and IdP are assumed to be trusted and to abide by the protocol specifications. If either of these components gets compromised by the adversary then the security of all user content is degraded to standard conventional convergent security (i.e., the semantic security of unpopular files is lost). We provide more detailed analysis of potential corruption consequences in Section 6 and discuss the limitation of the trusted IRS in Section 8.

4 Building Blocks

Modeling Deduplication. We describe the interactions between a storage provider (S) that uses deduplication and a set of users ($\mathcal{U}$) 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 $\mathcal{I}$ defined in Section 3.1. The storage provider's backend is modeled as an associative array $\mathsf{DB_S}$ mapping indexes produced by $\mathcal{I}$ to records of arbitrary length i.e., $\mathsf{DB_S}[\mathcal{I}_F]$ is the record mapped to the index of file F. Each record contains two fields, $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{data}$ and $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{users}$. The first contains the content of file F, the second is a list of users that have so far uploaded F. The storage provider and users interact using the following user-invoked algorithms: – Put$(\mathcal{I}_F, \mathsf{U}_i, F)$[Fig. 3], Get$(\mathcal{I}_F, \mathsf{U}_i)$[Fig. 4] and Delete$(\mathcal{I}_F, \mathsf{U}_i)$[Fig. 5]. For simplicity, we do not consider Put failure (e.g., storage full, request timeout etc.) and we implicitly expect all Put invoking functions to abort in such an occasion.

$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathcal{I}_F, \mathsf{U}_i$
S: **if**($\mathsf{U}_i \in \mathsf{DB_S}[\mathcal{I}_F].\mathsf{users}$)
$\mathsf{U}_i \longleftarrow \mathsf{S}$: **return** $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{data}$
else
$\mathsf{U}_i \longleftarrow \mathsf{S}$: **return** error

Fig. 4. The Get$(\mathcal{I}_F, \mathsf{U}_i)$ algorithm.

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 λ). K takes λ 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 c to produce m.

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 λ) (K, E, D). The two main differences with respect to $\mathcal{E}$ is that i) these algorithms are not probabilistic and ii) keys generated by K are a deterministic function of the cleartext message m; we then refer to keys generated by $\mathcal{E}_c$.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 [25]. Moreover, according to the security properties of threshold cryptosystems it is computationally infeasible to perform these operations with fewer than t correct shares. In our scheme we use a modification of a *threshold public-key cryptosystem*. A threshold public-key cryptosystem $\mathcal{E}_t$ is defined as a tuple (Setup, Encrypt, DShare, Decrypt), consisting of four probabilistic polynomial-time algorithms (in terms of a security parameter λ) with the following properties:

$\mathcal{E}_t$.Setup$(\lambda, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk}, \mathsf{S})$: generates the public key of the system pk, the corresponding private key sk and a set $\mathsf{S} = \{(\mathsf{r_i}, \mathsf{sk_i})\}_{i=1}^{n}$ 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 do not need to be secret.

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

$\mathcal{E}_t$.DShare$(\mathsf{r_i}, \mathsf{sk_i}, c) \rightarrow (\mathsf{r_i}, \mathsf{ds_i})$: takes as input a ciphertext c and a key share $\mathsf{sk_i}$ with its index $\mathsf{r_i}$ and produces a decryption share $(\mathsf{r_i}, \mathsf{ds_i})$.

$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathcal{I}_F, \mathsf{U}_i$
S: **if**($\mathsf{U}_i \in \mathsf{DB_S}[\mathcal{I}_F].\mathsf{users}$)
  $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} \leftarrow \mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} \setminus \{\mathsf{U}_i\}$
  if($\mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} = \emptyset$)
    delete the whole $\mathsf{DB_S}[\mathcal{I}_F]$ record
else
$\mathsf{U}_i \longleftarrow \mathsf{S}$: **return** error

Fig. 5. The Delete$(\mathcal{I}_F, \mathsf{U}_i)$ algorithm.

$\mathcal{E}_t$.Decrypt$(c, \mathsf{S}_t) \rightarrow (m)$: takes as input a ciphertext c, a set $\mathsf{S}_t = \{(\mathsf{r_i}, \mathsf{ds_i})\}$ of t pairs of decryption shares and indexes (e.g., $|S_t| = t$), and outputs the cleartext message m.

5 Proposed Scheme

In this section we describe a scheme based on the concept of popularity introduced in Section 3. Our goal is to provide a scheme guaranteeing semantic security for unpopular data (deduplication forbidden), and, transparently transitioning to convergent security offerings as soon as a file becomes popular (deduplication enabled). We first present the cryptosystem that forms the core of our proposed scheme. Next we discuss the role of the identity provider IdP and index repository service IRS and finally we present the scheme as a whole and describe the algorithms it is composed of.

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

$\mathcal{E}_\mu$ is a special-purpose threshold cryptosystem that allows all users to encrypt arbitrary messages m of fixed length λ associated with some label ℓ in such a way that once enough (more than some threshold) of the users provide their decryption shares (created using the same label ℓ), all the messages associated with ℓ can be decrypted. Differently from classic threshold cryptosystems described in Section 4, the Encrypt interface now includes the added label ℓ and the decryption process is designed to be non-interactive. Non-interactivity requires modification of the DShare interface—the decryption share is created using the label l instead of using the ciphertext and is stored in some repository until required for decryption.

For the purpose of this cryptosystem as well as in the remainder of this paper we will make use of $\Lambda = \{\lambda, q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g, \bar{g}, \hat{e}\}$ where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are pairing groups satisfying the Symmetric eXternal Diffie-Hellman (SXDH) assumption [26]; $\mathbb{G}_1 = \langle g \rangle$, $\mathbb{G}_2 = \langle \bar{g} \rangle$ are of prime order q, and $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an efficiently computable, non-degenerate bilinear pairing. SXDH requires the decisional Diffie-Hellman problem (DDH) to be intractable in both $\mathbb{G}_1$ and $\mathbb{G}_2$. Bitsize of q is determined by the security parameter λ which corresponds to the bitlength of the exploited symmetric encryption scheme key (for 128-bit security one would set λ to 128 and bitsize of q two times larger i.e., $|q| = 256$ as recommended in the literature [27], [28]). We will also use two cryptographic hash functions $H_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $H_2 : \mathbb{G}_T \rightarrow \{0,1\}^\lambda$, a semantically secure symmetric cryptosystem $\mathcal{E}$ and a convergent encryption scheme $\mathcal{E}_c$.

$\mathcal{E}_\mu$.Setup$(\lambda, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk}, \{(\mathsf{r_i}, \mathsf{sk_i})\}_{i=1}^n)$: at first, Λ is generated as described above. Next, let secret key $\mathtt{sk} \leftarrow_R \mathbb{Z}$ and generate key shares $\{(\mathsf{r_i}, \mathsf{sk_i})\}_{i=1}^n$ such that any set of t shares can be used to reconstruct $\mathsf{s}k$ [25]. Also, let $\bar{g}_{\text{pub}} \leftarrow \bar{g}^{\mathsf{sk}}$. Public key pk is set to $\{\Lambda, H_1, H_2, \bar{g}_{\text{pub}}\}$

$\mathcal{E}_\mu$.Encrypt$(\mathsf{pk}, \ell, m) \rightarrow (c)$: Let $r \leftarrow_R \mathbb{Z}$ and let $E \leftarrow \hat{e}(H_1(\ell), \bar{g}_{\text{pub}})^r$. Next, set $c_1 \leftarrow H_2(E) \oplus m$, $c_2 \leftarrow \bar{g}^r$. Compose the ciphertext c as (c_1, c_2).

$\mathcal{E}_\mu$.DShare$(\mathsf{r_i}, \mathsf{sk_i}, \ell) \rightarrow (\mathsf{r_i}, \mathsf{ds_i})$: let $\mathsf{ds_i} \leftarrow H_1(\ell)^{\mathsf{sk_i}}$.

$\mathcal{E}_\mu$.Decrypt$(c; \mathsf{S}_t = \{(\mathsf{r_i}, \mathsf{ds_i})\}) \rightarrow (m)$: parse c as (c_1, c_2). Using all decryption shares in S_t compute

$$\prod_{(\mathsf{r_i}, \mathsf{ds_i}) \in \mathsf{S}_t} \mathsf{ds_i}^{\lambda_{0,r_i}^{\mathsf{S}_t}} = \prod_{(\mathsf{r_i}, \mathsf{sk_i}) \in \mathsf{S}'_t} H_1(\ell)^{\mathsf{sk_i} \lambda_{0,r_i}^{\mathsf{S}_t}} = H_1(\ell)^{\sum_{(\mathsf{r_i}, \mathsf{sk_i}) \in \mathsf{S}'_t} \mathsf{sk_i} \lambda_{0,\mathsf{r}_i}^{\mathsf{S}_t}} = H_1(\ell)^{\mathsf{sk}},$$

where $\lambda_{0,\mathsf{r}_i}^{\mathsf{S}_t}$ are the Lagrangian coefficients of the polynomial with interpolation points from the set $\mathsf{S}'_t = \{(\mathsf{r_j}, \mathsf{sk_j})\}_{j=0}^{t-1}$. Note that sk cannot be reconstructed from neither the decryption shares nor from $H_1(\ell)^{\mathsf{sk}}$.

Compute $\hat{E}$ as $\hat{e}\left(H_1(\ell)^{\mathsf{sk}}, c_2\right)$ and $m = c_1 \oplus H_2(\hat{E})$.

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

$$\forall x : \hat{e}(H_1(\ell)^x, \bar{g}^r) = \hat{e}(H_1(\ell), \bar{g})^{rx} = \hat{e}(H_1(\ell), \bar{g}^x)^r.$$

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 live participation of the entities that executed the $\mathcal{E}_\mu$.DShare algorithm; ii) It mimics convergent encryption in that the decryption shares are deterministically dependent on the plaintext label. 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 $\mathcal{E}_\mu$.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 $\mathcal{E}_\mu$.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 Scheme Participants

Apart from typical deduplication scheme participants—the user wanting to store his data remotely and the storage provider offering his service and wanting to benefit from deduplication, our scheme requires the presence of two additional entities—the identity provider IdP and the index repository service IRS.

IdP serves as the identity authority as well as the trusted dealer of the secret shares of the $\mathcal{E}_\mu$ master secret. Upon scheme deployment, IdP is responsible for execution of $\mathcal{E}_\mu$.Setup. Each user interacts with IdP only once, when joining the scheme, and the interaction only includes IdP ensuring that the user is new (i.e., hasn't participated in the scheme yet) and providing the identity credentials and secret share. Security-wise, IdP is used to hinder exploitation of the threshold cryptosystem $\mathcal{E}_\mu$ by means of Sybil attacks [23] and master secret leakage.

IRS serves as a secure index generator and decryption share storage. To store data, IRS maintains an associative array $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$ with three fields, $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.ctr, $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.idxes and $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_F]$.dshares. The array is initialized empty and the records are added according to GenSecIdx implementation (Fig. 6). The associative array is indexed by $\mathcal{I}_F$ defined in Section 3.1. $\mathcal{I}_F$ is then disclosed to the storage

IRS: **if** ($\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.ctr $\geq t$)
IRS $\longrightarrow$ U_i: **return** $\mathcal{I}_{F_c}$
IRS: $\mathcal{I}_{\mathrm{rnd}} \leftarrow \mathtt{PRF}(\sigma, \mathsf{U}_i || \mathcal{I}_{F_c})$
if ($\mathcal{I}_{\mathrm{rnd}} \notin \mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.idxes)
increment $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.ctr
add $\mathcal{I}_{\mathrm{rnd}}$ to $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.idxes
add $(\mathsf{r_i}, \mathsf{ds_i})$ to $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.dshares
if ($\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.ctr $= t$)
IRS $\longrightarrow$ S: Deduplicate($\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.idxes, $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.dshares)
IRS $\longrightarrow$ U_i: **return** $\mathcal{I}_{\mathrm{rnd}}$

Fig. 6. The GenSecIdx$(\mathcal{I}_{F_c}, (\mathsf{r_i}, \mathsf{ds_i}))$ algorithm. Popularity is evaluated by comparison of per-index counter and threshold t. Behaviour corresponding to a popular file index is highlited in green (the lightest color), unpopular file index part is in blue (the darkest color) and part corresponding to popularity switch is in red.Note that the last line is common to both unpopular and switching state situations. Deduplicate implementation is available in Section 5.3.

provider S during Put. Being generated deterministically, $\mathcal{I}_F$ leaks information about file contents (notice that convergent encryption does not fix this leakage since the encryption is deterministic too). IRS is used to prevent this leakage by offering its secure index generation service GenSecIdx (Fig. 6) to the user—for a (leaky) convergent index $\mathcal{I}_{F_c}$ the user is given a random index $\mathcal{I}_{\mathrm{rnd}}$. Note that if the user invokes GenSecIdx more times with the same $\mathcal{I}_{F_c}$ he will always get the same index $\mathcal{I}_{\mathrm{rnd}}$ and the popularity counter will not increase. This prevents potentially corrupted user to force state transition of an unpoular file. Another, albeit much more limited, leakage could occur through the decryption shares used in $\mathcal{E}_\mu$-repeated encryption of message m using the same ℓ and pk by user U_i produces equal decryption shares, thus leaking equality of the associated ciphertexts. To prevent this leakage, instead of storing the decryption share together with the ciphertext to S, we store it separately to IRS. To suit the needs of our scheme, the decryption share store request is grouped with the secure index generation request in GenSecIdx. Additionally, if the stored decryption share was not yet used in the decryption process, the user is allowed to delete it from IRS via RemDShare (Fig. 7).

Implementation-wise, IRS uses a Pseudo-Random Function (PRF) that takes a concatenation of the requesting user identity U_i and convergent index $\mathcal{I}_{F_c}$ on the input (domain), uses a secret seed σ of length λ (key) and produces a random index $\mathcal{I}_{\mathrm{rnd}}$ (range) i.e., $\mathtt{PRF} : \{0,1\}^\lambda \times \{0,1\}^{|\mathsf{U_i}|+\kappa} \rightarrow \{0,1\}^\kappa$. σ is generated upon IRS instantiation once and used in each invocation of PRF, to assure that same input always generates same output.

5.3 Storage Scheme

We formally introduce our scheme, detailing the interactions between a set of n clients perceived as users U_i, a storage provider S and the two trusted entities, the identity provider IdP and the index repository service IRS.

IRS: **if**($\mathcal{I}_{\mathrm{rnd}} \in \mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$.idxes)
with $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}]$ **do**
.idxes $\leftarrow$ .idxes $\setminus \{\mathcal{I}_{\mathrm{rnd}}\}$
.dshares $\leftarrow$ .dshares $\setminus (\mathsf{r_i}, \mathsf{ds_i})$
.ctr $\leftarrow$ .ctr $- 1$

Fig. 7. The $RemDShare(\mathcal{I}_{F_c}, \mathcal{I}_{\mathrm{rnd}}, \mathsf{r_i})$ algorithm.

IRS			
index	ctr	idxes	dshares
1A35BC127CC36958	50		
82090A161718192A	1	A112927132910012	(1302;1A85227..)

S		
index	data	owners
1A35BC127CC36958	1B100510955476AC4125..	0015,0098,1023,..
A112927132910012	DD845A3362C5487FF14..	1302

Fig. 8. Examples of S and IRS records.

S is modeled as an indexed associative array $\mathsf{DB_S}$ supporting the Put, Get and Delete operations same as the deduplication model described in Section 4. IRS is modeled as described in Section 5.2. Examples of the the records maintained by S and the records of IRS are available in Fig. 8.

Recall that $\mathcal{E}$ and $\mathcal{E}_c$ are a symmetric cryptosystem and a convergent encryption scheme, respectively (see Section 4). $\mathcal{E}_\mu$ is our convergent threshold cryptosystem.

When a new client wants to join the scheme, she contacts IdP in a secure way. IdP verifies her identity; upon successful verification, it issues user credentials U_i and a secret key share $\mathsf{s}k_i$ (generating a brand new $\mathsf{sk_i}$ if necessary). From this point onwards, the client becomes the user U_i towards the scheme.

For simplicity and clarity, the core API offers only three user-invoked algorithms—Upload to put data into the storage, Download to get data from the storage and Remove to erase data from the storage. Due to complexity and limited length we intentionally do not provide extended API, but any functionality achievable in classic deduplication schemes should be achievable in our scheme too since the scheme design does not generally limit the functionality.

The initial deployment of the scheme starts with the Init algorithm:

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

The init algorithm is run only once throughout the whole lifetime of the scheme. Invoking Init again corresponds to deployment of a new scheme.

Upload(F, U_i)[Fig. 9]: The user U_i encrypts file F convergently and generates the index $\mathcal{I}_{F_c}$ which he uses in request to IRS.

U_i: $k_c \leftarrow \mathcal{E}_c.\mathsf{K}(F)$
$F_c \leftarrow \mathcal{E}_c.\mathsf{E}(k_c, F)$
$\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
$(\mathsf{r_i}, \mathsf{ds_i}) \leftarrow \mathcal{E}_\mu.\mathsf{DShare}(\mathsf{r_i}, \mathsf{sk_i}, F_c)$
$\mathsf{U}_i \longrightarrow$ IRS: $\mathcal{I}_{\mathrm{ret}} \leftarrow \mathsf{GenSecIdx}(\mathcal{I}_{F_c}, (\mathsf{r_i}, \mathsf{ds_i}))$
U_i: **if**($\mathcal{I}_{\mathrm{ret}} = \mathcal{I}_{F_c}$)
$\mathsf{U}_i \longrightarrow$ S: Put$(\mathcal{I}_{F_c}, \mathsf{U}_i, F_c)$
U_i: $F \leftarrow (k_c, \mathcal{I}_{F_c})$
else
U_i: $k \leftarrow \mathcal{E}.\mathsf{K}()$; $c \leftarrow \mathcal{E}.\mathsf{E}(k, F_c)$
$c_\mu \leftarrow \mathcal{E}_\mu.\mathsf{Encrypt}(\mathsf{pk}, F_c, k)$
$F' \leftarrow (c, c_\mu)$
$\mathsf{U}_i \longrightarrow$ S: Put$(\mathcal{I}_{\mathrm{ret}}, \mathsf{U}_i, F')$
U_i: $F \leftarrow (k, \mathcal{I}_{\mathrm{ret}}, k_c, \mathcal{I}_{F_c})$

Fig. 9. The Upload(F, U_i) algorithm. Popular file upload part is highlited in green (lighter color), unpopular file upload part in blue (darker color).

U_i: **if**$(F = (k, \mathcal{I}_{\text{ret}}, k_c, \mathcal{I}_{F_c}))$
$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\text{ret} \leftarrow \mathsf{Get}(\mathcal{I}_{\text{ret}}, \mathsf{U}_i)$
U_i: **if**$(\text{ret} \neq \text{error})$
$\text{ret} \rightarrow (c, c_\mu)$; $F_c \leftarrow \mathcal{E}.\mathsf{D}(k, c)$
$F \leftarrow \mathcal{E}_c.\mathsf{D}(k_c, F_c)$
else
$F \leftarrow (k_c, \mathcal{I}_{F_c})$; $\mathsf{Download}(F, \mathsf{U}_i)$
else
$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\text{ret} \leftarrow \mathsf{Get}(\mathcal{I}_{F_c}, \mathsf{U}_i)$
U_i: **if**$(\text{ret} = \text{error})$
download failed
else
$\text{ret} \rightarrow F_c$; $F \leftarrow \mathcal{E}_c.\mathsf{D}(k_c, F_c)$

Fig. 10. The $\mathsf{Download}(F, \mathsf{U}_i)$ algorithm. Unpopular file download part is highlited in blue (the darkest color), part corresponding to file switch caused by the file being uploaded as unpopular but changed status to popular before download is in red and popular file download part is in green (the lightest color).

If IRS returns the index unchanged then the file is already popular and the following Put operation only adds the invoking user into the list of owners for file F_c, no data upload occurs and the user stores the index $\mathcal{I}_{F_c}$ and the convergent key k_c to be able to download his file in the future.

If IRS returns a different index $\mathcal{I}_{\text{rnd}}$ then the file is unpopular and it is necessary to encrypt the convergent ciphertext again, using a random key k and encrypt the random key k using the convergent threshold cryptosystem with label F_c. The doubly-encrypted file together with the encrypted random key are then transferred to the storage during the Put operation. This way, the unpopular file will be semantically secure until there are more than t of its copies uploaded. The user stores the index obtained from the IRS and the random key together with the convergent key and the convergent index.

Note that additional strengthening measures can be deployed at S to improve the security provided (such as PoW [9] that allow checking, whether the user really owns the popular file he tries to upload).

$\mathsf{Download}(F, \mathsf{U}_i)$ [Fig. 10]: If the user uploaded the file as unpopular (i.e., $F = (k, \mathcal{I}_{\text{ret}}, k_c, \mathcal{I}_{F_c})$) he first tries to get it from $\mathcal{I}_{\text{ret}}$. If he succeeds, he can decrypt the unpopular content to recover his file. If he fails, the file must have gotten popular in the meantime so he replaces F (i.e., $F = (k_c, \mathcal{I}_{F_c})$) and retries the download. If the file is popular, the user gets the popular content from $\mathcal{I}_{F_c}$ and he can decrypt it to recover his file.

$\mathsf{Remove}(F, \mathsf{U}_i)$ [Fig. 11]: If the file is unpopular, the user first tries to delete it as unpopular (i.e., invokes $\mathsf{Delete}(\mathcal{I}_{\text{ret}}, \mathsf{U}_i)$) and, if he succeeds, he additionally requests removal of his decryption share from the IRS database. If he fails then the file got popular in the meantime and he deletes it as popular. Popular file deletion is straight invocation of $\mathsf{Delete}(\mathcal{I}_{F_c}, \mathsf{U}_i)$.

$\mathsf{Deduplicate}(\mathsf{idxes}, \mathsf{dshares})$ [Fig. 12]: With the deduplication request from IRS, S receives t indexes and t decryption shares. S checks that records for all provided indexes exist and if not, it waits (synchronization purpose, waiting for the last copy of F to be uploaded). S recovers the decryption keys using the decryption shares and decrypts all the data contents of all the indexes provided in the notification (for performance reasons, decryption of a subset of indexes is preferable). As a result, S ideally obtains t equal convergent ciphertexts, stores the ciphertext to a new record under the convergent index $\mathcal{I}_{F_c}$ and deletes all the now-excessive copies. If the convergent ciphertexts differ then some of the uploading users must have cheated S aborts deduplication and leaves the stored files unmodified. Optionally, S could notify IRS that deduplication failed and allow it to collect additional decryption shares before trying deduplication again.

U_i: **if**$(F = (k, \mathcal{I}_{\text{ret}}, k_c, \mathcal{I}_{F_c}))$
$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\text{ret} \leftarrow \mathsf{Delete}(\mathcal{I}_{\text{ret}}, \mathsf{U}_i)$
U_i: **if**$(\text{ret} \neq \text{error})$
$\mathsf{RemDShare}(\mathcal{I}_{F_c}, \mathcal{I}_{\text{ret}}, \mathsf{r_i})$
else
$F \leftarrow (k_c, \mathcal{I}_{F_c})$; $\mathsf{Remove}(F, \mathsf{U}_i)$
else
$\text{ret} \leftarrow \mathsf{Delete}(\mathcal{I}_{F_c}, \mathsf{U}_i)$
if$(\text{ret} = \text{error})$
remove failed

Fig. 11. The $\mathsf{Remove}(F, \mathsf{U}_i)$ algorithm. Unpopular file remove part is highlited in blue (the darkest color), part corresponding to file switch caused by the file being uploaded as unpopular but changed status to popular before removal is in red and popular file remove part is in green (the lightest color).

6 Security Analysis

First, we formally analyze the security of the core building block of our scheme—the $\mathcal{E}_\mu$ cryptosystem. Then we analyze the view of each scheme participant and discuss its corruptability as well as the possible information leakage caused by collusion of participants. Finally, we provide comparison of security properties of our scheme with other current secure deduplication proposals [3], [16], [20], [21]. Differently from the previous version of the paper [1], the decryption share repository has been moved from S to IRS. This allows for conceptually easier security analysis and groups the possible plaintext-related leakages that stem from both the convergent indexes and the decryption shares.

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

To define and analyze the security of $\mathcal{E}_\mu$ we use a straightforward adaptation of the IND-CPA experiment (INDistinguishability under Chosen Plaintext Attack), henceforth referred to as IND_μ-BCPA (B for Bounded). The experiment requires the adversary to declare upfront the set of users to be corrupted, similarly to selective security [29].

S: $\mathcal{F} \leftarrow \emptyset$; $\mathcal{U} \leftarrow \emptyset$
foreach$(\mathcal{I}_i \in \mathsf{idxes})$
$(c, c_\mu) \leftarrow \mathsf{DB_S}[\mathcal{I}_i].\mathsf{data}$
$K \leftarrow \mathcal{E}_\mu.\mathsf{Decrypt}(c_\mu, \mathsf{dshares})$
$F_c \leftarrow \mathcal{E}.\mathsf{D}(k, c)$
$\mathcal{F} \leftarrow \mathcal{F} \cup \{F_c\}$; $\mathcal{U} \leftarrow \mathcal{U} \cup \mathsf{DB_S}[\mathcal{I}_i].\mathsf{users}$
forall$(F_c \in \mathcal{F})$ check equality; fail $\rightarrow$ abort
$\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
execute $\mathsf{Put}(\mathcal{I}_{F_c}, \mathcal{U}, F_c)$
delete all records indexed by idxes

Fig. 12. The $\mathsf{deduplicate}(\mathsf{idxes}, \mathsf{dshares})$ algorithm.

Informally, in IND_μ-BCPA, 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 $n_\mathcal{A} < t - p_{\lim} - 1$ of predeclared users, and obtains their secret keys through an oracle $\mathcal{O}_{\mathsf{C}corrupt}$. At the end of the game, the adversary outputs a message m_* and label ℓ_*; 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 $\mathsf{b}' = \mathsf{b}$. Formally, the security of $\mathcal{E}_\mu$ is defined through the IND_μ-BCPA experiment between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, given a security parameter λ:

Setup Phase. $\mathcal{C}$ executes the $\mathcal{E}_\mu$.Setup algorithm with λ, and generates a set of user identities $\mathsf{U} = \{\mathsf{U}_i\}_{i=0}^{n-1}$. 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 $|\mathsf{U}_\mathcal{A}| = n_\mathcal{A} < t - p_{\lim} - 1$ identities of users that will later on be subject to $\mathcal{O}_{\mathsf{C}corrupt}$ 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 $\mathcal{O}_{\mathsf{C}corrupt}$ and receive the secret key share that corresponds to any user $\mathsf{U}_i \in \mathsf{U}_\mathcal{A}$.

Challenge Phase. $\mathcal{A}$ picks the challenge message m_* and label ℓ_* and sends it to $\mathcal{C}$. $\mathcal{C}$ chooses at random (based on a coin flip b) whether to return the encryption of m_* i.e., $\mathcal{E}_\mu.\mathsf{Encrypt}(\mathsf{pk}, \ell_*, m_*)$ $(\mathsf{b} = 1)$, 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 lemma shows that IND_μ-BCPA is guaranteed in $\mathcal{E}_\mu$ as long as the SXDH problem is intractable [26].

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

6.2 Security Analysis of the Scheme

In Section 3 we claimed that our scheme provides convergent security for popular files and semantic security for unpopular files under the assumption that IRS and IdP are trusted and there are no more than $n_\mathcal{A}$ corrupted users. Since convergent security was formally analyzed by Bellare et al. [13] and our scheme implements classic convergent encryption for popular files, we focus on the analysis of semantic security for unpopular files.

Claim. Adversary $\mathcal{A}$ cannot break semantic security of any unpopular file F if he can only corrupt the storage provider S and a set of users $\mathcal{U}$ where $|\mathcal{U}| \leq n_\mathcal{A}$ and none of the corrupted users owns F.

Proof Sketch. $\mathcal{A}$ can obtain any unpopular file record from S (an intrinsic property of unpopular files is to be associated with a single user in the storage provider database DB_S). Let the obtained record be indexed $\mathcal{I}$ contaning data $\mathsf{DB}_\mathsf{S}[\mathcal{I}].\mathsf{data} = (c, c_\mu)$. Since $\mathcal{I}$ was obtained by PRF using the secret seed and input unknown to $\mathcal{A}$, it does not leak any information. c was obtained using a semantically secure cryptosystem $\mathcal{E}$ and c_μ was obtained by cryptosystem $\mathcal{E}_\mu$ guaranteeing IND_μ–BCPA, both also guaranteeing no leakage. $\mathcal{A}$ cannot use corrupted users $\mathcal{U}$ to enforce deduplication since $|\mathcal{U}| \leq n_\mathcal{A}$ and the deduplication threshold was defined as $t \geq p_{\lim} + n_\mathcal{A}$ and cannot invoke deduplication in S since he does not have the required input (i.e., index mapping and decryption shares). Having no other information conduit, $\mathcal{A}$ cannot break semantic security of unpopular file F.

Maintaining semantic security of unpopular files even in presence of corrupted S and $n_\mathcal{A}$ users is a good property, but requires very strong assumption that the adversary does not corrupt IdP or IRS. To analyze the situation where an adversary could corrupt these entities, we provide the view of both entities with respect to unpopular files and discuss consequences of their corruption.

The identity provider IdP does not have any information about files, however, he can generate any number of legitimate scheme users on demand and knows the master secret sk of the $\mathcal{E}_\mu$ cryptosystem. If the adversary $\mathcal{A}$ is allowed to corrupt IdP, he can exploit the user-generating process to spawn more than $n_\mathcal{A}$ corrupted users and thus enforce deduplication of any file for which he knows $H_1(F_c)$. Interestingly, if the adversary only chooses some record of an unpopular file copy from S, he cannot force that particular file to become popular since he has no way of obtaining $H_1(F_c)$ from $\mathcal{I}$ nor can he decrypt c_μ (and, consequently, c) since he does not know the label used during encryption (i.e., F_c). The only way would be to enforce all files to become popular, which is infeasible.

The index repository service IRS is security-wise the weakest point of our scheme. Since IRS knows the mapping of convergent index $\mathcal{I}_{F_c}$ to the random index $\mathcal{I}_{\mathrm{rnd}}$ for every unpopular file together with the decryption shares (thus knowing a set of indexes $\{\mathsf{r}_i\}$ corresponding to the uploaders, but not their identities $\{\mathsf{U}_i\}$). Therefore, if $\mathcal{A}$ corrupts IRS, security of all unpopular files is degraded from semantic to convergent (since $\mathcal{A}$ can bruteforce $\mathcal{I}_{F_c}$). We further discuss the limitation of trusted IRS in Section 8.

Compared to other current secure deduplication proposals security-wise, our scheme is the only one that implicitly allows corruption of the storage provider and up to $n_\mathcal{A}$ users without compromising semantic security of unpopular files, but also exhibits the single point of failure vulnerability by requiring participation of a trusted IRS. DupLESS [3] uses a key server similar to our IRS but with better resilience (it does not store nor see the convergent indexes), yet it cannot prevent the honest-but-curious storage provider to check for file equality nor attacks based on knowledge of the hash of the plaintext file. ClearBox [16] uses gateway G similar to IRS. While G can prevent known-hash-based attacks (thanks to PoW), it cannot prevent attacks based on the known (guessed) content—a user may learn whether a file was stored before and if so, the Attest procedure eventually lets him know approximately how many users stored the file. Adding high-entropy string to low-entropy files to counter these attacks (as suggested in [16]) is required and will work for strictly confidential files (in our solution too, yet at a higher storage cost), but is

rather tedious. For unpopular potentially sensitive files, where deduplication is acceptable if they get popular, IRS protects against this type of attack. Additionally, G has access to the encrypted files (and thus knows also the file size). Liu et al. [21] present a scheme that does not require the trusted component, but it also cannot prevent an HBC S to check for file equality and is prone to user collusion attacks. Our scheme prevents the listed attacks for unpopular files. ClouDedup [20] isolates the storage provider from the deduplication procedure completely, not leaking any information, yet introduces two quite complex trusted components to achieve it.

7 Performance Evaluation

Our scheme was designed to provide more fine-grained trade-off between security and space efficiency compared to the standard deduplication scheme exploiting convergent encryption [11]. Specifically, deduplication efficiency is decreased for the benefit of increased security of the unpopular files. This section evaluates the scheme storage space reduction efficiency as well as the overhead incurred by the scheme, and provides comparison with other current secure deduplication schemes [3], [16], [20], [21].

7.1 Storage Space Reduction Definition

Deduplication schemes typically store the actual file contents encrypted in the storage provider space (S) and keep various metadata (e.g., indexes, keys) in the user local storage and in some assistive entity (e.g., key server, IRS). Since metadata must be marginal in size for deduplication to make sense, the efficiency indicator focuses on the file content stored by S.

For a deduplication scheme, the deduplication efficiency is defined in the form of the space reduction ratio $\mathsf{SRR} = \textit{bytes in}/\textit{bytes out}$ [7] i.e., size of the dataset before deduplication divided by its size after deduplication. For perfect deduplication schemes ([3], [16], [20]) it holds that every file F is stored only once after deduplication i.e., per-file space reduction is the file popularity p_F (for file F of size $|F|$, the reduction is $p_F \times |F| \rightarrow |F|$). Imperfect deduplication schemes depend on various settings and properties. The scheme by Liu et al. [21] depends mainly on the user offline rate, for our scheme, the space reduction occurs only for popular files i.e., files where $p_F \geq t$, whereas there is no space reduction for unpopular files (reminder from Section 3.2, $t \geq p_{\text{lim}} + n_{\mathcal{A}}$). The efficiency of our scheme depends directly on two factors—the value of t (tunable parameter of the scheme) and the popularity distribution of files in the dataset (determined by the dataset, non-settable).

Having a dataset $\mathcal{F} = \{F_i\}_{i=1}^{N}$ where file F_i has popularity p_{F_i}, the SRR of a perfect deduplication scheme is $\mathsf{SRR} = \sum_{i=1}^{N}(|F_i| \times p_{F_i}) / \sum_{i=1}^{N} |F_i|$. To compute the SRR for our scheme, we have to choose t and split the dataset into a set of popular files $\mathcal{F}_p = \{F_i \mid p_{F_i} \geq t\}$ and a set of unpopular files $\mathcal{F}_u = \{F_i \mid p_{F_i} < t\}$. The space reduction ratio is then computed as

$$\mathsf{SRR} = \sum_{i=1}^{N}(|F_i| \times p_{F_i}) \Big/ \left(\sum_{F \in \mathcal{F}_p} |F| + \sum_{F \in \mathcal{F}_u} (|F| \times p_F) \right).$$

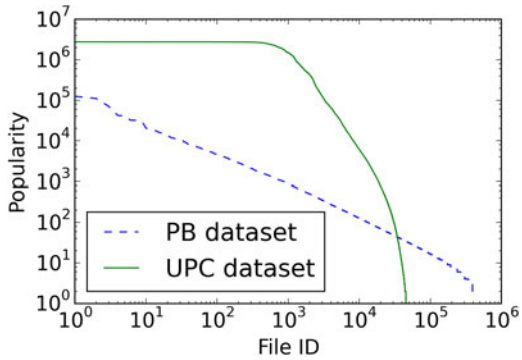

Fig. 13. File popularity distributions in the evaluated datasets presented in the logarithmic scale.

For simpler comparison we define the space reduction percentage as $\mathsf{SRP} = (1 - 1/\mathsf{SRR}) \times 100$.

7.2 Analysis of Space Reduction Efficiency

To test the efficiency of our scheme on real data we use two publicly available datasets—the *PB dataset* comprised of data collected by F. Hecht, T. Bocek and D. Hausheer from the popular BitTorrent tracker Pirate Bay [30], representing an example of user data backup from multiple users, and the *UPC dataset*, similar to the one used by Liu et al. [21], consisting of data provided by the Ubuntu Popularity Contest [31] (snapshot taken on March 15, 2016) and representing an example of a system hard drive backup from multiple users.

The *PB dataset* is a collection composed mostly of audio, video and software. Since no information about torrent contents (i.e., file-level granularity) is provided, we consider each torrent to correspond to one file for the purpose of our measurement (note that this simplification does not positively impact the results—on the contrary, the savings would only be better in case some file was shared among the different torrents). This way, we obtain 679,515 unique files of size ranging from 0 to 224 GB. To compute popularity of each of these files we sum the number of "seeders" i.e., peers already having the whole file and "leechers" i.e., peers having only part of the file at the moment, but intending to get the whole file in near future. The popularity ranges between 0 and 124,975. We remove files with zero size or zero popularity (inactive torrents), getting a dataset consisting of 442,332 unique files with popularity ranging from 1 to 124,975. The dataset contains 10,836,260 files in total (including duplicates) and has total size of 23.149 PB.

The *UPC dataset* represents a collection of Ubuntu software packages including the information about how many users downloaded and installed each package. Using the list [31] and the *apt-cache* command on a Ubuntu 15.10 x86_64 machine, we extract sizes of the packages ranging from 736 B to 1.01 GB, omitting unavailable packages. This way we obtain a dataset consisting of 46,040 unique files with popularity in range from 1 to 2,755,245. There are 3,641,060,666 files total in the dataset with the total size of 2.282 PB.

Popularity distributions of both datasets are shown at logarithmic scale in Fig. 13. To compare the efficiency of our scheme with perfect deduplication schemes ([3], [16], [20], [21]), we provide SRP comparison for both datasets in

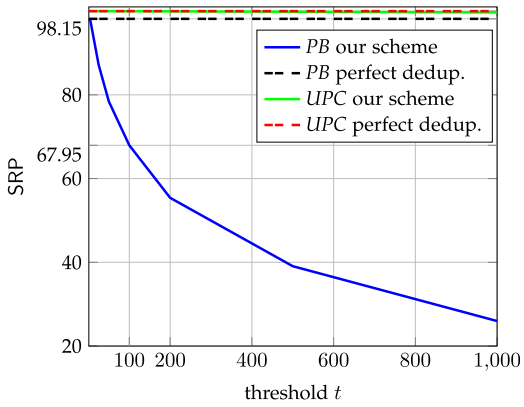

Fig. 14. Space reduction percentage (SRP) comparison of our scheme and perfect deduplication schemes for the *PB dataset* and the *UPC dataset*.

Fig. 14. Results of the scheme by Liu et al. [21] depend mainly on the user offline rate and would be very similar to the results listed for the enterprise and media datasets in their work [21].

As the SRPs demonstrate, our scheme offers very good reduction for the *UPC dataset*, even for quite high values of threshold t (99.68 percent reduction for $t = 1{,}000$) whereas for the *PB dataset* efficiency decreases faster and the reduction capabilities for high values of threshold t are much lower (26 percent reduction for $t = 1{,}000$). The difference in scheme efficiency in the two evaluated datasets is caused by the difference in their popularity distribution. Even though both datasets have the total size in the order of PB, the *PB dataset* contains only two files with popularity larger than 100,000 whereas the *UPC dataset* contains 3,267 such files (Fig. 13). Note that the higher the popularity of a file, the better the reduction by its deduplication.

Using the results of the analysis of the two datasets, we postulate the following observations regarding efficiency of our scheme: i) for datasets containing many files with very high popularity (such as the *UPC dataset*) the efficiency is very good even for quite high values of t; ii) for datasets having the popularity distribution close to a power-law distribution (such as the *PB dataset*) the efficiency is notably worse compared to perfect deduplication for very high values of t, but a compromise between security and efficiency can be found for reasonable values of t (e.g., $\mathsf{SRP} = 67.95$ for $t = 100$ in the *PB dataset*); iii) for datasets having steep long-tailed popularity distribution (i.e., many files with low popularities, only a few files with high popularity) the efficiency of our scheme is poor and it should not be used for such datasets.

7.3 Computation and Communication Cost Analysis

Analysis Setup. To measure consumption of computational and communication resources in practice and provide comparison with other schemes, we implemented a prototype of our scheme, we use the publicly available prototype of DupLESS [3], a prototype kindly provided by Liu et al. [21] and our prototype implementation of ClearBox [16] (authors could not provide their code due to company policies). The Attest procedure of ClearBox was not implemented as neither of the other solutions provides such functionality. To eliminate measurement discrepancies caused by implementation, we prototyped our scheme and ClearBox mostly in Python using DupLESS code as basis and used Python libraries *Crypto* and *hashlib* for cryptographic operations and hashing and a wrapper for the C-implemented PBC library [28]. The prototype by Liu et al. is implemented in Javascript and we used it "as is" with one modification—to be comparable with others, instead of storing files locally at the server, the server uses Dropbox for the actual file storage and internally stores only a hash of the file for comparison purposes. To prevent confusion, we use the term "client" for the client-side application, "server" for the server-side application (i.e., our IRS, gateway in ClearBox, KS in DupLESS, $\mathcal{S}$ in the scheme of Liu et al.) and Dropbox as the cloud storage backend. All programs were tested on an Intel Xeon E3-1220 machine with 4 CPU cores 3.1 GHz, and 16 GB of RAM running Ubuntu 14.04.

To keep prototypes as aligned as possible we set the general bit-security to 128—we use AES-128-CTR as symmetric encryption, SHA256 for hashing and type F bilinear pairing provided by the PBC library [28] for group operations, where applicable. Note that if bit-security 256 or larger is required, we recommend to use newer curves introduced by Aranha et al. [32] since the PBC library tends to get rather slow for such settings. Our scheme specific settings include the bitsize of the order of the exploited groups $|q| = 256$ and threshold $t = 1{,}000$.

To emulate WAN network delay (since we only use one testing server), we use the *tc* Linux command shaping all traffic using a Pareto distribution with mean 20 ms and variance of 4 ms, same as Armknecht et al. [16].

Since scheme initialization and user registration procedures are varied and relatively rare (compared to file manipulation operations), we leave them out of the analysis and comparison. For neither of the tested prototypes these procedures took longer than a few seconds. Instead, we focus mostly on the Put (respectively Upload) operation for deduplicable files (since that is the most important from the practical usage perspective) and compare the costs for the different prototypes and a plain Dropbox service (without deduplication). Afterwards we briefly analyze the Get operation, communication overhead and analysis of its specific deduplication operation cost.

Deduplicable Put *Request*. To compare the prototypes practically we use the *UPC dataset* from Section 7.2. To avoid the initialization period with an empty storage and no deduplication we model a situation where every file F from the dataset was already uploaded approx. $p_F/2$ times. Next, we randomly sample 100 files from the dataset, generate 100 Put requests for these files per each prototype and measure processing time using the Python *time* (respectively JS *Date*) module. Aggregate results plotted in Fig. 15 demonstrate that solutions not interacting with Dropbox for a deduplicated file upload (ClearBox and Liu et al.) have much better results. The outliers suggest a few Put requests taking significantly longer than the others. Interestingly, even though our scheme uses Dropbox only to store a short hash per file, interaction with Dropbox takes similar time as the plain Dropbox solution that always stores the entire file. This suggests that for most files in the sample, connection initiation and request set-up with Dropbox take significantly more time than the actual file transfer.

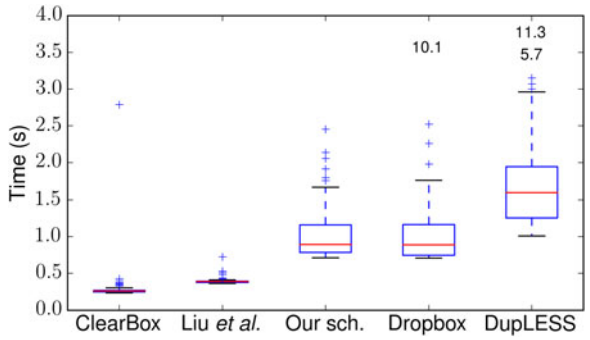

Fig. 15. Average processing time of a Put request for a deduplicable file from a randomly chosen 100-file sample from the *UPC dataset*.

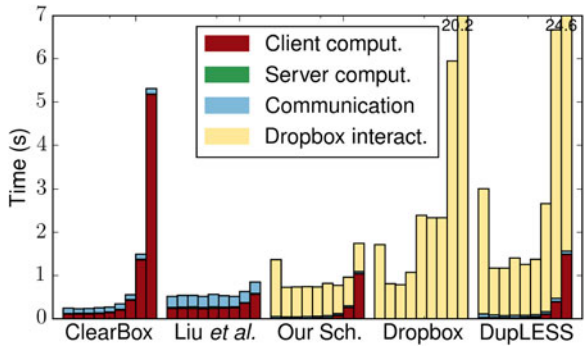

Fig. 17. Processing time of Put requests for deduplicable files of size 2^{2i} KB for $i \in \{0, 1, \ldots, 8\}$ (plotted left to right per prototype).

To provide a more detailed analysis, we have chosen two Put requests—the "most expensive" and the "least expensive" one (across all prototypes) and split the processing times into client computation, server computation, communication between client(s) and server and interaction between client/server and Dropbox, as shown in Fig. 16.

To demonstrate that file size is likely the major influence factor we use a dataset of random content files of size 2^{2i} KB for $i \in \{0, 1, \ldots, 8\}$ (i.e., from 1 KB to 64 MB), pre-upload them enough times to make them popular and then measure Put requests for each, see Fig. 17. The results demonstrate that ClearBox is the best for deduplicable small files, but for larger files the client-side processing is increasing notably (mostly due to the complex FID computation). The prototype by Liu et al. shows very smooth results, only lightly dependent on the file size (initial file hashing) but has the biggest communication overhead. Our scheme and DupLESS are highly influenced by interaction with Dropbox and would benefit from a cloud backend with much faster connection initiation. Interestingly, Dropbox is obviously slower during the initial upload request (first file for our scheme and DupLESS) than during the following ones.

Communication Overhead. We use *tcpdump* to measure communication on the network interface for a deduplicable 1 MB file (overhead is file-size-independent, provided that PoW in ClearBox is capped for 64 MB buffer as suggested). We repeated the Put operation $100\times$ to avoid single-measurement errors. Both in our scheme and in DupLESS the client only sends one request and receives one reply, together these correspond to less than 1 KB (including the DupLESS-based session tracking and rate limiting information). ClearBox uses three requests and responses per Put (key request, FID and PoW challenge). The overhead depends on the PoW parameters, for the tested settings it did not reach 10 KB. The proposal of Liu et al. contains significant overhead due to the PAKE processing (set to default 30 requests) and averaged at 110 KB per Put.

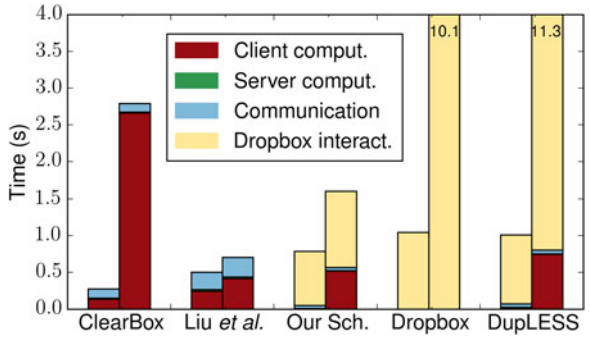

Fig. 16. Processing time for the least expensive (left; 157 KB file) and the most expensive (right, 33 MB file) Put request split into individual costs (same sample as Fig. 15).

Non-Deduplicable Put *Request*. Due to space reasons we leave out the detailed results for non-deduplicable Put requests, as these are less likely to occur for datasets with good deduplication rate. To provide a quick summary—all prototypes inherit the Dropbox interaction cost of DupLESS (see Fig. 17, DupLESS, yellow bar) corresponding to the actual data upload, ClearBox adds intensive PoW computation on the server-side, the proposal of Liu et al. spares some PAKE communication and our scheme adds additional encryption layer (which is quite fast and corresponds only to about $1/3$ more client computation overhead). DupLESS retains the cost as it does not differ between Put for deduplicable and non-deduplicable file. Even though ClearBox has the highest overhead, it happens only once per file (first upload), but potentially more times for Liu et al. and t times for our scheme. Whichever scheme is used, the initial deployment overhead will be considerably higher than the processing overhead once the storage "fills in" reasonably.

Get *Request*. The comparison of Get requests processing reveals that all schemes perform near-equally. ClearBox adds overhead by generating the download URL, the proposal by Liu et al. was originally designed to use storage directly on server so the data flows through it from Dropbox to client. Our scheme and DupLESS allows the clients to connect directly to Dropbox and get their files; our scheme requires an additional decrypt in case of an unpopular file. All the added overheads are marginal compared to the cost of Dropbox interaction and the actual data download.

Deduplication. The actual deduplication process is nearly "free" (comparison of hashes or encrypted data) for all prototypes but that of our scheme. In our scheme, deduplication must be implemented in the cloud backend and consumes approx. 850 ms (considering 1 MBps link between IRS and S) + symmetric decryption of the upper encryption layer for at least one file (more if checks are required). This cost, while higher than for other schemes, is still under 1 s for a 100 MB file. This measurement was done on a separate cloud application as Dropbox does not support it.

Summary. All the proposals have very low server computation cost, allowing the server components to serve multiple clients and scale well when needed. Comparison with a plain Dropbox service shows that deployment of the

prototypes lowers the user-perceived latency for popular file Put requests (with the exception of DupLESS which only increases the latency by less than 1/10 per Put) and adds only a minimal overhead for Get requests. Considering the offered storage space savings, all solutions create a win-win situation for both users and storage provider. Results also demonstrate that there is no "winner", each solution has its pros and cons. Depending on the cloud back-end performance, our scheme is on par with ClearBox for smaller files and outperforms it for bigger files (but does not offer PoW). The proposal of Liu et al. outperforms other schemes for bigger files but is ineffective for small files and requires cooperating online users. Apart from ClearBox, all proposals require modification of the cloud back-end but it enables the possibility to download files even if the server component is down. Since all proposals are different, also from the security perspective, we recommend potential adopters to choose the one that best fits their requirements and environment from both security and performance perspectives.

8 Discussion

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

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 degraded for users who own popular files. One solution for the latter case would be to incorporate anonymous and unlinkable credentials for authentication [33], [34]. 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. To prevent uncontrollable boom, the thresholds should be few and chosen apriori by the provider. Users are then free to choose a threshold they prefer (from the list of offered thresholds) to use for encryption of their data, with the property that a file uploaded with a given threshold t_1 does not count towards popularity of 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 index repository 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 GenSecIdx interface.

Trusted Index Repository Service . Trusted IRS is a known limitation of the scheme. While it is currently not possible to let IRS be corruptible without violating the semantic security of unpopular files, we are considering possible workarounds. It is impossible to get rid of the indexes without sacrificing the deduplication capability. However, the IRS not having access to the actual data content (not even in the encrypted form) opens way to various options. For example, if the indexes could collide, the attacker that compromised IRS could not be sure if the users really uploaded the file he knows the hash/content for or some other file with colliding index. Deduplication at S could still be possible since the encrypted contents may be easily groupable, e.g., via file size (not known to IRS). We are currently evaluating the necessary scheme changes and practicality of this idea.

9 Conclusion

This work deals with the inherent tension between storage optimization methods and end-to-end encryption.

In our previous work [1] we introduced a novel approach that allows to vary the security level of a file based on how popular that file is among the users of the system and presented an encryption scheme that implements this approach. In this work we modify and significantly enhance the originally proposed scheme, focusing on its practicality. Specifically, re-located storage of decryption shares and removal of file popularity information from the storage provider allowed us to strengthen the security of unpopular files and simplify the security proofs while modification of scheme internal processes led to decreased complexity and clear functionality isolation of individual algorithms. The resulting scheme prevents deduplication of unpopular data and allows their automatic transition to a popular state (once they are uploaded by enough users) where deduplication can be applied.

To evaluate space savings efficiency and the computational and communication overhead of the proposal, we present an extensive performance evaluation using real datasets and real-like environment. To provide comparison to other state-of-the-art secure deduplication schemes we include their prototypes in the performance evaluation. The results show that there is no clear "winner"—none of the schemes outperforms others under all conditions, each has advantages and disadvantages with regards to a particular data mix, environment and requirements.

Security-wise, our scheme is resilient to user-collusion attacks (up to a clearly defined point) and to an honest but curious storage provider. By automatically differentiating between popular and unpopular data, our scheme alleviates the user's need to handle low-entropy files differently (if their eventual deduplication is acceptable) and surpasses the other schemes in the fact that it never deduplicates unpopular files nor leaks whether there are any duplicates of unpopular files in the storage (unless the indexing server is compromised). These advantages come at the cost of lower space-saving efficiency compared to the other schemes.

In future work we will focus on further improvements of the scheme based on the community feedback. Security-wise, finding an alternative solution to the trusted index repository service and removing this limiting component from the scheme seems a reasonable next step.

ACKNOWLEDGMENTS

We thank Alessandro Sorniotti and Elli Androulaki, both from IBM Research-Zurich, for invaluable comments and support that helped to improve this publication. We also thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions and authors of related works who provided prototype implementations of their solutions. This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/139/OHK3/2T/13.

APPENDIX
PROOF OF LEMMA 1

SXDH assumes two groups of prime order q, $\mathbb{G}_1$, and $\mathbb{G}_2$, such that there is no 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 Decisional Diffie-Hellman (DDH) holds in both $\mathbb{G}_1$, and $\mathbb{G}_2$, and the bilinear decisional Diffie-Hellman (BDDH) holds given the existence of $\hat{e}$ [26].

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. The algorithm $\mathcal{C}$ simulates an environment in which polynomial-time bounded adversary $\mathcal{A}$ operates, using its advantage in the game IND_μ-BCPA to decide whether $W = \hat{e}(g', \bar{g}')^{abc}$. $\mathcal{C}$ interacts with $\mathcal{A}$ within an IND_μ-BCPA game:

Setup Phase. $\mathcal{C}$ sets $q \leftarrow q'$, $\mathbb{G}_1 \leftarrow \mathbb{G}'_1$, $\mathbb{G}_2 \leftarrow \mathbb{G}'_2$, $\mathbb{G}_T \leftarrow \mathbb{G}'_T$, $\hat{e} = \hat{e}'$, $g \leftarrow g'$, $\bar{g} \leftarrow \bar{g}'$, $\bar{g}_{\text{pub}} = \bar{A}$. Notice that the secret key $\mathsf{sk} = a$ is not known to $\mathcal{C}$. $\mathcal{C}$ also generates the list of user identities $\mathsf{U} = \{\mathsf{U}_i\}_{i=0}^{n-1}$. $\mathcal{C}$ sends $\mathsf{pk} = \{q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T\, \hat{e}, \mathcal{O}_{\mathsf{H}_1}, \mathcal{O}_{\mathsf{H}_2}, \bar{g}, \bar{g}_{\text{pub}}\}$ and U to $\mathcal{A}$. $\mathcal{A}$ declares the list $\mathsf{U}_\mathcal{A}$ of $n_\mathcal{A} < t - p_{\text{lim}} - 1$ user identities that will later on be subject to $\mathcal{O}_{\mathsf{C}corrupt}$ calls. Let $\mathsf{U}_\mathcal{A} = \{\mathsf{U}_i\}_{i=0}^{n_\mathcal{A}-1}$. Next, $\mathcal{C}$ picks $t - p_{\text{lim}} - 1$ random integers $y_i \leftarrow_R \mathbb{Z}_q^*$. Let P be a $t - p_{\text{lim}} - 1$ degree Lagrange polynomial implicitly defined to satisfy $\mathsf{P}(0) = a$ and $\mathsf{P}(i) = y_i$ for $i = 1, \ldots, t - p_{\text{lim}} - 1$. $\mathcal{C}$ then sets the key-shares to $(i, \mathsf{sk}_\mathsf{i}) \leftarrow y_i$, $i \in [1, t - p_{\text{lim}} - 1]$ and assigns $(i, \mathsf{sk}_\mathsf{i})$ for $i \in [1, n_\mathcal{A}]$ to corrupted users.

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

$\mathcal{O}_{\mathsf{H}_1}$: to respond to $\mathcal{O}_{\mathsf{H}_1}$-queries, $\mathcal{C}$ maintains a list of tuples $(v, h_v, \rho_v, \mathrm{c}_v)$ 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 '1' 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 $h_v \in \{0,1\}^\lambda$. To be consistent in her $\mathcal{O}_{\mathsf{H}_2}$ responses, $\mathcal{C}$ maintains the history of her responses in her local memory.

$\mathcal{O}_{\mathsf{C}corrupt}$: $\mathcal{C}$ responds to a $\mathcal{O}_{\mathsf{C}corrupt}$ query involving user $\mathsf{U}_i \in \mathsf{U}_\mathcal{A}$, by returning the coordinate y_i chosen in the Setup Phase.

Challenge Phase. $\mathcal{A}$ submits m_* and ℓ_* to $\mathcal{C}$. Next, $\mathcal{C}$ runs the algorithm for responding to $\mathcal{O}_{\mathsf{H}_1}$-queries for ℓ_* to recover the entry from the $\mathcal{O}_{\mathsf{H}_1}$-list. Let the entry be $(\ell_*, h_{\ell_*}, \rho_{\ell_*}, \mathrm{c}_{\ell_*})$. If $\mathrm{c}_{\ell_*} = 0$, $\mathcal{C}$ aborts. Otherwise, $\mathcal{C}$ computes $E_* \leftarrow W^{\rho_{\ell_*}}$, sets $c_* \leftarrow (m_* \oplus \mathsf{H}_2(E_*), \bar{C})$ and returns c_* to $\mathcal{A}$.

Guessing Phase. $\mathcal{A}$ 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, $W^{\rho_{\ell_*}} = \hat{e}(g, \bar{g})^{abc\rho_{\ell_*}} = \hat{e}((B^{\rho_{\ell_*}})^a, \bar{g}^c) = \hat{e}\Big(H_1(\ell_*)^{\mathsf{sk}}, \bar{C}\Big)$. Clearly, if $\mathrm{c}_{\ell_*} = 1$ and $\mathrm{c}_\ell = 0$ for all other queries to $\mathcal{O}_{\mathsf{H}_1}$ such that $\ell \neq \ell_*$, then the execution environment is indistinguishable from the actual game IND_μ–BCPA. This happens with probability $\Pr[\mathrm{c}_{\ell_*} = 1 \wedge (\forall \ell \neq \ell_* : \mathrm{c}_\ell = 0)] = \delta(1-\delta)^{\mathcal{Q}_{H_1}-1}$, 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 $\mathsf{Adv}^\mathcal{A}_{\mathsf{IND}_\mu\text{-BCPA}}$ is bounded as $\mathsf{Adv}^\mathcal{A}_{\mathsf{IND}_\mu\text{-BCPA}} \leq e \cdot (\mathcal{Q}_{H_1} + 1) \cdot \mathsf{Adv}^\mathcal{C}_{\mathsf{SXDH}}$.

REFERENCES

[1] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A secure data deduplication scheme for cloud storage," in *Proc. 18th Int. Conf. Financial Cryptography Data Secur.*, 2014, pp. 99–118.

[2] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Secur. Privacy*, vol. 8, no. 6, pp. 40–47, Nov./Dec. 2010.

[3] S. Keelveedhi, M. Bellare, and T. Ristenpart, "DupLESS: Server-aided encryption for deduplicated storage," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 179–194.

[4] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in *Proc. SYSTOR: Israeli Exp. Syst. Conf.*, 2009, Art. no. 8.

[5] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Proc. ACM/IFIP/USENIX Middleware Conf. Companion*, 2008, pp. 12–17.

[6] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proc. SYSTOR: Israeli Exp. Syst. Conf.*, 2009, Art. no. 6.

[7] M. Dutch and L. Freeman, "Understanding data de-duplication ratios," [Online]. (2008). Available: http://www.snia.org/sites/default/files/Understanding_Data_Deduplication_Ratios-20080718.pdf, Accessed on: Mar. 7, 2016.

[8] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik, "Estimation of deduplication ratios in large data sets," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.

[9] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 491–500.

[10] R. D. Pietro and A. Sorniotti, "Boosting efficiency and security in proof of ownership for deduplication," in *Proc. 7th ACM Symp. Inf. Comput. Commun. Secur.*, 2012, pp. 81–82.

[11] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proc. 22nd Int. Conf. Distrib. Comput. Syst.*, 2002, pp. 617–624.

[12] M. W. Storer, K. M. Greenan, D. D. E. Long, and E. L. Miller, "Secure data deduplication," in *Proc. ACM Workshop Storage Secur. Survivability*, 2008, pp. 1–10.

[13] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-locked encryption and secure deduplication," in *Proc. 32nd Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2013, pp. 296–312.

[14] M. Bellare and S. Keelveedhi, "Interactive message-locked encryption and secure deduplication," in *Proc. 18th IACR Int. Conf. Practice Theory Public-Key Cryptography*, 2015, pp. 516–538.

[15] J. Xu, E. Chang, and J. Zhou, "Weak leakage-resilient client-side deduplication of encrypted data in cloud storage," in *Proc. 8th ACM Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 195–206.

[16] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 886–900.

[17] J. Li, X. Chen, M. Li, J. Li, P. P. C. Lee, and W. Lou, "Secure deduplication with efficient and reliable convergent key management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1615–1625, Jun. 2014.
[18] P. Meye, P. R. Parvédy, F. Tronel, and E. Anceaume, "A secure two-phase data deduplication scheme," in *Proc. 6th IEEE Int. Symp. Cyberspace Safety Secur.*, 2014, pp. 802–809.
[19] Y. Duan, "Distributed key generation for encrypted deduplication: Achieving the strongest privacy," in *Proc. 6th Edition ACM Workshop Cloud Comput. Secur.*, 2014, pp. 57–68.
[20] P. Puzio, R. Molva, M. Önen, and S. Loureiro, "ClouDedup: Secure deduplication with encrypted data for cloud storage," in *Proc. IEEE 5th Int. Conf. Cloud Comput. Technol. Sci.*, 2013, pp. 363–370.
[21] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 874–885.
[22] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li, "Liquid: A scalable deduplication file system for virtual machine images," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1257–1266, May 2014.
[23] J. R. Douceur, "The Sybil attack," in *Proc. 1st Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 251–260.
[24] S. Goldwasser and S. Micali, "Probabilistic encryption," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984.
[25] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.
[26] G. Ateniese, J. Camenisch, S. Hohenberger, and B. de Medeiros, "Practical group signatures without random oracles," IACR Cryptology ePrint Archive 2005/385.
[27] P. S. L. M. Barreto, B. Lynn, and M. Scott, "Efficient implementation of pairing-based cryptosystems," *J. Cryptology*, vol. 17, no. 4, pp. 321–334, 2004.
[28] B. Lynn, "The pairing-based cryptography library," [Online]. (2007). Available: http://crypto.stanford.edu/pbc/, Accessed on: Mar. 7, 2016.
[29] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proc. 13th ACM Conf. Comput. Commun. Secur.*, 2006, pp. 89–98.
[30] The Pirate Bay 2008–12 Dataset. Department of Informatics, University of Zurich. [Online]. Available: http://www.csg.uzh.ch/publications/data/piratebay.html, Accessed on: Mar. 7, 2016.
[31] Ubuntu Popularity Contest. [Online]. (2016). Available: http://popcon.ubuntu.com/, Accessed on: Mar. 15, 2016.
[32] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López, "Faster explicit formulas for computing pairings over ordinary curves," in *Proc. 30th Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2011, pp. 48–68.
[33] J. Camenisch, S. Hohenberger, and A. Lysyanskaya, "Balancing accountability and privacy using E-cash (extended abstract)," in *Proc. 5th Int. Conf. Secur. Cryptography Netw.*, 2006, pp. 141–155.
[34] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf, "Pseudonym systems," in *Proc. 6th Annu. Int. Workshop Select. Areas Cryptography*, 1999, pp. 184–199.

Jan Stanek received the Bc (BSc equiv.) and Mgr (MSc equiv.) degrees in computer science from the Faculty of Mathematics and Physics, Charles University, Prague. He is working toward the PhD degree in the Department of Telecommunications, Faculty of Electrical Engineering, Czech Technical University in Prague. He is a researcher in the Research and Development Centre for Mobile Applications by the Czech Technical University. Prior to that, he was a researcher with the Institute of Photonics and Electronics, Czech Academy of Sciences. His current research focuses on security and privacy in clouds and networks. The topic of his PhD thesis is data protection in cloud environments. He is a member of the IEEE.

Lukas Kencl received the MSc degree in computer science from Charles University, Prague, and the PhD degree in communication networks from EPFL, Switzerland, in 1995 and 2003, respectively. He has, since 2007, been the director of the R&D Centre for Mobile Applications (RDC), Department of Telecommunications Engineering, FEE, Czech Technical University in Prague (CTU). Prior to CTU, he was with Intel Research, Cambridge, United Kingdom, (2003-2006) and with IBM Research-Zurich (1999-2003). His research focuses on cloud and network services, network architecture and system optimization, and on network privacy and security. He is co-inventor of five patents and holder of many industrial awards (Electrolux, IBM Research, Cisco Research, Microsoft Research). He was general chair of the IFIP Networking 2012 Conference and regularly serves on major networking conference TPCs. He is a member of the IEEE.