Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Telecommunication Engineering

Secure Data Deduplication in Cloud Storage Services

Doctoral Thesis

Mgr. Jan Staněk

Ph.D. programme: P2612 Electrical Engineering and Information Technology
Branch of study: 2601V013 Telecommunication Engineering
Supervisor: Dr. Lukáš Kencl

Prague, March 2018

Thesis Supervisor:
Dr. Lukáš Kencl
Department of Telecommunication Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czech Republic

Declaration

I hereby declare I have written this doctoral thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, March 2018

.....
Mgr. Jan Staněk

Abstract

Cloud storage services became a viable alternative to other data storage options, however, their operators are facing an issue of optimization versus privacy. To ensure data confidentiality, clients would like to use end-to-end data encryption, however, proper encryption renders all of the classic storage optimization techniques, such as data deduplication, useless. On the other hand, these optimization techniques typically save space by exploitation of similarities and/or equalities in the data, which, by definition, breaks data confidentiality.

Data deduplication is an optimization technique that offers very good space-saving possibility for multi-user cloud storage services, though its deployment raises multiple security and privacy concerns. To address these concerns, we analyze the (in)compatibility of deduplication and encryption and propose a novel *secure deduplication solution* based on the concept of "data popularity". Our proposal rests on the real-life assumption that "a secret shared too many times ceases to be a secret" *i.e.* that identical data outsourced to the cloud by many users do not require as strong protection as unpopular data that are outsourced by few users only. By introducing this distinction, our solution allows to protect unpopular data in a stronger, deduplication preventing, way, while providing weaker protection, allowing deduplication, to popular data. Moreover, the proposed mechanism supports transition between the popular and unpopular states to happen automatically, requiring only low computational overhead and no direct user interaction.

While our solution is reasonably cost effective for some datasets, it is not very efficient for others. The same holds true for other secure deduplication proposals – there is often an advantage accompanied by a disadvantage. To demonstrate efficiency of our solution we include an extensive performance evaluation as well as comparison of our scheme to other state-of-the-art secure-deduplication solutions. We analyze the different approaches and their features and comment on their applicability for deployments with varying requirements.

Keywords: security, data protection, deduplication, convergent encryption, cloud storage, popularity, threshold cryptosystem, multiple encryption

Abstrakt

Cloudová úložiště se poslední dobou stala výhodnou alternativou k jiným typům úložišť, jejich provozovatelé se však potýkají s problémem jak skloubit optimalizaci a ochranu uložených dat. Pro zajištění utajenosti svých dat by klienti úložiště preferovali šifrovat data ještě na svém zařízení, nicméně takové šifrování efektivně znemožňuje provozovateli úložiště využít klasické techniky pro optimalizaci uložení dat, jako je například deduplikace. Na druhou stranu, optimalizační techniky určené k úspornějšímu uložení dat často využívají právě shodnost a podobnost dat, čímž, již z definice, porušují jejich utajení.

Deduplikace dat je optimalizační technika, která nabízí velmi výhodný způsob jak šetřit úložnou kapacitu v cloudových úložištích využívaných mnoha klienty, nicméně její nasazení vyvolává mnoho otázek z hlediska bezpečnosti a ochrany dat. Soustředili jsme se na tyto otázky a analyzovali jsme možnosti jak skloubit šifrování a deduplikaci, čehož výsledkem je návrh nového řešení umožňujícího bezpečnou deduplikaci dat založenou na principu popularity dat. Náš návrh se opírá o předpoklad založený na reálném pozorování, že "tajemství sdílené příliš mnohokrát přestává býti tajemstvím", tedy, že pokud mnoho uživatelů ukládá shodná data, tato data pravděpodobně nevyžadují takovou ochranu, jako data uložena pouze několika málo uživateli. S využitím tohoto dělení dat, navržené řešení umožňuje chránit "nepopulární data" bezpečnějším způsobem, který znemožňuje deduplikaci a chránit "populární data" o něco méně bezpečným způsobem, který však deduplikaci umožňuje. Navržený mechanismus navíc podporuje automatický přechod mezi nepopulárními a populárními daty bez potřeby uživatelova zapojení do tohoto procesu a je úsporný z hlediska požadovaného výpočetního výkonu.

Přestože naše řešení poskytuje rozumnou volbu pro některé skupiny dat, existují i skupiny dat, pro které se nehodí. To samé lze obecně říci o jakémkoliv v současnosti známém návrhu bezpečné deduplikace - typicky pro každou výhodu existuje i nějaká, k ní vztažená, nevýhoda. Abychom demonstrovali efektivitu našeho řešení, udělali jsme poměrně rozsáhlé porovnání jeho výkonnosti s výkonností dalších state-of-the-art řešení pro bezpečnou deduplikaci. Analyzovali jsme rozdílné přístupy k implementaci bezpečné deduplikace a jejich vlastnosti a popsali jsme jejich využitelnost v prostředích s různorodými požadavky.

Klíčová slova: bezpečnost, ochrana dat, deduplikace, konvergentní šifrování, cloudové úložiště, popularita, kooperativní kryptosystémy, vícenásobné šifrování

Acknowledgements

I would like to thank everyone who helped me to finish this thesis, one way or another. Specifically: my thesis supervisor Lukas Kencl who opposed my ingenious ideas that were often leading to blind alleys or were simply wrong, my girlfriend and family, who were patient with me when I was grumpy and unmotivated, and all my friends, colleagues and fellow students for their understanding. Special thanks goes to IBM Research for the generous scholarship and the possibility to spend some time as an intern at IBM Research Zurich and meet many wonderful people that all helped me in my research career: Alessandro Sorniotti and Elli Androulaki who helped me a lot when the first vague secure deduplication scheme idea was forming and later supported me both technically and morally to shape it into its original form; Jens Jelitto and Evangelos Eleftheriou for support and very efficient management; and all the people that I met during my internship in Zurich. Big thanks goes also to professor Simak, my colleagues and staff at the Department of Telecommunications at Czech Technical University in Prague for succeeding in creation of a friendly and stimulating work environment. And of course I must not forget Jiri Kuthan and guys from iptel.org that helped me a lot in my early PhD studies era and provided me with ideas and data that I couldn't obtain from anywhere else. I owe thanks also to authors of other secure deduplication schemes that provided interesting different views on the topic and their works gave me a lot of ideas and material to compare to. Last but not least, my thanks goes to all the reviewers of the papers we published (and tried to publish) on the secure deduplication topic – their insightful comments, even the negative ones, led to notable improvement of the final scheme.

List of Tables

List of Figures

Contents

Chapter 1

Introduction

Cloud solutions introduce a viable cost-effective alternative to standard in-house IT solutions for various clients, be it private users or big companies. However, migration to the cloud requires clients to outsource their data – instead of residing in a specific place on a specific disk, the data may end up virtually anywhere where the cloud service provider has his storage warehouses. This poses a big challenge for both clients and cloud providers – clients typically do need to process their data and cloud providers need to deploy storage optimization techniques to save space in their storage, thus it is not possible for the client to encrypt the data before outsourcing them to cloud and retain the key locally. Without the key, the cloud provider cannot decrypt the data, which effectively prevents both data processing and storage optimization. Straightforward solutions such as sharing the key with the cloud provider or renting security as a service (letting someone else do the "security stuff") means the user is giving up control over his data, which might not be acceptable for various reasons (e.g. law regulations). Solving the cloud-data security issue in a general manner that would suit everyone is a nearly-unsolvable problem, as the different approaches typically introduce restrictions that are not acceptable for some of the use-cases. Therefore, research focusing on cloud security typically does not try to encompass the whole issue of outsourced data and, instead, tries to ensure improved security only for a subset of use cases.

There are two major research areas focusing on cloud data security – the first focuses on processing of encrypted data and its ultimate goal is to allow arbitrary processing of encrypted data without their actual decryption; the second approach focuses on storage optimization techniques for encrypted data. While both areas seem similar (both attempt to enable some mechanism for encrypted data), they differ notably and the requirements are contradictory. The first typically requires more storage space to offer more generic processing whereas the second focuses primarily on saving storage space. In our research, we initially tried to tackle both areas at once, but since a general approach proved overly

conflicting, we chose to focus more on the latter approach – *enabling storage space saving and optimization techniques while retaining as high level of security as possible.*

Encrypted data processing therefore falls out of scope of this thesis, though for people interested in that area we recommend to start with the dissertation thesis of Craig Gentry [1] and follow with works that cite it and/or use the "homomorphic encryption" key words.

A cloud provider can deploy various storage optimization techniques to save some storage space. All share a common denominator – to save space, they exploit some similarity in the data being stored. This is in direct contradiction with the user-required data confidentiality, which requires that no information about the protected data can be extracted. Additionally, considering encryption as the protection mechanism of choice for data confidentiality, proper encryption generates ciphertexts with very high entropy, reducing efficiency of any similarity- or equality-based storage optimization technique to near-zero. To provide an example, we took a file that is likely to be heavily copied and shared, encrypted it and then compared the files for similarities (for simplicity we used the default Microsoft Word 2013 encryption and an on-line tool for binary comparison), see Figure 1.1. As the results demonstrate, encryption using different keys produces very different ciphertexts and thus plaintexts that are very similar (even equal) produce ciphertexts with little-to-none similarity, effectively preventing deduplication.

Researching various storage-optimization techniques, we identified data deduplication

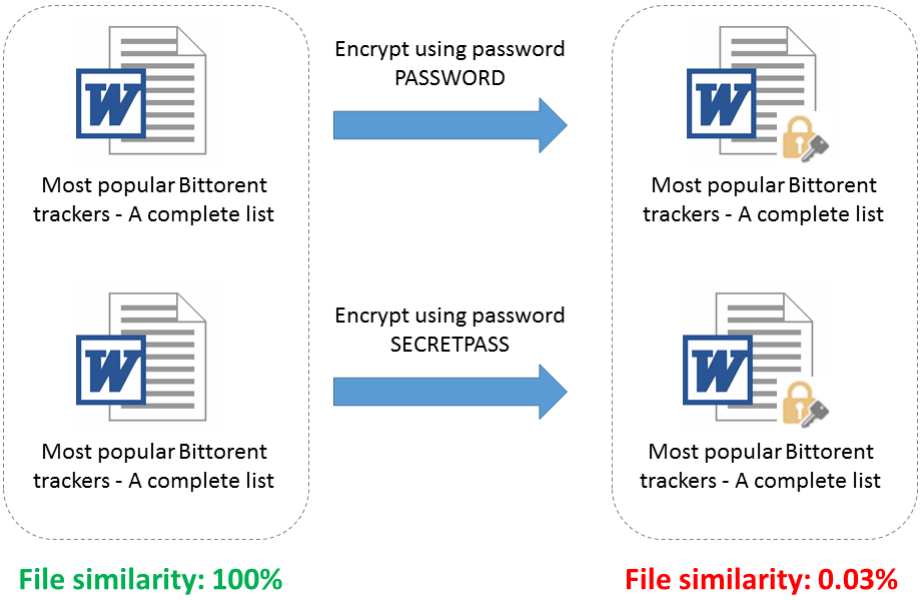

Figure 1.1: Demonstration of how encryption breaks similarities. Note that some minor similarity remains even after encryption – this is caused by the Microsoft Word file format that encrypts only text contents and properties, but leaves default templates and file structure unchanged.

as potential candidate for a *secure* storage-optimization scheme. *Data deduplication* is an optimization technique that achieves enormous data reduction rate for some datasets (*e.g.* over 99% reduction for typical backup scenarios [2]). Since multi-user cloud storage services storing, among other data, popular songs, photos and movies seem to be a good fit for highly-efficient deduplication, we analyze how encryption and data deduplication can be combined to achieve both a *storage-efficient* and *secure* storage service.

Data deduplication is a process where the storage provider stores only a single copy of a file (or its part) stored by several users. This way, if there are four owners of an identical file, who all store the file using the same cloud-storage service, the cloud provider can only store the file once, saving space equal to three-times the file size. To avoid full-size file comparisons, deduplication uses indexes (sometimes also called tags or locators). To obtain an index the file is first processed using an indexing function (typically a collision-resistant hash function) and then the comparison is done using the indexes instead of comparing entire files. Based on the location where the index computation actually occurs, we differentiate between *client-side* and *server-side deduplication*. Client side deduplication, as the name suggests, computes the index in the client application and then sends only the index to the storage service. The storage service checks whether it already has a file with this index stored, and if so, file upload is not necessary and the client is just added as another owner of the already-stored file. This way, client-side deduplication also benefits from network bandwidth savings – instead of transferring the whole file, only the index is being transferred. Server-side deduplication requires the client to upload the whole file, which is then processed on the provider-side.

While security was not an integral part of early deduplication designs, its need soon became imminent with users requiring protection of their data. The first attempt to combine encryption and deduplication to a "secure deduplication" solution came in 2002 under the term of convergent encryption [3]. *Convergent encryption* is a special form of deterministic encryption, in which the encryption key is derived from the plaintext and thus any owner of the same plaintext generates the same ciphertext. This way, if two users encrypt the same file, they obtain the same encrypted file, which, in turn, can be deduplicated when stored using the same storage service. Convergent encryption thus seems like a perfect secure-deduplication solution – data is encrypted and, at the same time, deduplication is possible. Unfortunately, convergent encryption was proven insecure [4]. Moreover, a follow-up work of Bellare et al. proved that a general impossibility result holds stating that classic semantic security is not achievable for schemes implementing plain convergent encryption [5]. Despite the general impossibility result, multiple research teams refused to drop the secure deduplication idea, ours included.

Our core idea formed around a real-life assumption that "a secret shared many times

ceases to be a secret". Based on this assumption, we created the notion of "data popularity" – if data (*i.e.* a file) is stored by a few users only, it is unpopular and should be well protected; once data is stored by a certain threshold amount of users, it becomes popular and the level of protection can be decreased. Arguably, if the user stores a copy of his personal documents, his payslip or a draft of an unsubmitted scientific paper, there will likely be not many more users (if any) storing the same file and the file remains unpopular and thus strongly protected. On the other hand, if the user stores a popular song or video, there will likely be many more users storing it too – being already widely-shared, such a file does not require such a strong protection and the level of protection can be decreased. Note that this popularity-based differentiation also automatically solves the issue how to automatically choose whether a file should or should not be deduplicated – it is not necessary to split the files between those potentially sensitive that should not be deduplicated and those that could, since all data are first considered unpopular (and thus non-deduplicable) by design and change the state to popular automatically, once a popularity condition is met.

The technical challenge is to design a solution that implements the data popularity idea – the stronger protection required for unpopular data has to be some sort of semantically secure encryption whereas the weaker protection must allow deduplication. We decided to solve this challenge using *two-layered encryption* – the outer layer is semantically-secure encryption and the inner layer convergent encryption. Once enough users share the same file, the outer layer can be automatically peeled off, revealing the inner, convergent, layer. Convergently encrypted file can then be immediately deduplicated. The trickiest part of the process is the automatic transition of file from unpopular to popular – this requires to be triggered by some event initiated either by the users themselves or by some trusted third party. For simplicity, we decided to introduce a trusted third party in the initial design, but later in the work we discuss ways how to reduce the data-related knowledge and trust required by this third party.

Since our solution is quite complex by design, introducing a trusted third party and two layers of encryption, in this work we include an overhead evaluation, demonstrating its cost effectiveness (or ineffectiveness, for some cases). Also, as mentioned, there are multiple other teams working on different secure deduplication solutions. After finalizing and testing our solution, we have analyzed also the results of others and compared all the different solutions using multiple factors. Interestingly, the comparison proved that there is no clear winner – each solution has its own specific requirements, offers different security notions and different efficiency for various datasets. We focused on identification of the differences and offer discussion regarding suitability of the solutions for concrete use cases based on various processing, security and dataset requirements and properties.

The core contributions of this work can be summarized as follows:

- 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 secure deduplication scheme
- we analyze security of the proposed deduplication scheme and comment on its possible strengthening and its cost
- we provide analysis of scheme deduplication efficiency based on popularity properties of real datasets
- we compare our secure deduplication scheme with other state-of-the-art schemes in terms of security and efficiency and comment on their specific differences

The rest of this work is organized as follows: *Chapter 2* describes the preliminary knowledge, concepts and notation necessary for understanding the work; *Chapter 3* summarizes the history and evolution of secure date deduplication and lists related state-of-the-art works; *Chapter 4* provides an introductory overview of our scheme, including the system and security models; *Chapter 5* describes our secure deduplication scheme in detail, listing scheme participants, properties and algorithms; *Chapter 6* presents security analysis of our scheme, including security proofs and alternative settings; *Chapter 7* consists of extensive performance evaluation of scheme properties and efficiency; and *Chapter 8* concludes the work

Chapter 2

Preliminaries

In this chapter we describe the knowledge base that our secure deduplication solution builds upon. Since the terms and notation often vary in the deduplication-oriented and cryptography-oriented literature, we also define uniform terms and notation that will be used throughout our work.

2.1 Data Deduplication

Data deduplication is a storage optimization technique aiming at, as the name suggests, eliminating duplicities. The core idea of data deduplication is simple – if a dataset contains some (part of) data in multiple copies then one can easily store only one copy of such duplicate data and thus save storage space. Efficiency of deduplication is typically expressed in the form of deduplication ratio (also called duplicity ratio) DR that is a ratio of the original dataset size compared to the size after deduplication. The deduplication process is depicted in Figure 2.1.

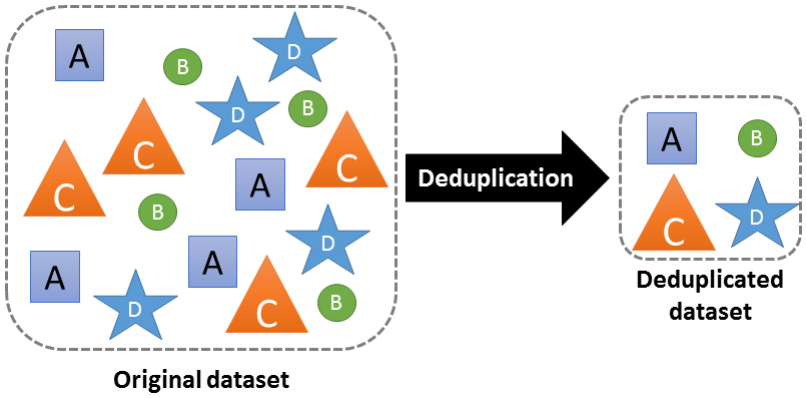

Figure 2.1: Illustration of the data deduplication process. Objects represent different files. Deduplication ratio (DR) in the shown example is $DR = 4 : 1$.

In practice, the trivial core deduplication idea gets a bit more complicated since the implementation has to solve the issues of

1. identifying the duplicate data
2. removing the duplicate data
3. storing the information about the removed duplicate(s)

in some uniform and easy to manage way.

Identifying the duplicate data requires splitting the dataset into pieces that will be compared for equality. Two typical approaches are file-based and chunk-based. File-based approach simply compares the data on the file level – if some file in the dataset has equal contents as some other file, contents are stored only once. Chunk-based approach is more coarse-grained – each file is processed using some splitting function (typically a fixed-offset partitioning or rolling hash) and the newly obtained chunks are then compared. The advantage of file-based approach is simplicity in implementation and lesser computation resource consumption, the advantage of chunk-based approach is better deduplication ratio (since it is able to eliminate duplicates caused by minor in-file changes and common in-file parts). Technically, each deduplication solution can work as file-based or chunk-based, though chunk-based typically requires notably more effort, as the file-level dataset-splitting offered by common filesystems needs to be refined by additional processing.

Another important aspect of identifying duplicate data is when and/or how often the process is done. If we want to deduplicate an already existing dataset, we need to process it as a whole at least once. However, if the dataset is only being formed and/or is dynamically changing (as is typical for cloud storage services), it is more efficient to keep the dataset always in a deduplicated state then to re-do the deduplication process of the whole dataset once in a while. This approach is called on-line or real-time data deduplication – each new piece of data that should be added to the dataset is first processed to obtain it's deduplication index (or indexes, if split to chunks), the index is then compared to a list of already stored contents and if a hit is found, the data is deduplicated on-the-fly (note that the index is sometimes also called tag or locator in deduplication-oriented literature). There exists also a special version of on-line data deduplication that is called lazy or post-poned where the duplicate existence is found in real-time but the actual deduplication process is postponed until some special condition occurs. This special version is often used in secure deduplication solutions.

Focusing on our cloud-based storage service scenario, there are two "deduplication participants" – *data producer* (also called owner, source, client) and *data consumer* (cloud storage service, often called just server). Based on where the processing required to find

the duplicate data (*i.e.* splitting and index computation) is happening we speak about *client-side* (source-based) or *server-side* (target-based) *data deduplication*. *Client-side* requires more computing resources at the client but has the advantage that it is not necessary to transfer the actual deduplicated data – only indexes are being sent to the server if a hit is found, the file/chunk in question does not need to be transferred. Server-side requires no computing resources at the client, as all processing is happening at the server-side, though this requires that all data (even data that would be deduplicated) needs to always be transferred from the client to the server.

Once the duplicate data has been identified the removal process is straightforward – the duplicate data are either deleted from storage (if already stored) or simply not stored (in the on-line deduplication case where identification precedes storing). However, this causes the challenge of how to store information about the deleted/not stored duplicates. If client A stored file F and client B wants to store the same file F and thus deduplication occurs, the cloud storage service still has to record the information that client B is the owner of file F as well. In practice this is often solved by decoupling the data and the metadata information – data files (or chunks) are stored indexed by their deduplication indexes in one copy only and metadata are stored and managed separately either in some unique per-file per-client form or in some aggregate form, depending on the actual storage capabilities and properties. Each data file must have a counter specifying how many clients own it at a time. If the counter reaches zero, the file is deleted.

To sum-up, data deduplication can be either file-based or chunk-based (granularity level) and either client-side or server-side (processing location). For secure deduplication solutions, the granularity level is often less important and most of the solutions can work in both versions, however, the processing location is of utmost importance as it typically has direct influence on data security. Since it is necessary to formalize the descriptions, we cover how to model deduplication in the next section.

2.1.1 Modeling Data Deduplication

To model deduplication in our cloud-based storage service scenario, we define the two deduplication participants – a storage provider (S) that offers storage space with enabled cross-user deduplication and a set of users (U) who store data content on the server. For simplicity of notation, we assume that deduplication happens at the file level. Note that to model chunk-level deduplication one can simply substitute chunk C instead of file F. To identify files and detect duplicates, the scheme uses an indexing function $\mathcal{I}$: $\{0,1\}^* \rightarrow \{0,1\}^*$; we will refer to $\mathcal{I}_F$ as the index for a given file F. As the metadata is typically unimportant from the deduplication perspective (size should be marginal compared to data), we simplify the storage provider's back-end model as an associative

array $\mathsf{DB_S}$ mapping indexes produced by $\mathcal{I}$ to records of arbitrary length e.g. $\mathsf{DB_S}[\mathcal{I}_F]$ is the record mapped to the index of file F. Each record contains two fields, $\mathsf{DB_S}[\mathcal{I}_F]$ .data and $\mathsf{DB_S}[\mathcal{I}_F]$ .users. The first contains the content of file F, the second is a list of users that have so far uploaded F. In practice, the $\mathsf{DB_S}[\mathcal{I}_F]$ .users list would be replaced by a simple counter and the actual users would have their own metadata-based file abstractions with file properties (such as date of creation/storage, size, date of modification, type etc.) stored in a separate storage structure. Note that we only chose our simple model to easily describe which users and how many do own the actual data contents of a file. Evolving the simple model into a real cloud-based storage service structure scenario should nevertheless be quite straightforward.

The storage provider and users interact using three user-invoked algorithms: Put, Get and Delete. Depending on whether we are modeling client-side or server-side deduplication, the indexing function $\mathcal{I}$ is computed either by the client or by the server. In the following algorithms we model the client-side version (server-side version can be directly derived by substituting $\mathcal{I}_F$ with F and adding $\mathcal{I}$ computation and $\mathcal{I}_F$ propagation back to the user in the Put algorithm):

$\mathsf{Put}(\mathcal{I}_F, \mathsf{U}_i, F)$: user U_i sends $\mathcal{I}_F$ to S. S checks whether $\mathsf{DB_S}[\mathcal{I}_F]$ exists and if so, appends U_i to $\mathsf{DB_S}[\mathcal{I}_F]$ .users. Otherwise, it requests U_i to upload the content of F, which will be assigned to $\mathsf{DB_S}[\mathcal{I}_F]$ .data. $\mathsf{DB_S}[\mathcal{I}_F]$ .users is then initialized with U_i.

$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathcal{I}_F$, 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: $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{data} \leftarrow F$; $\mathsf{DB_S}[\mathcal{I}_F].\mathsf{users} \leftarrow \{\mathsf{U}_i\}$

Figure 2.2: The $\mathsf{Put}(\mathcal{I}_F, \mathsf{U}_i, F)$ algorithm.

$\mathsf{Get}(\mathcal{I}_F, \mathsf{U}_i)$: user U_i sends $\mathcal{I}_F$ to S. The latter checks whether $\mathsf{DB_S}[\mathcal{I}_F]$ exists and whether $\mathsf{DB_S}[\mathcal{I}_F]$ .users contains U_i. If it does, S responds with $\mathsf{DB_S}[\mathcal{I}_F]$ .data. Otherwise, it answers with an error message.

$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathcal{I}_F$, 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

Figure 2.3: The $\mathsf{Get}(\mathcal{I}_F, \mathsf{U}_i)$ algorithm.

$\mathsf{Delete}(\mathcal{I}_F, \mathsf{U}_i)$: user U_i sends $\mathcal{I}_F$ to S. The latter checks whether $\mathsf{DB_S}[\mathcal{I}_F]$ exists and whether $\mathsf{DB_S}[\mathcal{I}_F]$.users contains U_i. If it does, S removes U_i from $\mathsf{DB_S}[\mathcal{I}_F]$.users and if the list is empty after the removal, whole record $\mathsf{DB_S}[\mathcal{I}_F]$ is deleted. Otherwise, it answers with an error message.

$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathcal{I}_F$, 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

Figure 2.4: The $\mathsf{Delete}(\mathcal{I}_F, \mathsf{U}_i)$ algorithm.

2.2 Cryptography

To build our secure deduplication scheme we use many cryptographic schemes and principles with multiple major and minor modifications. The core of our scheme is based on symmetric and convergent encryption schemes that are combined in a thresholded construct exploiting a secret sharing scheme. The secret sharing scheme principle is well described in the literature [6], the rest of the building blocks are described in this section (especially to unify notation and terminology).

2.2.1 Symmetric Cryptosystem

A *symmetric cryptosystem* $\mathcal{E}$ is defined as a tuple (K, E, D) of probabilistic polynomial-time algorithms (assuming a security parameter 1^λ). K stands for key derivation function which takes security parameter 1^λ as input and is used to generate a random secret key k. E stands for encryption function which is used to encrypt a message m with key k and generate a ciphertext c. D stands for decryption function that is used to decrypt c using k to produce m. Note that the same key is used for message encryption and decryption and thus both the user that encrypts and user that decrypts a message have to know the same key k. Keeping k secret is a strong requirement for a symmetric encryption scheme to remain secure.

The algorithms are defined as follows:

$\mathcal{E}.\mathsf{K}(1^\lambda) \rightarrow (k)$: generates random key k of bit-length 1^λ

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

$\mathcal{E}.\mathsf{D}(k, c) \rightarrow (m)$: takes as input a ciphertext c and key k and outputs the cleartext message m.

2.2.2 Convergent Encryption Scheme

A *convergent encryption scheme* $\mathcal{E}_c$, also known as message-locked encryption scheme [7], is similar to a symmetric cryptosystem – it is also defined as a tuple of three polynomial-time algorithms (assuming a security parameter 1^λ) ($\mathcal{E}_c.\mathsf{K}$, $\mathcal{E}_c.\mathsf{E}$, $\mathcal{E}_c.\mathsf{D}$). Differently to a symmetric cryptosystem $\mathcal{E}$ and its key derivation function $\mathcal{E}.\mathsf{K}$ generating random, message-independent, keys, keys generated by $\mathcal{E}_c.\mathsf{K}$ are a deterministic function of the cleartext message m. As a direct consequence, multiple invocations of $\mathcal{E}_c.\mathsf{K}$ and $\mathcal{E}_c.\mathsf{E}$ (on input of a given message m) produce identical keys and ciphertexts, respectively, as output. Also note that since k is derived from m, a convergent encryption scheme cannot pre-generate the encryption key without knowing the message m to be encrypted apriori.

We define the algorithms as follows (note that k_m specifies k corresponding to message m):

$\mathcal{E}_c.\mathsf{K}(1^\lambda, m) \rightarrow (k_m)$: generates message-dependent key k_m of bit-length 1^λ

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

$\mathcal{E}_c.\mathsf{D}(k_m, c) \rightarrow (m)$: takes as input a ciphertext c and key k_m and outputs the cleartext message m.

2.2.3 Public-key Cryptosystem

A *public-key cryptosystem* $\mathcal{E}_{pk}$ (often also called asymmetric cryptosystem) is very similar to a symmetric cryptosystem, only instead of using one key k for both encryption and decryption it uses a key pair consisting of two keys – a public key pk and a secret key sk. Public-key cryptosystems do not require the public key to be kept secret (hence the name public) and are often used as a core building block of a public key infrastructure (PKI) [8]. Since we do not need nor want to get lost in many different ways to exploit asymmetric cryptosystems, we describe them in their simplest form and usage only defined by the algorithms as follows:

$\mathcal{E}_{pk}.\mathsf{K}(1^\lambda) \rightarrow (pk, sk)$: generates random keypair (pk, sk) of bit-length 1^λ

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

$\mathcal{E}_{pk}.\mathsf{D}(sk, c) \rightarrow (m)$: takes as input a ciphertext c and secret key sk and outputs the cleartext message m.

2.2.4 Threshold Cryptosystem

Threshold cryptosystem is an evolution of a basic cryptosystem (symmetric or asymmetric) that offers the ability to share the power of performing certain cryptographic operations (e.g. decrypting a message) among n authorized users, such that any t of them can do it efficiently. Moreover, according to the security properties of threshold cryptosystems it is computationally infeasible to perform these operations with fewer than t (authorized) users.

Threshold ElGamal Cryptosystem A typical example of a threshold public-key cryptosystem is a threshold variant of the popular ElGamal cryptosystem [9] where a threshold number of users have to cooperate to be able to successfully decrypt a message. Same as in the original non-threshold version, the threshold version is defined over a multiplicative cyclic group G and its security is reduced on the discrete logarithm computation problem. There is only one master secret of the system and each participant obtains a share of this secret upon joining the system. Public key of the system is published openly. Every participant can encrypt a message using the public key of the system. As no participant knows the whole master secret key, decryption is only possible if $t + 1$ participants cooperate. Labeling $\mathcal{E}_{tEG}$ the threshold ElGamal cryptosystem, the following algorithms are defined:

$\mathcal{E}_{tEG}.\mathsf{Setup}(1^\lambda, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk}, \mathsf{S})$: generates an efficient description of a multiplicative cyclic group G of order 1^λ with generator g. Then it chooses an integer sk from interval $[1\,;\, 1^\lambda - 1]$ and computes $h = g^{\mathsf{sk}}$. Public key $\mathsf{pk} = (G, q, g, h)$ is published, secret key sk is distributed among participants in a (t+1)-out-of-n secret sharing scheme (for example Shamirs' scheme described in [6]). i-th participant gets share sk_i of the master secret key.

$\mathcal{E}_{tEG}.\mathsf{Encrypt}(\mathsf{pk}, m) \rightarrow (c)$: takes as input message m and produces its encrypted version as follows: integer r is randomly chosen from interval $[1\,;\, 1^\lambda - 1]$ and $c_1 = g^r$ is computed. Next, h^r is computed and used to form $c_2 = m \cdot h^r$. Ciphertext c is then a tuple $c = (c_1\,,\, c_2)$.

$\mathcal{E}_{tEG}.\mathsf{Decrypt}(c) \rightarrow (m)$: t+1 decryptors have to actively participate: i-th decryptor computes a decryption share $d_i = c_1^{\mathsf{sk}_i}$ and provides a tuple $(i,\, d_i)$. Once t+1 tuples from different decryptors are collected, c_1^{sk} can be computed using the Lagrange

formula $\prod_{i \in S} d_i^{\lambda_{0,i}^S} = \prod_{i \in S} c_1^{\mathsf{sk}_i \lambda_{0,i}^S} = c_1^{\sum_{i \in S} \mathsf{sk}_i \lambda_{0,i}^S} = c_1^{\mathsf{sk}}$. Finally, m can be decrypted since $c_2 = m \cdot h^r$ thus $m = c_2 \cdot h^r$ and $c_1^{\mathsf{sk}} = g^{r^{\mathsf{sk}}} = h^r$ which gives $m = c_2 \cdot c_1^{\mathsf{sk}}$.

Non-interactive Threshold Cryptosystem Threshold ElGamal cryptosystem is not usable in situations where users cannot actively participate "on-line" when decryption is required. For these situations a modification is required that allows the users to create so called "decryption shares" that can be used during decryption without active participation of the user. Here we provide a generalized description of such a modified threshold public-key cryptosystem.

A non-interactive 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 1^λ) with the following properties:

$\mathcal{E}_t.\mathsf{Setup}(1^\lambda, n, t) \rightarrow (\mathsf{pk}, \mathsf{sk}, \mathsf{S})$: 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.

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

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

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

Chapter 3

Secure Data Deduplication – Evolution, Properties and State of the Art

In this chapter we first describe the issues related to secure data deduplication definition, then specify the security goals of our secure deduplication solution and describe other state of the art secure deduplication solutions with focus on how their approach differs from our proposed solution.

3.1 Secure Data Deduplication

Whereas data deduplication is a well-known mechanism that is relatively simple to describe (see Section 2.1), secure data deduplication is an actively researched topic and there is not even mutual agreement on what requirements a secure data deduplication scheme should satisfy. This issue is caused mainly by the fact that there are multiple views of "security" that notably differs in various scenarios. In this section we first demonstrate this "inconsistency" issue via a short history overview, follow with the description of the actual weakness of deduplication "causing all the problems" and finish with our view or secure data deduplication requirements and targets, explaining why we chose them.

3.1.1 History of Convergent Encryption and Secure Data Deduplication

Even though the idea to use deterministic keys computed from file contents to encrypt the corresponding files is quite old (the first note that we were able to confirm is from 1996 mentioned by John Pettitt in the cypherpunks mailing list [10]), the first secure

deduplication solution proposal based on convergent encryption including proper technical description and analysis was published by Douceur *et al.* in 2002 [3]. The aim of convergent encryption was to combine data confidentiality with deduplication and the implementation was straightforward – first convergently encrypt the file to be stored and then compute the deduplication index of this file which is then used for equality matching during deduplication. Since the index is computed over encrypted data, the knowledge of the index should not be a problem.

For a few years, the convergent encryption-based approach seemed viable and was adopted in actual production solutions such as Tahoe Least Authority File Storage [11] or Dropbox [12]. However, in 2008 Drew Perttula was awarded the Hack-Tahoe-LAFS award for finding a security issue caused by secure deduplication implemented in Tahoe-LAFS. The issue allowed for the "Learn the Remaining Information Attack" – having a partial file (*e.g.* well known document template) that is expected to be already stored in the storage, the attacker can try to guess the unknown parts of the file and compute the corresponding deduplication indexes of the "guessed files"; if a computed index results in deduplication (corresponding file already stored in the storage) the guess is confirmed and the attacker thus "learns contents of encrypted data" which should not be possible. See Figure 3.1 for a spot-on weakness-documenting code snippet that Pertulla got printed on a t-shirt as a reward.

In 2010, Harnik *et al.* published a paper [4] with a bit more thorough security analysis of convergent encryption, pointing out several severe exploitations resulting in the possibility to guess file contents (same as demonstrated by Perttula), use storage service as a covert communication channel or exploit storage service as a content distribution network (without consent or cooperation of the storage provider). Publication of these findings led to forming of a new research area – secure data deduplication.

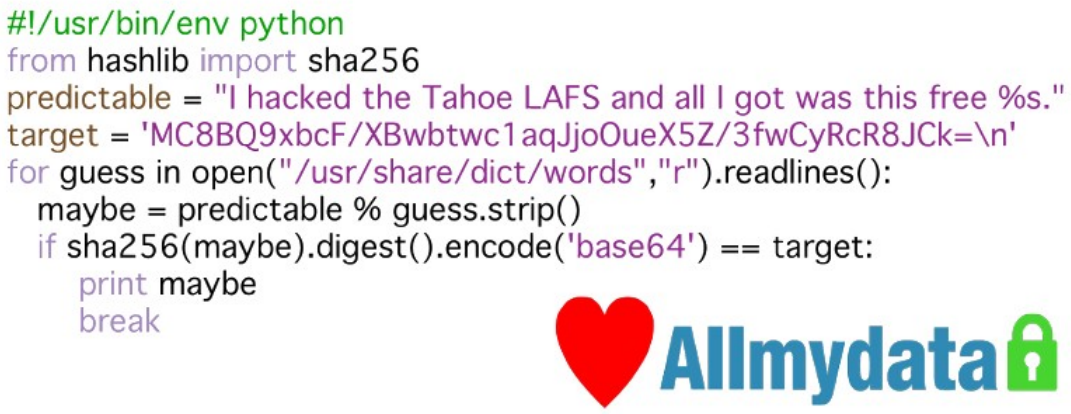

Figure 3.1: Code implementing a simple form of "Learn the Remaining Information Attack". © Allmydata & Drew Perttula.

Different teams chose different strategies – some were trying to find "simple modifications" of deduplication solutions using convergent encryption to thwart (some of) the attacks described by Harnik *et al.* [4], some focused on one or more of the concrete attacks and developed independent solutions to thwart them without modifying the actual convergent-encryption based secure deduplication solution itself. To stay compact, we don't describe the individual works here but in a follow-up Section 3.2.

The facts that most of the teams published their results labeled as "secure deduplication" schemes and solutions independently of whether they were addressing one/some/all of the attacks described by Harnik *et al.*, improving convergent encryption or coming with an altogether novel secure deduplication solution, and that many of the "secure deduplication reviews" address all of the works together independently on their actual targets confused the field a bit. To clarify our understanding of the "secure data deduplication" term we first describe the principal weakness exploited by the attacks and then define a few categories.

3.1.2 The "Weak Point" of Deduplication

For deduplication to work there has to be a mechanism that compares the data to find out whether or not the same data was already stored before and thus can be deduplicated. In our model of deduplication, the indexing function $\mathcal{I}$ is such a mechanism (see Section 2.1.1). From a security standpoint, this mechanism must be perceived as information leakage and forms the core weak point of simple deduplication, causing it to be generally insecure.

Independently of whether deduplication (deduplication index computation) is done by the client-side or by the server-side, the same principal "weakness" in secure deduplication solution based solely on convergent encryption lies in the deterministic nature of the index and of the encryption key, respectively. If the encryption keys were chosen randomly (independently of the file contents), both the ciphertexts and the deduplication indexes for the same file encrypted twice would differ and deduplication would not work. Convergent encryption "fixes" this by enforcing the encryption key (and thus also the deduplication index) to be always the same for files with the same contents – key is deterministically computed from file contents, encryption itself is deterministic and index is, also deterministically, computed from the ciphertext. This creates a seemingly impossible situation – a deduplication index deterministically dependent on the file contents and known/shared with the storage provider must exist for deduplication to work yet the dependency of the index on the file contents creates possibility of contents leakage and thus must be eliminated.

Harnik *et al.* [4] focused in their analysis on the client-side (source-based) deduplica-

tion only to stress the severity of the attacks (any user or even just a traffic observer can mount the attack). Another valid argument was that source-based deduplication saves not only storage space but also bandwidth, which is getting more and more important with the proliferation of mobile devices and mobile Internet connection. We agree with the arguments presented, though we stress that exactly the same attacks can be mounted by the storage provider himself in case of server-side (target-based) deduplication. Technically, all the attacks can be prevented for a "normal" attacker (*i.e.* not the storage provider) by using target-based instead of source-based deduplication. However, there would still be the inquisitive doubts – can we trust the storage provider? And even if so, should we do it?

3.1.3 Our Secure Data Deduplication Targets

When forming our view of secure data deduplication, we started with the evaluation of "typical user needs". Users own a lot of data that they want to store in a cloud storage for various reasons (*e.g.* backup copies, ease of accessing data from multiple machines/in multiple places). Some of the data is of personal nature and the users do not want anyone to be able to access this data apart from them (*e.g.* family photos, payslips), some of the data may be top secret (*e.g.* medical documentation, work files), some might be of no big concern (*e.g.* application binaries, music hits). Users can be motivated to use a storage service that uses data deduplication, even accepting to do some extra computations on their side if it saves them time and/or money (*e.g.* lower prices for deduplicated data, bandwidth savings by not uploading deduplicated data) but they require confidentiality for their personal and secret data. Users do not blindly trust the storage provider that he will not browse/share their data and want to have some provable mechanism for protecting data confidentiality even from the storage provider himself. Additionally, users should not be forced to manually classify files as confidential or non-confidential as that could be a very tedious task, especially in backup scenarios.

Getting back to the weak point of deduplication – for deduplication to work, data must be compared for equality but such a comparison always leaks information about the data. Encrypting the data convergently does not help, encrypting the data non-deterministically breaks deduplication. Simply put, data that require perfect confidentiality (in the sense of semantic security [13]) should never be deduplicated, since deduplication inherently breaks perfect confidentiality (a comparison mechanism *i.e.* a leakage has to exist). This impossibility result seems obvious and was even formally described by Bellare *et al.* [7]. Apart from the impossibility result, Bellare *et al.* described and formalized security notions that are actually achievable with convergent encryption-like solutions. Unfortunately, viewed from our typical user perspective, these notions are simply not strong-enough for

confidential data – non-predictability is quite a hard assumption to accept generally.

Considering user requirements together with the impossibility result, we reached a conclusion that a secure deduplication solution would need to make sure that confidential data is never deduplicated and is protected by a semantically secure encryption (to prevent any information leakage; apart from data size/length – we discuss this limitation in detail later in the work). For the solution to actually also be "deduplication" and not just "secure", we also need a way to distinguish between confidential and non-confidential data and make sure the non-confidential data gets deduplicated (if possible). Since we specified in the user requirements that the user should not be forced to classify his/her data manually, our target is to design a solution that would automatically distinguish between confidential and non-confidential data and protect the confidential data on the "semantic security level", while protecting the non-confidential data on the "convergent encryption level" that allows deduplication (more formal security definitions are available in Section 6).

To achieve our target we construct a multi-layered encryption scheme exploiting the popularity principle to distinguish between popular ("widely-known") and thus non-confidential and unpopular and thus potentially confidential data (Section 4). As noted in the introduction of this section, since other teams researching secure deduplication set different views and targets for their secure deduplication solutions, we provide extensive comparison between our scheme and a few other representative secure deduplication works both in terms of security and performance to demonstrate the similarities and differences (Sections 6 and 7).

3.2 State of the Art – Related Work

3.2.1 Deduplication (Without Security)

Data deduplication is a very popular storage space optimization technique. Same as other such techniques, the efficiency of deduplication is highly dependent on the dataset it is being applied on – deduplication is totally ineffective for datasets composed solely of unique data with no duplicates and can be very efficient for scenarios where datasets contain a lot of duplicate data. A compact description of how to measure deduplication efficiency and what it depends on was provided by Dutch [2]. Harnik *et al.* [14] commented on the complexity of deduplication efficiency estimation and presented a novel algorithm to estimate it, providing both formal and empirical results for their approach.

Many research works focused on the analysis of efficiency of deduplication itself, omitting its (in)security. Meister and Brinkmann [15] studied the influence of different chunk-

ing approaches on multiple levels and analyzed how changes in the dataset influence deduplication efficiency based on the chosen chunking approach. Mandagere *et al.* [16] provided a comprehensive deduplication taxonomy and compared the different deduplication approaches using real-world data, measuring not only deduplication efficiency but also consumed system and time resources.

Aronovich *et al.* [17] focused on optimization of known deduplication strategies and proposed a novel similarity-based deduplication system. Instead of using classical hash-based indexing function for data comparison, the proposal uses a more cost-effective solution based on similarity signatures and discusses the possibilities how to combine this new approach with the existing one to optimize deduplication systems.

Zhao *et al.* [18] present a novel scalable deduplication file system for virtual machine images. This work demonstrates that there are various ways of introducing deduplication into cloud deployments to achieve better resource utilization.

3.2.2 Convergent Encryption

The first work that considered security in a deduplication setting was published by Douceur *et al.* [3]. While the primary goal of the published research was to reclaim space from duplicate files in a distributed file system, the work focused also on the possibility how to combine data confidentiality (*i.e.* encryption) and deduplication and introduced convergent encryption to be used for this purpose. It is worth noting that the information leakage identified as exploitable later by Harnik *et al.* [4] was openly accepted by Douceur *et al.*– their formal security proof was stating, that "convergent encryption deliberately leaks a controlled amount of information" and they only proved that "we are not accidentally leaking more information than we intend". The leakage was not seen as a security issue.

Storer *et al.* [19] published a paper introducing two secure data deduplication models – authenticated and anonymous. The protection mechanism of choice in both cases was convergent encryption and, unfortunately, the security analysis was focusing on adversaries attacking the storage service itself rather than analyzing the properties of the underlying deterministic convergent encryption. The information-leakage-based attacks were not considered.

Deduplication research took a major turn towards security in 2010 when Harnik *et al.* [4] presented how information leakage caused by client-side (source-based) deduplication can be exploited to mount three types of attack – identifying files, learning file contents and misusing a deduplication-enabled storage as a covert communication channel.

To prevent inadvertent omissions of risks stemming from convergent encryption, Bel-

lare *et al.* [7] formalized convergent encryption under the name *message-locked encryption*. As expected, the security analysis presented in [7] highlights that message-locked encryption offers confidentiality for unpredictable messages only, clearly failing to achieve semantic security.

While not strictly security-oriented, Lou *et al.* [20] noticed that key management in the convergent encryption setup can be quite challenging and aims to solve the problem in an easy-to-handle way. The presented solution solves the key management issue quite neatly though it does not address the security weakness of convergent encryption in any way.

3.2.3 Secure Deduplication Solutions (post convergent encryption only)

Bellare *et al.* present DupLESS [5], a server-aided encryption for deduplicated storage. Similarly to our proposed solution, DupLESS uses a modified convergent encryption scheme with the aid of a secure component for key generation. In DupLESS, clients encrypt files using message-based keys obtained from a key-server via an oblivious PRF protocol. This approach enables clients to store encrypted data with an existing storage service, have the service perform deduplication on their behalf, and yet achieves strong confidentiality guarantees.While DupLESS offers the possibility to securely use server-side (target-based) deduplication, our scheme aims at secure client-side (source-based) deduplication to take advantage of the bandwidth savings.

Puzio *et al.* [21] present ClouDedup, a solution based on convergent encryption strengthened by additional encryption provided by a trusted metadata manager. To achieve as best efficiency as possible, ClouDedup offers fine-grained block-level deduplication and addresses the related issue of key management by introducing another component handling key management together with the actual per-block deduplication. 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 to prevent existence of a single point of potential data leakage.

Armknecht *et al.* present ClearBox [22], a gateway-aided encryption for deduplication storage with built-in Proof of Ownership mechanism and transparent deduplication pattern attestation. ClearBox extends the primary goal of securing data privacy by enabling cloud users to verify the effective storage space that their data is occupying in the cloud, and consequently to check whether they qualify for benefits such as price reductions, etc. 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.

Liu *et al.* [23] present a solution based on a password-authenticated key-exchange (PAKE) protocol to alleviate the need for a trusted third party. In a PAKE setup the deduplication information is distributed among the users of the system and the users are required to be on-line to participate in the deduplication process. This setting notably limits the processing that needs to be handled by the storage provider and prevent an honest but curious storage provider from getting contents of files stored by the users yet it puts more processing and requirements on the users, needing them to participate when deduplication of files that they also stored is required. Our solution aims to minimize the necessity of user involvement, even though it does require a trusted third party to work.

Meye *et al.* [24] present a two-phase data deduplication scheme trying to solve the known convergent encryption issues. While the work methodically addresses each issue and presents per-issue solutions including proofs, the proposed solutions are often quite inefficient with respect to practical usability (e.g. masking deduplication with bogus transfer). We strive to address all the issues in a compact one-solution way.

Duan [25] presents a solution similar to that of DupLESS [5] but instead of a one-component key server he suggests using a threshold scheme for pseudo-convergent key generation. Using a threshold component is similar to our proposal, even though it is used for a different purpose.

Xu *et al.* [26] present a Proof of Ownership-incorporating secure deduplication scheme allowing client-side deduplication in a weak-bounded-leakage setting. Security proof in a random oracle model for the solution is provided, however, the weak leakage setting is a quite strong assumption and low-entropy files are not being addressed. Our solution aims specifically to cover also the low-entropy files that pose the greatest risk in the "learning file contents" attack.

Li *et al.* [27] present a solution based on hybrid cloud – trusted private cloud is used for sensitive metadata management and public cloud is used for the actual data storage. The solution provides authorized deduplication, adding user roles and priviliges to the standard deduplication setting and additionally incorporates also the Proof of Ownership mechanism. While provably secure in the presented security model, the solution requires trusted private cloud and does not eliminate the vulnerabilities pointed out by Harnik *et al.* [4] for users with overlapping privileges. Our solution aims to eliminate the mentioned vulnerabilities for unpopular files.

Chia-Mu [28] presents two novel variants of secure deduplication schemes abbreviated SDedup and XDedup. The schemes combine the approach of DupLESS [5] with the approach of Liu *et al.* [23] and are based on a Merkle puzzle. While being secure against user-adversaries, the cloud storage provider is given the deduplication index of the stored

files allowing him to mount attacks based on this information.

3.2.4 Proofs of Ownership (PoW)

Apart from attacks identified by Harnik *et al.* [4] one more attack vector against source-based deduplication was identified by Halevi *et al.* [29] – deduplication allows to obtain a potentially large file stored by a legitimate user by presenting only the short deduplication index. Since deduplication identifies files via indexes, a user that does not own a file but possesses its deduplication index can falsely ask to store the file identified by this index. Since deduplication is enabled, the actual file upload would not be required and the user will become one of the legitimate owners of the file and can thus download it from the storage. Note that the same weakness can be used to misuse a deduplication-enabled storage as a content distribution network (upload file once, distribute hash, download many times). To thwart such attacks, Halevi *et al.* [29] introduced the concept of Proof of Ownership (PoW) – the user is first required to prove that he really has the file before the file is deduplicated and he is accepted as its rightful owner.

The proofs of ownership were researched e.g. by DiPietro and Sorniotti [30], different variants were suggested and optimized [31]. Note that while mitigating another deduplication weakness, proofs of ownership are somewhat parallel to the "core" secure deduplication research as they do not target the issue of end-user data confidentiality. Indeed, PoWs can be often used along the other listed secure deduplication solutions (with some exceptions that already bundle them in the solution, such as the works by Armknecht *et al.* [22], Xu *et al.* [26] or Li *et al.* [27]).

Chapter 4

Proposed Solution – Principles and Overview

This chapter provides a compact overview of our proposed secure data deduplication solution, summarizing the requirements it tries to satisfy and the principles employed to do so. We start with a short explanation of suitability of our solution for different scenarios and follow with a description of the core principle our solution builds upon and how the whole proposed secure data deduplication scheme is organized.

4.1 Introduction

As the analyses demonstrated (*e.g.* [2, 14]), efficiency of each deduplication system is highly dependent on the actual type (and structure) of data expected to be stored. Our secure data deduplication solution was designed with the idea of popular multi-user cloud data storages in mind but is generally applicable in many scenarios where cross-user deduplication is deployed. In this respect, our scheme was tailored specifically for datasets likely to contain *relatively few instances of some data items and many instances of others* which is what we expect from a typical multi-user cloud data storage. Other scenarios where our solution is expected to have good efficiency are *e.g.* those working with datasets generated by backup tools for multiple machines and users, hypervisors handling linked clones of VM-images for many clients and alike. Notice that the more duplicates there are in the dataset, the better efficiency of our solution is. On the other hand the efficiency of our solution would be very low for scenarios with low numbers of users or with low presence of duplicates. We analyse the efficiency in-detail in Section 7.

The main intuition behind our scheme is that, in general, different data require different degrees of protection. Indeed, organizations such as governments, agencies and companies typically do have a classification system that categorizes every file based on

its contents into a clearly-defined class (*e.g.* top secret, confidential, public) and do have policies corresponding to these categories (*e.g.* top secret must be always encrypted by AES-256-CBC in transit and storage and must never leave the secure perimeter of the intranet; public is published on the web in documents archive). Unfortunately, in the case of user-managed data there are typically no classes and no universal metrics, recommendations or algorithms that could be used to categorize data. Even worse, it is practically impossible to design such sorting mechanisms based on the data contents only since the required degree of protection is typically unique per-file. To give an example – a computer game configuration file would likely be less important than a personal photo, unless the file was tuned by the gamer specifically to give him an edge in the game and the photo is already published on multiple social networks, in which case the importance is reversed. To solve this "unsolvable" issue, we propose a popularity principle for categorization – instead of deciding based on data contents, we suggest deciding based on how *popular* a datum is (*e.g.* how many users already own it).

The *popularity principle* is an implementation of the saying that "open secret is not a secret". Phrased differently, if "almost everyone already knows *it*", there is no point in trying to keep *it* secret. Based on this principle, data can be either categorized as *unpopular* (known/owned by *a few* users only) which require strong protection or as *popular* (known/owned by *many* users) which do not require such a strong protection (still some protection can be useful; even popular data must not be "given for free to anyone"). In the world of multi-user cloud storage services, the implementation is straightforward in theory – protect the data "strongly" until they are uploaded by "enough" users and then weaken the protection. And that is exactly what we propose in our secure data deduplication scheme: defining what is "strong protection", "weak protection" and how to set the "engouh" a bit more formally.

To implement the data classification idea and the popularity principle in our secure data deduplication solution we use a cryptographic construct called *multi-layered* cryptosystem. All files are initially declared unpopular and are encrypted with two layers, as illustrated in Figure 4.1: the inner layer is applied using *convergent* encryption ("weaker encryption"), whereas the outer layer is applied using a semantically secure cryptosystem ("stronger encryption"). 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. The decryption shares are stored together with their convergent index (*i.e.* hash of their convergent ciphertext) by a trusted third party (TTP). 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:

1. security notion for the now popular file is downgraded from semantic to standard

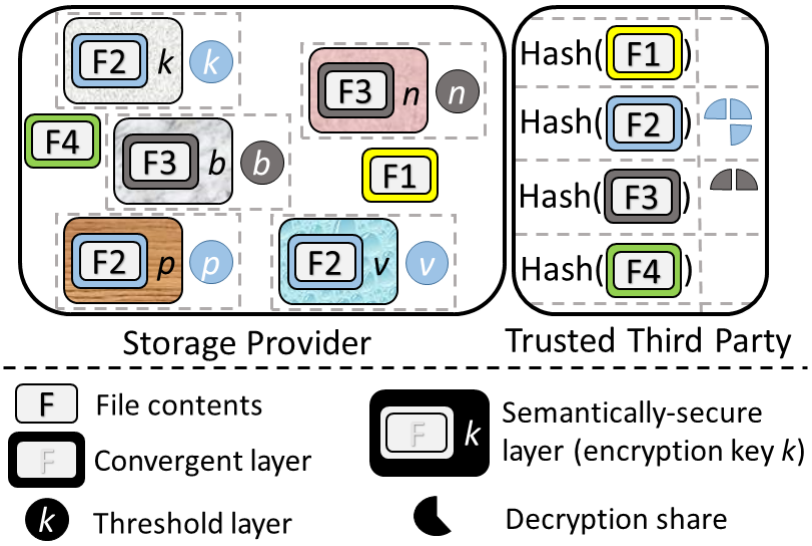

Figure 4.1: The multi-layered cryptosystem used in our scheme. Unpopular files (F2 and F3) are protected using two layers, whereas for popular files (F1 and F4), the outer layer can be removed. The inner layer is obtained through convergent encryption that generates identical ciphertext at each invocation on the same file. The outer layer (for unpopular files) is obtained through a semantically secure cryptosystem.

convergent (see [7]), and

2. properties of the remaining convergent encryption layer allow deduplication to happen naturally.

Security is thus traded for storage efficiency, as for every file that transits from unpopular to popular status, storage space can be reclaimed. Once a file reaches the popular status, space is reclaimed for the copies uploaded so far, and normal deduplication can take place for future copies.

There are two further challenges in the secure design of this scheme. Firstly, without proper identity management, Sybil attacks [32] 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 (source-based) 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 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. We tackle these issues by introducing two additional entities in our system model.

4.2 System Model

Our solution focuses primarily on the scenario of multi-user cloud storage services and requires strict user authentication and obfuscation of the deduplication index (see Section 4.1). Our system model is composed of:

- **users** of the system (physical beings)
- **a set of user identities** $\mathsf{U}_i \in \mathcal{U}$ identifying users participating in the system (using the cloud storage service)
- **storage provider** S offering storage services
- **identity provider (IdP)** is a trusted third party (TTP) deploying a strict user identity control and hinders Sybil attacks
- **index repository service (IRS)** is a TTP providing secure indexation for unpopular files

A user who wants to start using the storage service is required to register by the IdP to be assigned a user identity U_i used during interaction with other system participants. As the user, as physical entity, is not identified by any other means than by user identity U_i (in this work), we often simply refer to "user U_i" meaning the user that was assigned user identity U_i by IdP when joining the system. The actual implementation of IdP or concrete structure of the user identity are transparent from the system point of view and can be implemented by any existing registration mechanism. The only requirement is that IdP does not assign multiple identites to one physical user. Similarly, S stands for the storage provider, but is often used meaning the storage provider service as a whole.

Once the user joins the system he can start using the storage service. A file is identified within S via a unique file identifier ($\mathcal{I}$), which 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 stores also the associated decryption shares. The concrete algorithms provided by the system are described in Section 5, an overview of user interaction during the registration process and during an unpopular file upload are shown in Figure 4.2.

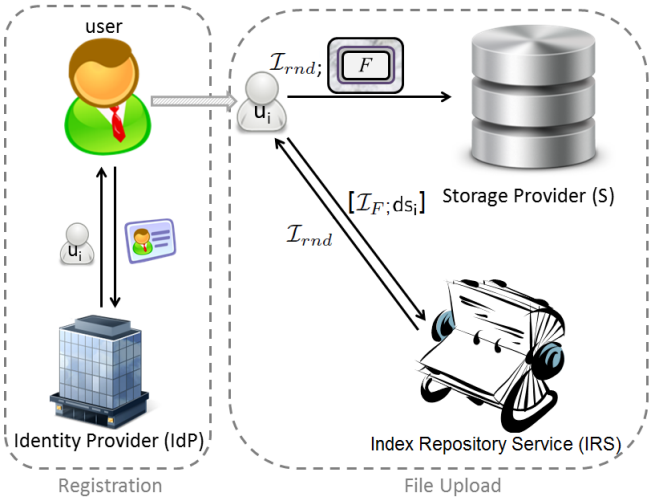

Figure 4.2: Illustration of our system model. The schematic shows the main four entities and their interaction for registration and unpopular file upload process.

4.3 Security Model

The primary objective of our scheme is the confidentiality of user content. Specifically, two different security notions, depending on the nature of each datum, are achieved:

1. *Semantic security* [13] for unpopular data;
2. *Conventional convergent security* [7] for popular data.

Note that integrity and data origin authentication exceed the scope of this work. The issue of deduplication index misuse to fake data ownership is also not addressed inherently by our scheme, but many of the proposed proof of ownership solutions [29–31] are compatible with our scheme and can be deployed to address this issue.

In our security 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 that the adversary can control (corrupt) 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.

To be able to model security formally, we need to first 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 of 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 in our work, extension to multiple thresholds is possible quite straightforwardly albeit for the cost of performance decrease. The simplest way of relaxing the single-threshold requirement would be to create multiple instances of the scheme, each with different values of t, and issue as many keys to each user. Naturally, the higher the value of threshold t, the more copies are required for a file to become popular. Storage provider could use this difference *e.g.* to set different pricing levels (higher t means more storage required, means more expensive). Users are then free to choose the threshold that best fits their need (*e.g.* highest for financial data, lowest for common stuff etc.). Note that a file uploaded with a given threshold t_1 does not count towards popularity for the same file uploaded with a different threshold t_2 (necessary performance decrease since otherwise malicious users could easily compromise the popularity principle to extract information of a file encrypted with higher threshold using the lower threshold). Additionally, a label identifying the chosen threshold (which does not leak other information) must be uploaded together with the ciphertext and 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.

The IRS and IdP are modeled as trusted third parties (TTP) and thus 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 and discuss possibilities how to alleviate this limitation in Section 6.

Chapter 5

Proposed Solution – Algorithms and Implementation

In this chapter we describe a secure deduplication scheme based on the concept of popularity. Our scheme guarantees semantic security for unpopular data (where deduplication is not possible) and enables their automatic transparent transition to only convergently encrypted (and thus deduplicable) popular data.

The core of the scheme is formed by a threshold convergent cryptosystem $\mathcal{E}_\mu$ described in Section 5.1 and two trusted third parties – the identity provider IdP and the index repository service IRS described in Section 5.2. The scheme as a whole is presented in Section 5.3 together with 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 t) 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 2.2.4, the Encrypt interface now includes the added label ℓ and the decryption process is designed to be non-interactive. Additionally, modification of the DShare interface is required – the decryption share is created using the label l instead of using the ciphertext c and is stored in some repository (separately from the ciphertext) until required for decryption. Note that the modifications enable multiple *different* plaintext messages to be associated with the same label during encryption and thus the same decryption shares (once enough are collected) can be used to decrypt all of them. This makes the cryptosystem flexible and usable in different scenarios, even though we do not use this property in our scheme.

For the purpose of the $\mathcal{E}_\mu$ 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 λ is a security parameter which corresponds to the bitlength of the exploited symmetric encryption scheme key and determines the bitsize of prime q (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 [33, 34]). $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are pairing groups satisfying the Symmetric eXternal Diffie–Hellman (SXDH) assumption [35]; $\mathbb{G}_1 = \langle g \rangle$, $\mathbb{G}_2 = \langle \bar{g} \rangle$ are of prime order q, and $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ 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 $\mathbb{G}_2$. We will also use two cryptographic hash functions $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \mathbb{G}_T \to \{0,1\}^\lambda$, a semantically secure symmetric cryptosystem $\mathcal{E}$ and a convergent encryption scheme $\mathcal{E}_c$. t is used to denote the threshold *i.e.* the number of decryption shares necessary to decrypt a ciphertext (both decryption shares and ciphertext must be generated using the same label ℓ).

Cryptosystem $\mathcal{E}_\mu$ is defined as a tuple (Setup, $\mathsf{Encrypt}$, DShare, $\mathsf{Decrypt}$), consisting of four probabilistic polynomial-time algorithms (in terms of a security parameter 1^λ) with the following properties:

$\mathcal{E}_\mu.\mathsf{Setup}\big(\lambda, n, t\big) \to \big(\mathsf{pk}, \mathsf{sk}, \big\{\big(\mathsf{r_i}, \mathsf{sk_i}\big)\big\}_{i=1}^{n}\big)$: at first, Λ is generated as described in the previous paragraph. Next, let secret key $\mathsf{sk} \leftarrow_R \mathbb{Z}$ and generate n key shares $\big\{\big(\mathsf{r_i}, \mathsf{sk_i}\big)\big\}_{i=1}^{n}$ such that any set of t shares can be used to reconstruct sk [6]. Also, let $\bar{g}_{\text{pub}} \leftarrow \bar{g}^{\mathsf{sk}}$. Public key pk is set to $\big\{\Lambda, H_1, H_2, \bar{g}_{\text{pub}}\big\}$

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

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

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

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

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 $H_1(\ell)^{\mathsf{sk}}$.

Thanks to the properties of bilinear pairings it holds that:

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

Setting $x = \mathsf{sk}$, we get:

$$\hat{e}\left(H_1(\ell)^{\mathsf{sk}}, \bar{g}^r\right) = \hat{e}\left(H_1(\ell), \bar{g}\right)^{r\mathsf{sk}} = \hat{e}\left(H_1(\ell), \bar{g}^x\right)^{\mathsf{sk}}$$

Using $c_1 = H_2\big(E\big) \oplus m$, $c_2 = \bar{g}^r$ and $H_1(\ell)^{\mathsf{sk}}$ we can now decrypt m by computing $\hat{E}$ as $\hat{e}\left(H_1(\ell)^{\mathsf{sk}}, c_2\right)$ and $m = c_1 \oplus H_2(\hat{E})$.

This equality satisfies considerations on the correctness of $\mathcal{E}_\mu$.

$\mathcal{E}_\mu$ has a few noteworthy properties:

1. The decryption algorithm is non-interactive, meaning that it does not require live participation of the entities that executed the $\mathcal{E}_\mu.\mathsf{DShare}$ algorithm;
2. It mimics convergent encryption in that the decryption shares are deterministically dependent on the plaintext label. However, in contrast to plain convergent encryption, the label does not need to be related to the actual message being encrypted in any way and if it is not (such as in our scheme presented in the next section), it cannot leak any information about it;
3. The cryptosystem can be reused for an arbitrary number of messages, *i.e.*, the $\mathcal{E}_\mu.\mathsf{Setup}$ algorithm should only be executed once.

Finally, note that it is possible to generate more shares sk_j $(j > n)$ anytime after the execution of the $\mathcal{E}_\mu.\mathsf{Setup}$ 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 $\mathcal{E}_\mu.\mathsf{Setup}$. 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 [32] 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: $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}].\mathsf{ctr}$ - counter keeping track of how many different users uploaded data to this record; $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_{F_c}].\mathsf{idxes}$ - random index used by the user to identify data corresponding to encrypted F_c in the storage provider space; $\mathsf{DB}_{\mathsf{IRS}}[\mathcal{I}_F].\mathsf{dshares}$ - decryption shares associated with $\mathcal{I}_{F_c}$. The array is initialized empty and the records are added according to GenSecIdx implementation (Figure 5.1). The associative array is indexed by $\mathcal{I}_F$ defined in Section 4.2. $\mathcal{I}_F$ is then disclosed to the storage 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 (Figure 5.1) to the user – for a (leaky) convergent index $\mathcal{I}_{F_c}$ the user is given a random index $\mathcal{I}_{\text{rnd}}$. Note that if the user invokes GenSecIdx more times with the same $\mathcal{I}_{F_c}$ he will always get the same index $\mathcal{I}_{\text{rnd}}$ and the popularity counter will not increase. This prevents potentially corrupted user to force state transition of an unpopular file. Another, albeit much more limited, leakage could occur through the decryption shares used in $\mathcal{E}_\mu$– encryptions using the same ℓ (and pk) by the same user U_i produce equal decryption shares, thus leaking information that the ciphertexts were encrypted using the same label. 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 (Figure 5.2).

```
IRS:            if (DB_IRS[I_Fc].ctr ≥ t)
IRS ⟶ U_i:        return I_Fc
IRS:            I_rnd ← PRF(σ, U_i||I_Fc)
                if (I_rnd ∉ DB_IRS[I_Fc].idxes)
                  increment DB_IRS[I_Fc].ctr
                  add I_rnd to DB_IRS[I_Fc].idxes
                  add (r_i, ds_i) to DB_IRS[I_Fc].dshares
                if (DB_IRS[I_Fc].ctr = t)
IRS ⟶ S:          Deduplicate(DB_IRS[I_Fc].idxes,
                              DB_IRS[I_Fc].dshares)
IRS ⟶ U_i:      return I_rnd
```

Figure 5.1: The $\mathsf{GenSecIdx}(\mathcal{I}_{F_c}, (\mathsf{r_i}, \mathsf{ds_i}))$ algorithm. Popularity is evaluated by comparison of per-index counter and threshold t. Behaviour corresponding to a popular file index is highlighted 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. The complementary Deduplicate implementation is described in Section 5.3.

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

Figure 5.2: The $\mathsf{RemDShare}(\mathcal{I}_{F_c}, \mathcal{I}_{\text{rnd}}, \mathsf{r_i})$ algorithm.

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 $\mathcal{I}_{\text{rnd}}$ (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 users U_i, a storage provider S and the two trusted entities, the identity provider IdP and the index repository service IRS.

S is modeled as an indexed associative array DB_{S} supporting the Put, Get and Delete operations, same as the deduplication model described in Section 2.1. 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. 5.3.

$\mathcal{E}$ is a semantically secure (indistinguishable under chosen-plaintext attack; IND-CPA) symmetric cryptosystem and $\mathcal{E}_c$ is a convergent encryption scheme (see Section 2.2). $\mathcal{E}_\mu$ is our convergent threshold cryptosystem.

When a new user 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

IRS			
index	ctr	idxes	dshares
1A35BC127CC36958	51	∅	∅
82090A161718192A	2	{A112927132910012, C05B228C48371BC7}	{(1302;1A85227..), (0124;545D114..)}

S		
index	data	owners
1A35BC127CC36958	1B100510955476AC4125..	0015,0098,1023,..
A112927132910012	DD845A3362C5487FF14..	1302
C05B228C48371BC7	1CAA5767052A4443720..	0124

Figure 5.3: Examples of S and IRS records. Note that threshold t is set to 50 for this example. Index starting 1A represents a popular file, Index starting 82 an unpopular file.

key share $\mathsf{sk_i}$ (generating a brand new $\mathsf{sk_i}$ if necessary). From this point onwards, the user becomes the user U_i towards other participants in 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. To keep complexity at a reasonable level 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 algorithm is run only once throughout the whole lifetime of the scheme. Invoking Init again corresponds to deployment of a new scheme.

Once the scheme is initialized, all interaction is always started by the user invoking once of the three following algorithms:

Upload(F, U_i)[Fig. 5.4]: The user U_i encrypts file F convergently and generates the index $\mathcal{I}_{F_c}$ which he uses in request to IRS.

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*. The user then 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 transition from unpopular to popular state is indeed triggered by the Upload algorithm but does not influence its actual processing. Upload is always processed the same way as described in Fig. 5.4. If the file state transition (deduplication) is to take place it is triggered during GenSecIdx processing and managed by Deduplicate (see Fig. 5.1 and Fig. 5.7)

Note that additional strengthening measures can be deployed at S to improve the security provided (such as PoW [29] that allow checking, whether the user really owns the popular file he tries to upload).

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

Figure 5.4: The $\mathsf{Upload}(F, \mathsf{U}_i)$ algorithm. Popular file upload part is highlighted in green (lighter color), unpopular file upload part in blue (darker color).

$\mathsf{Download}(F, \mathsf{U}_i)$[Fig. 5.5]: 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 the $\mathcal{I}_{\text{ret}}$ index location. 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.

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

Figure 5.5: The $\mathsf{Download}(F, \mathsf{U}_i)$ algorithm. Unpopular file download part is highlighted 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).

$\mathsf{Remove}(F, \mathsf{U}_i)$[Fig. 5.6]: 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 $\mathsf{Delete}(\mathcal{I}_{F_c}, \mathsf{U}_i)$.

Note that a file that is popular can never become unpopular again and the Remove algorithm is designed to prohibit such a transition. Allowing a popular file to become unpopular would break the scheme security properties.

Notice that secure deletion of content requires storage provider S to be trusted, while it is "honest but curious" in our setting. Therefore S may well pretend to delete the actual content, and yet store it for later information extraction (notice that this makes sense for unpopular files only). However, the index repository service, which is a trusted entity, would perform the deletion step honestly by removing the random index and the corresponding decryption share generated for the file and decreasing the popularity. This alone however does not guarantee any security. Indeed, we may be faced with the scenario in which the popularity threshold has not yet been reached (that is, the storage provider has not been given the set of indexes), and yet more than t encrypted contents exist at unknown locations (since S didn't delete some of them when it properly should). However, since S does not know which indexes are the "right ones" and the ciphertexts cannot be linked by any means (no information can be extracted, as we prove in section 6), the only gain for S for not deleting the file contents when supposed to is to have some storage occupied by useless data (useless since the legitimate user who wanted to download it deletes his local copies of decryption keys and thus the encrypted data become irrecoverable).

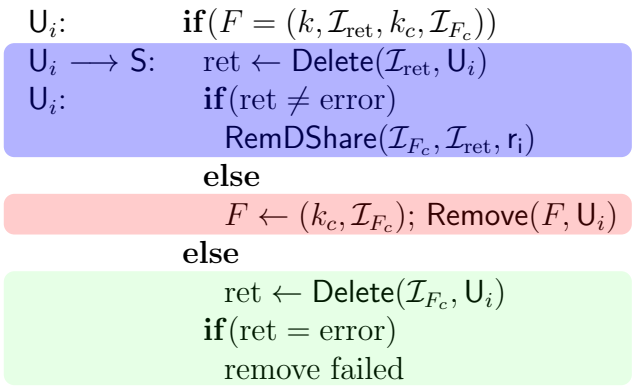

Figure 5.6: The $\mathsf{Remove}(F, \mathsf{U}_i)$ algorithm. Unpopular file remove part is highlighted in blue (the darkest color), part corresponding to file switch caused by the file being uploaded as unpopular but changed status to popular before removal is in red and popular file remove part is in green (the lightest color).

Deduplicate(idxes, dshares)[Fig. 5.7]: 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, computes the "convergent index" $\mathcal{I}_{F_c} = \mathcal{I}(F_c)$, 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.

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

Figure 5.7: The Deduplicate(idxes, dshares) algorithm.

Chapter 6

Security Analysis

In this section we focus on the security aspect of our proposed secure deduplication scheme. Since the proposed scheme is quite complex we first list all the cryptographic building blocks (cryptosystems) and their respective usage in the scheme along with the data that are being encrypted by the respective cryptosystems. Next we formally analyze the security of the core building block of our scheme – the $\mathcal{E}_\mu$ cryptosystem. Since the $\mathcal{E}_\mu$ cryptosystem introduces the concept of decryption shares we formally demonstrate that the decryption shares themselves do not leak any information, specifically that it is not possible to find out whether or not decryption shares were created using the same label. Then we analyze the scheme as a whole, analyzing the view of each scheme participant and discuss its corruptability as well as the possible information leakage caused by collusion of corrupted participants. Finally, we provide comparison of security properties of our scheme with other current secure deduplication proposals [5, 21–23].

6.1 Scheme Building Blocks – A Security Overview

Recall from Section 5, the proposed scheme incorporates three different cryptosystems – a semantically secure symmetric cryptosystem $\mathcal{E}$, a convergent encryption scheme $\mathcal{E}_c$ and a novel convergent threshold cryptosystem $\mathcal{E}_\mu$. For a new file F to be processed by our scheme we need to make sure it is first convergently encrypted (*i.e.* file contents encrypted using a convergent encryption scheme) $F_c = \mathcal{E}_c.\mathsf{E}(k_F, F)$ and a deduplication index $\mathcal{I}_{F_c} = \mathcal{I}(F_c)$ is computed over this convergently encrypted file. Next, the convergently encrypted file has to be encrypted using a semantically secure cryptosystem with a random key k_{rnd}, $F_s = \mathcal{E}.\mathsf{E}(k_{rnd}, F_c)$ and the encryption key k_{rnd} used during this encryption must be encrypted using $\mathcal{E}_\mu$: $c = \mathcal{E}_\mu.\mathsf{Encrypt}(\mathsf{pk}, F_c, k_{rnd})$ where pk is the public key of our scheme. Finally, decryption share $\mathsf{ds}_i = \mathcal{E}_\mu.\mathsf{DShare}(\mathsf{r}_i, \mathsf{sk}_i, \ell)$ must be computed to allow later (during deduplication) to decrypt key k_{rnd} and transfer from unpopular file F_s to a

popular file F_c. Note that to make sure that the decryption shares produced by $\mathcal{E}_\mu$ will correspond to the same convergently encrypted file, the whole convergently encrypted file itself is used as a label for $\mathcal{E}_\mu$ encryption and decryption share generation.

From a security standpoint, knowledge of the deduplication index $\mathcal{I}$, convergently encrypted file F_c as well as knowledge of the decryption share ds_i can be considered potential threats if they constitute some leakage (*i.e.* can be used to devise some information about the contents of the original file F). Since $\mathcal{I}$ is deterministically computed from F_c which is deterministically-encrypted with a deterministically-derived key from F, these two surely do constitute a leakage and we eliminate this threat by letting only a trusted third party IRS to know $\mathcal{I}$ for unpopular files (we discuss consequences of IRS corruption later in this section). For popular files the leakage is accepted as reasonable. Whether or not a decryption share constitutes a leakage is not directly clear (recall that F_c is used as a label for $\left(\mathsf{r_i}, \mathsf{ds_i}\right)$ creation). We therefore define a property of $\mathcal{E}_\mu$ called "unlinkability of decryption shares" that makes sure that decryption share does not leak any information about the label it was created with (*i.e.* the convergently-encrypted F_c) and formally prove it.

Note that we do not prove that $\mathcal{E}$ is semantically secure nor that $\mathcal{E}_c$ is convergently secure, as we simply choose such cryptosystems that already meet these requirements. For syntactical completeness we specify that "convergent security" as used in this work corresponds to a formal definition of a PRV$-CDA STC message-locked encryption scheme [7].

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

Differently from classic encryption schemes $\mathcal{E}_\mu$ produces not only ciphertexts but also decryption shares. For a security analysis to be complete, we first address the decryption shares and prove that they are "unlinkable" *i.e.* that they do not leak any information about the label nor secret key share that they were created with. Informally, we show that having multiple different decryption shares, there is no way of saying whether some of them were created with the same label. We can postulate an even stronger property that it is not possible to say whether or not they are valid decryption shares at all. Then we analyse the security of the ciphertexts and show what the attacker is (or is not) able to devise from the ciphertext.

Note that both the unlinkability of decryption shares and the security of the ciphertexts is by design bounded by the number of corrupted users – if the attacker is able to corrupt more than $n_\mathcal{A}$ users (where $n_\mathcal{A}$ must be lesser than $t - p_{\text{lim}} - 1$, see Section 4.3) then the security is broken. This stems from the threshold nature of the cryptosystem and must be always taken into account.

6.2.1 Unlinkability of Decryption Shares

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

Formally, unlinkability of decryption shares is defined using the experiment DS_μ-IND between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, given security parameter 1^λ:

Setup Phase $\mathcal{C}$ executes the $\mathcal{E}_\mu.\mathsf{Setup}$ algorithm with λ, and generates a set of user identities $\mathsf{U} = \{\mathsf{U_i}\}_{\mathsf{i}=0}^{\mathsf{n}-1}$. Further, $\mathcal{C}$ gives pk to $\mathcal{A}$ and keeps $\{\mathsf{sk_i}\}_{i=0}^{n-1}$ secret. At this point, $\mathcal{A}$ declares the list $\mathsf{U}_\mathcal{A}$ of $|\mathsf{U}_\mathcal{A}| = n_\mathcal{A} < t - p_{\lim} - 1$ identities of users that will later on be subject to $\mathcal{O}_{\mathsf{Corrupt}}$ calls.

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

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

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

The following lemma shows that unlinkability of decryption shares is guaranteed in $\mathcal{E}_\mu$ as long as the SXDH problem is intractable [35].

Lemma 6.2.1. *Let* H_1 *and* H_2 *be random oracles. If a* DS_μ*-IND adversary* $\mathcal{A}$ *has a non-negligible advantage* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{DS}_\mu\text{-}\mathsf{IND}} := \Pr[\mathsf{b}'_{m_*} \leftarrow \mathcal{A}(m_*, \mathsf{ds}_{*,m_*}) : \mathsf{b}'_{m_*} = \mathsf{b}_{m_*}] - \frac{1}{2}$ *, then, a probabilistic, polynomial-time algorithm* $\mathcal{C}$ *can create an environment where it uses* $\mathcal{A}$*'s advantage to solve any given instance of the SXDH problem.*

Proof. 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 $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. 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 $\hat{e}$ [35].

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

Setup Phase $\mathcal{C}$ sets $q \leftarrow q'$, $\mathbb{G}_1 \leftarrow \mathbb{G}'_1$, $\mathbb{G}_2 \leftarrow \mathbb{G}'_2$, $\mathbb{G}_T \leftarrow \mathbb{G}'_T$, $\hat{e} = \hat{e}'$, $g \leftarrow g'$; picks a random generator $\bar{g}$ of $\mathbb{G}_2$ and sets $\bar{g}_{pub} = (\bar{g})^{\mathsf{sk}}$, where $\mathsf{sk} \leftarrow_R \mathbb{Z}^*_q$. $\mathcal{C}$ also generates the set of user identities $\mathsf{U} = \{\mathsf{U}_i\}_{i=0}^{n-1}$. The public key $\mathsf{pk} = \{q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T\, \hat{e}, \mathcal{O}_{\mathsf{H}_1}, \mathcal{O}_{\mathsf{H}_2}, \bar{g}, \bar{g}_{pub}\}$ and U are forwarded to $\mathcal{A}$. $\mathcal{A}$ declares the list $\mathsf{U}_\mathcal{A}$ of $n_\mathcal{A} < t - p_{\text{lim}} - 1$ user identities that will later on be subject to $\mathcal{O}_{\mathsf{Corrupt}}$ calls. Let $\mathsf{U}_\mathcal{A} = \{\mathsf{U}_i\}_{i=0}^{n_\mathcal{A}-1}$. 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 $\mathrm{P}(0) = \mathsf{sk}$ and $\mathrm{P}(i) = y_i$ for $i = 1, .., t - p_{\text{lim}} - 1$. $\mathcal{C}$ then sets the key-shares to $(i, \mathsf{sk_i}) \leftarrow y_i, i \in [1, t - p_{\text{lim}} - 1]$ and assigns $(i, \mathsf{sk_i})$ for $i \in [1, n_\mathcal{A}]$ to corrupted users.

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

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

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

$\mathcal{O}_{\mathsf{DShare}}$: simulation of $\mathcal{O}_{\mathsf{DShare}}$ is performed as follows. As before, $\mathcal{C}$ keeps track of the submitted $\mathcal{O}_{\mathsf{DShare}}$ queries in her local memory. Let $\langle \mathsf{U}_i, \ell \rangle$ be a decryption query submitted for label ℓ and user identity U_i. If there is no entry in H_1-list for ℓ, then $\mathcal{C}$ runs the $\mathcal{O}_{\mathsf{H}_1}$ algorithm for ℓ. Let $\{\ell, h_\ell, \rho_\ell, \mathrm{c}_\ell\}$ be the $\mathcal{O}_{\mathsf{H}_1}$ entry in $\mathcal{C}$'s local memory for label l. Let $\mathrm{P}' \leftarrow \mathrm{P} \setminus \mathrm{P}(0)$. $\mathcal{C}$ responds with $\left(\mathsf{r_i}, \mathsf{ds_i}\right)$ where $\mathsf{r_i}$ corresponds to U_i and $\mathsf{ds_i} = \left(g^{\sum_{(\mathsf{r_j}, \mathsf{sk_j}) \in \mathrm{P}'} \mathsf{sk_j} \lambda^{\mathrm{P}'}_{\mathsf{r_i},\mathsf{r_j}}} X^{\lambda^{\mathrm{P}'}_{\mathsf{r_i},\mathsf{r_0}}} \right)^{\rho_m}$ where $X \leftarrow A$ iff $\mathrm{c}_\ell = 0$, and $X \leftarrow W$ iff $\mathrm{c}_\ell = 1$. In both cases, $\mathcal{C}$ keeps a record of her response in her local memory.

Challenge Phase $\mathcal{A}$ selects the challenge label ℓ_*. Let the corresponding entry in the $\mathcal{O}_{\mathsf{H}_1}$ list be $\{\ell_*, h_{\ell_*}, \rho_{\ell_*}, \mathrm{c}_{\ell_*}\}$. If $\mathrm{c}_{\ell_*} = 0$, then $\mathcal{C}$ aborts.

Guessing Phase $\mathcal{A}$ outputs one bit b'_{ℓ_*} representing the guess for b_{ℓ_*}. $\mathcal{C}$ responds positively to the DDH challenger if $\mathsf{b}'_{\ell_*} = 0$, and negatively otherwise.

It is easy to see, that if $\mathcal{A}$'s answer is $'0'$, it means that the $\mathcal{O}_{\mathsf{DShare}}$ responses for ℓ_* constitute properly structured decryption shares for ℓ_*. This can only be if $W = g^{ab}$ and $\mathcal{C}$ can give a positive answer to the SXDH challenger. Clearly, if $\mathrm{c}_{\ell_*} = 1$ and $\mathrm{c}_\ell = 0$ for all other queries to $\mathcal{O}_{\mathsf{H}_1}$ such that $\ell \neq \ell_*$, the execution environment is indistinguishable from the actual game DS_μ-IND. 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 distinct $\mathcal{O}_{\mathsf{H}_1}$ queries. By setting $\delta \approx \frac{1}{\mathcal{Q}_{H_1}+1}$ the above probability becomes greater than $\frac{1}{e \cdot (\mathcal{Q}_{H_1}+1)}$ and the success probability of the adversary can be bounded as $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{DS}_\mu\text{-}\mathsf{IND}} \leq e \cdot (\mathcal{Q}_{H_1}+1) \cdot \mathsf{Adv}^{\mathcal{C}}_{\mathsf{SXDH}}$. □

6.2.2 Indistinguishability of the $\mathcal{E}_\mu$ Ciphertexts

Similarly to the decryption shares, we need to prove that ciphertexts do not leak any information about the plaintext unless, of course, there are more than t eligible users which would lead to decryption and plaintext revelation.

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 [36].

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 $n_{\mathcal{A}} < t - p_{\text{lim}} - 1$ of pre-declared users, and obtains their secret keys through an oracle $\mathcal{O}_{\mathsf{Corrupt}}$. 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 $\mathcal{E}_\mu.\mathsf{Setup}$ algorithm with λ, and generates a set of user identities $\mathsf{U} = \{\mathsf{U_i}\}_{\mathsf{i=0}}^{\mathsf{n-1}}$. Further, $\mathcal{C}$ gives pk to $\mathcal{A}$ and keeps $\{\mathsf{sk_i}\}_{i=0}^{n-1}$ secret. At this point, $\mathcal{A}$ declares the list $\mathsf{U}_\mathcal{A}$ of $|\mathsf{U}_\mathcal{A}| = n_\mathcal{A} < t - p_{\text{lim}} - 1$ identities of users that will later on be subject to $\mathcal{O}_{\mathsf{Corrupt}}$ calls.

Access to Oracles Throughout the game, the adversary can invoke oracles for the hash functions H_1 and H_2. Additionally, the adversary can invoke the corrupt oracle $\mathcal{O}_{\mathsf{Corrupt}}$ and receive the secret key share that corresponds to any user $\mathsf{U}_i \in \mathsf{U}_\mathcal{A}$.

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 [35].

Lemma 6.2.2. *Let* H_1*, and* H_2 *be random oracles. If an* IND_μ*-BCPA adversary* $\mathcal{A}$ *has a non-negligible advantage* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_\mu\text{-}\mathsf{BCPA}} := \text{Prob}[\mathsf{b}' \leftarrow \mathcal{A}(c_*) : \mathsf{b} = \mathsf{b}'] - \frac{1}{2}$ *, then, a probabilistic, polynomial-time algorithm* $\mathcal{C}$ *can create an environment where it uses* $\mathcal{A}$*'s advantage to solve any given instance of the SXDH problem.*

Proof. 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 $\mathcal{O}_{\mathsf{Corrupt}}$ 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 $\mathrm{P}(0) = a$ and $\mathrm{P}(i) = y_i$ for $i = 1, .., t - p_{\text{lim}} - 1$. $\mathcal{C}$ then sets the key-shares to $(i, \mathsf{sk}_i) \leftarrow y_i, i \in [1, t - p_{\text{lim}} - 1]$ and assigns (i, sk_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 $\mathcal{O}_{\mathsf{Corrupt}}$:

$\mathcal{O}_{\mathsf{H}_1}$: 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.

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

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, $W^{\rho_{\ell_*}} = \hat{e}\,(g, \bar{g})^{abc\rho_{\ell_*}} = \hat{e}\,((B^{\rho_{\ell_*}})^a, \bar{g}^c) = \hat{e}\left(H_1(\ell_*)^{\mathsf{sk}}, \bar{C}\right)$. 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 $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_{\mu}\text{-}\mathsf{BCPA}}$ is bounded as $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{IND}_{\mu}\text{-}\mathsf{BCPA}} \leq e \cdot (\mathcal{Q}_{H_1} + 1) \cdot \mathsf{Adv}^{\mathcal{C}}_{\mathsf{SXDH}}$. □

6.3 Security Analysis of the Scheme

In chapter 4 we set the goal of our scheme to provide convergent security for popular files and semantic security for unpopular files. We claim that the goal was reached by our scheme under the assumptions that IRS and IdP are trusted (and thus not corruptable by an adversary) and there are no more than $n_{\mathcal{A}}$ corrupted users. In this section we first sketch a proof of this claim and then discuss the individual assumptions and what would happen with the security provided by the scheme in case they were violated. We demonstrate the security properties by the "views" of each of the scheme participants and compare the views in our scheme and in other secure deduplication proposals to show the differences.

Note that our work systematically focuses on security only, setting the privacy aspect aside. This is intentional since user identities are handled by IdP and the concrete implementation and deployment of IdP and authentication is a complex problem orthogonal to our main goals. To at least briefly comment on privacy, we note that 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 (this was proven in the previous section; provided that IRS is trusted), 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 [37, 38]. 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. Undoubtedly, this would lead to a more complex IdP that would need to handle "obfuscation" of user identities (*e.g.* for the billing purposes) which falls out of scope of our work. Note that if we relax the trusted requirement of the IRS (such as we do in section 6.5) then in the case when only IRS gets compromised, the privacy is not breached since IRS view consists only of a set of indexes $\{\mathsf{r_i}\}$ corresponding to the owners of the unpopular files, but does not contain the actual user identities $\{\mathsf{U}_i\}$.

6.3.1 Semantic Security of Unpopular Files

Convergent security was formally analyzed by Bellare *et al.* [7] and our scheme implements classic convergent encryption for popular files. Semantic security for unpopular files is assured by the semantically-secure cryptosystem $\mathcal{E}$. What remains to show is that our scheme does not inadvertently break the security level for unpopular files *i.e.* that an adversary cannot break the security of an unpopular file without breaking the security

assumption of trusted IdP, IRS and threshold of corruptable users.

Claim: Adversary $\mathcal{A}$ cannot break semantic security of any unpopular file F (*i.e.* having obtained the ciphertext created by $\mathcal{E}.\mathsf{E}$ he cannot extract any additional information revealing anything about the plaintext F) if he can only corrupt the storage provider S and a set of users $\mathcal{U}$ where $|\mathcal{U}| \leq n_{\mathcal{A}}$.

Proof. $\mathcal{A}$ can obtain any unpopular file record from S (note that 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 $\mathcal{I}$ containing 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 (as proved earlier), 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 $t \geq p_{\text{lim}} + n_{\mathcal{A}}$ 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. □

6.3.2 Analyzing the Consequences of Broken Assumptions

Maintaining semantic security of unpopular files even in presence of corruptable honest but curious storage service provider S and $n_{\mathcal{A}}$ corrupted users is no small feat, but requires very strong assumptions that the adversary does not corrupt IdP, IRS and more than $n_{\mathcal{A}}$ users. To analyze the situation where an adversary could violate these assumptions, we provide the view of all scheme entities regarding popular and unpopular files in Table 6.1.

IdP Corruption / Corrupting an Unbounded Number of Users

We start with the identity provider IdP since it's view is the easiest to analyze. We also demonstrate that IdP corruption and corrupting an unbounded number of users (*i.e.* breaking the "no more than $n_{\mathcal{A}}$ corrupted users" assumption) are equivalent in terms of our security model.

Table 6.1: Scheme Participant Data Views

	Popular File	Unpopular File
IdP	$\emptyset$	$\emptyset$
IRS	$\mathcal{I}_{F_c}$	$\mathcal{I}_{F_c}, ctr, \{(\mathsf{r_i}, \mathsf{ds_i})\}_1^{ctr}, \{\mathcal{I}\}_1^{ctr}$
S	$F_c, \{\mathsf{U}_i\}_1^{p_F}$	$\mathcal{I}_{\text{rnd}}, (c, c_\mu), \mathsf{U}_i$

Regarding the actual data being stored by the storage service, the IdP observes (and thus can provide) no information. The reason why IdP has to be trusted is thus not directly related to the data, but to the user identities. If the adversary $\mathcal{A}$ is allowed to corrupt IdP, he can exploit the user-generating process to spawn an unbounded number of corrupted users and thus decrypt the upper (semantic security) encryption layer of any unpopular file for which he knows the label $\ell = H_1(F_c)$. The decryption is possible since the adversary can generate t valid decryption shares $\left(\mathsf{r_i}, \mathsf{ds_i}\right)$ with the corresponding label ℓ (recall that $\mathsf{ds_i} \leftarrow H_1\left(\ell\right)^{\mathsf{sk_i}}$). Having a set of enough decryption shares, the adversary can invoke $\mathcal{E}_\mu$.Decrypt.

Depending on the IdP implementation, the adversary could also extract the master secret skof the $\mathcal{E}_\mu$ cryptosystem upon IdP corruption (unless the implementation stores skin some inextractable way *e.g.* in a hardware security module). Notice that knowledge of skhas exactly the same consequences as generating an unbounded number of corrupt user identities since the adversary can skip the decryption share combination step of $\mathcal{E}_\mu$.Decrypt

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

and compute $H_1(\ell)^{\mathsf{sk}}$ directly. The rest of the decryption algorithm can run as usual.

Notice that the attack is only possible if the adversary is able to guess (or obtain from some other source) the encryption label $\ell = H_1(F_c)$. Since the label is not stored in S, the adversary cannot choose some record of an unpopular file copy inside S (consisting of index $\mathcal{I}_{\text{rnd}}$ and ciphertext c) and force that particular file to become popular since there is no way how to extract label ℓ from $\mathcal{I}_{\text{rnd}}$ or from c. The adversary cannot decrypt c since ha lacks the sufficient number of decryption shares and cannot generate them without ℓ. The only way for the attacker to force an unpopular file with unknown label ℓ to become popular would thus be to enforce all files in the storage to become popular, which is infeasible.

Also note that apart from the data-related information, a corruptable IdP can be seen as a potential privacy breach since it has to validate (and thus know) the real user identity and knows the real user identity to scheme user identity mapping. This issue is a complex one to address and it falls out of scope of this thesis, yet there are works dedicated to this issue countering it with pseudonymity and anonymous authentication in various scenarios – see *e.g.* works by Camenisch *et al.* [37] or Lysyanskaya *et al.* [38].

IRS Corruption

The index repository service IRS contains a lot of information for each unpopular file F – the deduplication (convergent) index $\mathcal{I}_{F_c}$, it's mapping to the random indexes $\{\mathcal{I}\}_1^{ctr}$ (used to index ciphertexts in S) and the corresponding decryption shares $\{(\mathsf{r_i}, \mathsf{ds_i})\}_1^{ctr}$. While the random indexes do not constitute any leakage (computed via PRF) and the decryption shares are unlinkable *i.e.* label is not extractable from them (see Section 6.2.1) the remaining piece of information is vital – $\mathcal{I}_{F_c} = H_1(F_c)$ is both the label ℓ used in $\mathcal{E}_\mu$.Encrypt and the deduplication index $\mathcal{I}$. Since H_1 is collision-resistant, the attacker can use the index to mount the same attacks as against convergent encryption thus security of all unpopular files is degraded from semantic to convergent.

Differently from IdP corruption, the attacker knows the label $\ell = H_1(F_c)$, but similarly as in the IdP corruption case, he also cannot choose some record of an unpopular file in S and decrypt its upper layer since he is not able to generate the necessary decryption shares. The situation in both cases is the same, but stems from different reasons. Note that in the case of IRS corruption, the attacker does not need the actual data contents stored in S to mount an attack – knowledge of the index is sufficient, since there are no collisions and thus the index always corresponds to the (correctly guessed) plaintext (convergently-encrypted plaintext, respectively). The fact that IRS does neither store nor ever has access to the actual data (nor their size) is vital for the strengthening measures suggested in section 6.5, focusing on the possibility to remove the need of a trusted IRS.

6.4 Security Comparison with Other Secure Deduplication Solutions

As described in chapter 3, different secure deduplication solutions often have different goals and thus design different ways to achieve them. As we have demonstrated in our security analysis, our scheme meets the goals we defined (*i.e.* semantic security for unpopular files, convergent security for popular files, if the defined assumptions hold) and is the only secure deduplication scheme that we know of that *implicitly allows corruption of the storage provider and up to $n_\mathcal{A}$ users without compromising semantic security of unpopular files.* On the other hand, it undeniably also exhibits the single point of failure (and potential leakage) vulnerability by requiring participation of a trusted IRS. Here we compare our scheme security-wise with four other state of the art secure deduplication schemes to demonstrate the differences.

DupLESS [5] uses a key server similar to our IRS but instead of index obfuscation, the key server actually changes the key used for file encryption – instead of encrypting

a file with deterministically derived convergent key k_c and then obfuscating index IND, the key k_c itself is "obfuscated". The undeniable advantage of this approach is that the key server does not store (and never actually sees) the deduplication index nor the key k_c since the user communicates with the key server using an oblivious transfer protocol and the key server only uses its own private secret key to obfuscate k_c. Since DupLESS was designed for target-based deduplication, it inherently counters attacks typical for convergent encryption, though, for the same reason it cannot prevent the honest-but-curious storage provider to check for file equality nor mount attacks based on the knowledge of the deduplication index (*i.e.* file hash).

ClearBox [22] uses gateway G similar to DupLESS key server to achieve server-aided key generation. G can prevent known-hash-based attacks thanks to incorporated PoW but it cannot prevent attacks based on the known (guessed) content – a user may learn whether a file was stored before. Interestingly, ClearBox does not consider this to be a threat in its security model and it even implements an Attest procedure that eventually "leaks" approximately how many users stored each file. This is caused by the different security goals set by the ClearBox authors which prioritize undeniable deduplication estimates over the "learn if file was stored" leakage. A suggested solution of adding high-entropy strings to low-entropy files to counter guess-based attacks is rather tedious and requires the user to somehow identify such low-entropy files. Our scheme offers automatic protection against such attacks (assured by the IRS) if the popularity principle is acceptable by the user (*i.e.* if the user agrees that the property may be lost in case more than $t-1$ other users also upload the same file). One more slight difference between our scheme and DupLESS compared to ClearBox is that G actually has access to the encrypted files (and thus knows also some additional properties such as the file size).

Liu *et al.* [23] present a scheme that does not require the trusted component in form of IRS, G or a key server and instead delegates the trust among the individual users of the system. The server component itself participates only minimally in the actual process of deduplication, most of the "sensitive" computation is done by the users. While the "user-trust-based" approach is definitely interesting, the presented scheme cannot prevent an honest but curious storage provider S to check for file equality and is prone to user collusion attacks. Our scheme prevents the user collusion attacks by introducing a concrete bounded limit for the number of corrupted users.

ClouDedup [21] isolates the storage provider from the deduplication procedure completely, not leaking any information. However, to achieve such a perfect isolation ClouDedup introduces two quite complex trusted components. The security setting of ClouDedup is more suitable for corporate-like environments where the "private network" with users can redirect data through the introduced trusted component(s) to enable their se-

cure storage in some "outside" cloud storage provider. Our scenario is slightly different and our trusted components a bit more "lightweight" whereas ClouDedup is undeniably better suited for actual practical deployment. Security-wise the general approach in both solutions is quite similar – layered encryption, eliminating known-hash-based attacks and prioritizing data confidentiality over other goals.

6.5 Relaxing the Requirement of the Trusted IRS

While our scheme does reach its goal, the requirement of two trusted components, IRS and IdP, is a very strong one. While a trusted IdP is quite common, since practically every system and network has to have some identity provider and management, a trusted IRS is not so common. Moreover, the fact that the IRS knows all the deduplication indexes and decryption shares and can thus bruteforce the deduplication index to gain some information about the stored data content (confirmation of file attacks and learn the remaining information attacks typical for convergent encryption) makes it very powerful and a potential viable target for an attack. Since transferring the ideal trusted third party model to real world is not that easy, we discuss the possibility how to make IRS potentially corruptable without (entirely) sacrificing the "better security assurance" for unpopular data.

In section 3.1.2 we have described the weak point of deduplication – the deduplication index. Knowledge of the index gives the ability to compare data contents with some other data contents and find out if they match, even if they are convergently encrypted. Since our lower layer of encryption is convergent and we need to know which unpopular files have the same content, the deduplication index has to be stored somewhere. Our original security model considers an honest but curious (HBC) storage provider S, up to $n_{\mathcal{A}}$ corrupted users and a trusted IRS. If we modify the model to allow IRS to be honest but curious (same as S), security of unpopular files will be automatically degraded to convergent (see section 6.3).

For deduplication to work in our scheme, the deduplication index has to be known to some of the scheme participants *i.e.* to IdP, IRS, S or users. IdP is out of scope, as it is included only to manage user identities and not to participate in the scheme otherwise. Splitting the information among users owning files with the same index would require notable redesign of the whole scheme and users to stay online most of the time, which we want to avoid. If such a requirement is acceptable, we recommend the work of Liu *et al.* [23], where a secure deduplication model based on this principle is described. That leaves two possibilities – IRS, where the index is stored in the current scheme, and S. Setting S as trusted does not make practical sense – if S were trusted, the proposed

complex deduplication scheme would not be required at all. This creates a seemingly impossible situation – IRS must store the index, yet we wish to make it corruptible.

To move further, we need to analyze the deduplication index weakness in the specific case of our scheme in more detail. For this we use the "confirmation of a file" attack scenario – the attacker who knows, or can guess, content of a file can use the index to find out if the file was already uploaded by someone else. Consulting the IRS view (see Table 6.1) this can be devised by computing the index of the known file and checking that a record in IRS indexed by this index exists. If so, the file was already stored. If there are any associated decryption shares, the attacker can be "near-sure" that i) the file was uploaded by *ctr* users and ii) is unpopular. The attacker can be "near-sure" since the indexing function is collision resistant and thus a chance that one index will be associated with more different files is extremely low. Note that if there are no shares associated with the index but *ctr* is non-zero, the attacker can be sure that the file was also uploaded and is popular (since deduplication already correctly happened).

Thus, even though the attacker that compromises IRS does not have access to the ciphertext (and even if he had access, the ciphertext is encrypted using semantically secure encryption and thus not decryptable unless t valid shares were already collected), our scheme is still prone to a "confirmation of file" attack if IRS is corrupted since IRS provides a "near-sure" confirmation of the file presence. Using this knowledge, we aim at modifying our scheme to achieve a situation where the "near-sure" confirmation will be removed, thus the attacker wouldn't be able to prove his guess and the security model would allow a corruptable IRS. In the rest of this section we analyze two possible relaxations of the trusted IRS requirement – a "weaker" one where the attacker may compromise either S or IRS, but not both, and a "stronger" one, where he can corrupt both. We note upfront that the sketched modified scheme proposals are by far not as simple and efficient as our original scheme, which is why we decided not to implement them in our core scheme proposal.

6.5.1 Adversary Can Corrupt either IRS or S

Let us first consider a security model where the attacker can corrupt up to $n_{\mathcal{A}}$ users and can corrupt either S or IRS, but not both. Thanks to the fact that the attacker cannot corrupt both IRS and S in this new model, we can decide not to store the deduplication index in any one of them, but to split it among them instead. Indeed, this constitutes a potential leakage of information about file contents inside S, which was fully prevented in the original setting, but it allows to lift the strong requirement for incorruptible IRS, replacing it by a weaker "either S or IRS can be corrupted" notion. Undoubtedly, corrupting two separate systems that have different implementation, can be placed in different places and

protected by different mechanisms is a much more complex task than finding a weakness in one target system. Additionally, the scheme modification sketched in this section is very versatile and allows to vary the degree of information leakage obtainable for both S and IRS in this new setting. The typical attacks against deduplication systems, such as the "confirmation of a file" attack scenario thus can be mitigated (up to some predefined degree of acceptable probability).

To modify our scheme to fit the new model, we need to implement the following:

Index splitting To be able to split the deduplication index into parts, one to store inside IRS and one inside S. It is not necessary to split the index into equal halves and it is also possible to split it to more than two parts (even though just two will be shared). This can be configured depending on the size of the index and acceptable leakage in the IRS and S.

Deduplicability feedback function Since the IRS newly stores only a part of the index, the decryption shares indexed with this part can correspond to different files. Thus, there has to be a deduplicability feedback function through which the IRS could ask S whether the set of random indexes likely corresponds to the same file (and thus it makes sense to try deduplication) or not.

Deduplication result Once a file is successfully deduplicated, the S has to notify the IRS, providing the whole $\mathcal{I}_{F_c}$ (since it is unknown to IRS).

Popular file list Since the user cannot use the whole $\mathcal{I}_{F_c}$ when checking if a file is popular (since that would leak $\mathcal{I}_{F_c}$ to IRS, which is not desired) there has to be a list of popular files published (and updated) by the IRS. We denote the list as PLIST.

Since the new approach requires the user to split $\mathcal{I}_{F_c}$ and not to share the whole index with either IRS or S until the user is sure that the file is already popular (otherwise the split of the index to parts wouldn't really make sense), we introduce a new mechanism to the scheme – the popular file list PLIST. The list is maintained and published by the IRS. A new deduplication index of a file is added to the list once IRS invokes the Deduplicate algorithm and obtains a success result for it. The user is responsible to download (or update, if he already downloaded a previous version) the popular file list as a first step when invoking Upload. Since the list of popular files is publicly available in this modified scheme, it is highly recommended to add also a Proof of Ownership (PoW) mechanism to protect against the covert file distribution attack. The PoW should be added to a deduplicable Put request algorithm.

U_i: $k_c \leftarrow \mathcal{E}_c.\mathsf{K}(F)$
$F_c \leftarrow \mathcal{E}_c.\mathsf{E}(k_c, F)$
$\mathcal{I}_{F_c} \leftarrow \mathcal{I}(F_c)$
$\mathsf{U}_i \longrightarrow \mathsf{IRS}$: $\mathsf{PLIST} \leftarrow$ current PLIST
U_i: **if**$(\mathcal{I}_{F_c} \in \mathsf{PLIST})$
$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathsf{Put}(\mathcal{I}_{F_c}, \mathsf{U}_i, F_c)$
U_i: $F \leftarrow (k_c, \mathcal{I}_{F_c})$
else
$\mathcal{I}_{F_{c|1}} \mathcal{I}_{F_{c|2}} \leftarrow \mathcal{I}_{F_c}$
$(\mathsf{r_i}, \mathsf{ds_i}) \leftarrow \mathcal{E}_\mu.\mathsf{DShare}(\mathsf{r_i}, \mathsf{sk_i}, F_c)$
$\mathsf{U}_i \longrightarrow \mathsf{IRS}$: $\mathcal{I}_{\text{ret}} \leftarrow \mathsf{GenSecIdx}(\mathcal{I}_{F_{c|1}}, (\mathsf{r_i}, \mathsf{ds_i}))$
U_i: $k \leftarrow \mathcal{E}.\mathsf{K}()$;
$c \leftarrow \mathcal{E}.\mathsf{E}(k, F_c)$
$c_\mu \leftarrow \mathcal{E}_\mu.\mathsf{Encrypt}(\mathsf{pk}, F_c, k)$
$F' \leftarrow (c, c_\mu, \mathcal{I}_{F_{c|2}})$
$\mathsf{U}_i \longrightarrow \mathsf{S}$: $\mathsf{Put}(\mathcal{I}_{\text{ret}}, \mathsf{U}_i, F')$
U_i: $F \leftarrow (k, \mathcal{I}_{\text{ret}}, k_c, \mathcal{I}_{F_c})$

Figure 6.1: Modified $\mathsf{Upload}(F, \mathsf{U}_i)$ algorithm. Popular file upload part is highlighted in green (lighter color), unpopular file upload part in blue (darker color).

To implement the index-splitting, we modify our scheme as follows: The convergent index $\mathcal{I}_{F_c}$ that is computed during the Upload algorithm is split to two parts $\mathcal{I}_{F_c} - > \mathcal{I}_{F_{c|1}}, \mathcal{I}_{F_{c|2}}$. Only the first part $\mathcal{I}_{F_{c|1}}$ is being sent to the IRS as part of the GenSecIdx algorithm. The second part $\mathcal{I}_{F_{c|2}}$ is stored locally and only in case that actual data upload is required (file not yet popular) the second part $\mathcal{I}_{F_{c|2}}$ is attached to the data that is being uploaded to S. We present the modified version of the Upload algorithm in Figure 6.1. We define a new parameter dedidx in the S record (i.e. for an S record indexed by idx we define $\mathsf{DB_S}[idx].\mathsf{dedidx}$) to store the $\mathcal{I}_{F_{c|2}}$. Note that if there is a requirement on higher level of "uncertainty" for the potential attacker, the index could be split into three parts and the third parts kept local, not shared with IRS nor with S. Note that this additional strengthening measure comes with a cost of (potentially many) more deduplicability feedback requests.

When a GenSecIdx request that would cause invocation of Deduplicate in the original scheme arrives at IRS, IRS invokes the deduplicability feedback function Deduplicable instead. Deduplicable is defined in Figure 6.2. The purpose of Deduplicable is to find out if all the "random indexes" indexed by the same $\mathcal{I}_{F_{c|1}}$ inside IRS correspond to the same file or not. S can answer this question since it stores $\mathcal{I}_{F_{c|2}}$ in the records indexed by the "random indexes" provided as parameters in Deduplicable – if there are at least t same $\mathcal{I}_{F_{c|2}}$ in the checked records, the corresponding records do belong to the same file and thus can be deduplicated. In such a case S returns *True* and a set of indexes having the same $\mathcal{I}_{F_{c|2}}$.

S: $\mathsf{Darr} \leftarrow [0, 0, .., 0]$; $\mathsf{Rarr} \leftarrow [\emptyset, \emptyset, .., \emptyset]$; $i \leftarrow 1$
foreach($\mathcal{I} \in \mathsf{idxes}$)
if($\exists j : \mathsf{DB_S}[\mathcal{I}].\mathsf{dedidx} = \mathsf{Darr}[j]$))
$\mathsf{Rarr}[j] \leftarrow \mathsf{Rarr}[j] \cup \mathcal{I}$
else
$\mathsf{Darr}[i] = \mathsf{DB_S}[\mathcal{I}].\mathsf{dedidx}$; $\mathsf{Rarr}[i] \leftarrow \mathsf{Rarr}[i] \cup \mathcal{I}$; $i \leftarrow i + 1$
if($\exists j : |\mathsf{Rarr}[j]| \geq t$))
return ($True$; $\mathsf{Rarr}[j]$)
else
return ($False$; $\emptyset$)

Figure 6.2: The Deduplicable($idxes$) algorithm. For a set of indexes of records inside S returns if at least t of the records do contain the same $\mathcal{I}_{F_{c|2}}$ value. If yes, it returns a list of such indexes.

If there is no set of records of size at least t that would share the same $\mathcal{I}_{F_{c|2}}$, Deduplicable returns *False* and IRS would need to re-try later, when there are more candidates in its respective record. The GenSecIdx function has to be modified to account for these re-tries and a new parameter needs to be introduced to specify when the re-try should occur. A reasonable option might be to retry every time the counter reaches a multiplied value of threshold t (e.g. $2t$, $3t$ etc.) – in this case we don't even need to specify a new parameter and can modify the GenSecIdx as shown in Figure 6.3.

Providing a deduplication result in the form of $\mathcal{I}_{F_c}$ from S to IRS after a successful deduplication (such that IRS can update and publish the popular file list) is the last required modification. This can be easily achieved by adding **return** $\mathcal{I}_{F_c}$ at the very end of the Deduplicate algorithm from the original scheme.

Analysing the proposed modified scheme, we can see that the actual security properties

IRS: $\mathcal{I}_{\text{rnd}} \leftarrow \mathsf{PRF}(\sigma, \mathsf{U}_i || \mathcal{I}_{F_{c|1}})$
if ($\mathcal{I}_{\text{rnd}} \notin \mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{idxes}$)
increment $\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{ctr}$
add $\mathcal{I}_{\text{rnd}}$ to $\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{idxes}$
add $(\mathsf{r_i}, \mathsf{ds_i})$ to $\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{dshares}$
if ($\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{ctr} = mt$; $m \in \mathbb{N}$)
IRS $\longrightarrow$ S: ret $\leftarrow$ Deduplicable($\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{idxes}$)
if(ret = ($True$; retidxes))
IRS $\longrightarrow$ S: $\mathcal{I}_{F_c} \leftarrow$ Deduplicate($\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{retidxes}$, $\mathsf{DB_{IRS}}\left[\mathcal{I}_{F_{c|1}}\right].\mathsf{dshares}$)
IRS: $\mathsf{PLIST} = \mathsf{PLIST} \cup \mathcal{I}_{F_c}$

Figure 6.3: Modified GenSecIdx($\mathcal{I}_{F_{c|1}}, (\mathsf{r_i}, \mathsf{ds_i})$) algorithm. There is no green part related to popular file since popular files are directly uploaded to S once found in PLIST. The red part related to deduplication is extended to account for the new Deduplicable algorithm.

highly depend on the specific setup and dataset. Considering an index size $|\mathcal{I}_{F_c}| = \kappa$, if we decide to split it such that $|\mathcal{I}_{F_{c|1}}| = \kappa - 1$ and $|\mathcal{I}_{F_{c|2}}| = 1$ then the "uncertainty" of the attacker that corrupted IRS when trying to guess if file F was uploaded is minimal since κ is chosen high enough to allow for collision-resistant $\mathcal{I}$, thus possible "hit" still means the attacker can be near-sure the file was uploaded (the probability that someone uploaded file F_2 whose hash would match that of the guessed file F but for the last bit is very low). If we decide to split the index in equal halves (*i.e.* $|\mathcal{I}_{F_{c|1}}| = |\mathcal{I}_{F_{c|2}}| = \kappa/2$), then the "uncertainty" for an attacker that corrupted IRS is substantially higher – considering a guess of file F and computing index $\mathcal{I}_{F_c}$, the attacker can check if there is any record for $\mathcal{I}_{F_{c|1}}$. However, if there is, it can either correspond to F or to any other uploaded file whose index has the same $\mathcal{I}_{F_{c|1}}$ (*i.e.* there are $2^{\kappa/2}$ other options). Note that the "uncertainty" of the attacker that corrupted S is the same in this respect since he can theoretically check the values of all records, comparing the $\mathcal{I}_{F_{c|2}}$. If this level of uncertainty is not acceptable, it is possible to split the index to three parts and increase size of $\mathcal{I}_{F_{c|3}}$ (thus decrease size of $|\mathcal{I}_{F_{c|1}}|$ and $|\mathcal{I}_{F_{c|2}}|$) to a required level. Note though that such behavior inherently decreases efficiency of the scheme as the number of Deduplicable and failed Deduplicate requests would notably increase.

6.5.2 Adversary Can Corrupt both IRS and S

The previously described modification where the adversary can corrupt either IRS or S for the cost of performance decrease does not completely solve the limitation posed by the requirement of a trusted third party since either IRS or S has to be trusted (*i.e.* not corrupted by the same attacker). Here we try to eliminate the trusted party completely (apart from IdP, as discussed earlier). In our new security model we postulate that the attacker can corrupt up to $n_{\mathcal{A}}$ users and both S and IRS are honest but curious. Note that this new security model allows the attacker to corrupt both IRS and S thus it is not possible to split the index between them as the attacker could simply "reassemble" it using the information he gets from IRS and S.

Considering the situation, the only possible approach that we identified is to sacrifice the part of the index shared with S. The scheme would look exactly as in the modification proposed in Section 6.5.1 but will not share $\mathcal{I}_{F_{c|2}}$ with S (*i.e.* there will be no $\mathcal{I}_{F_{c|2}}$ inside the S records, the S records are exactly the same as in the unmodified scheme presented in Section 5). This also influences Deduplicable, which cannot be implemented. Thus, instead of invoking Deduplicable as defined in the modification in Section 6.5.1, we would always need to invoke Deduplicate and S would need to try all the possibilities.

Note that if there are more than t deduplication shares during deduplication computation, the only way to successfully deduplicate is to try (in the worst case all) possible

combinations. Having $t + x$ decryption shares in the Deduplicate request, this means $\binom{t+x}{t}$ combinations of decryption shares where for each combination the shares have to be combined ($\mathsf{H}_1(\ell)^{\mathsf{sk}}$) and file contents have to be decrypted and compared. It is easy to see that for higher values of x this quickly becomes computationally infeasible, thus the "uncertainty" of the attacker has to be kept relatively small (to make that Deduplicate will have high probability to succeed for reasonable values of x).

Due to the described complexity, this modification is quite impractical and serves more as a theoretical concept than a useful proposal. Finding a better solution that would allow to have both IRS and S corruptible is an open problem.

In our proposed modifications we have noted "performance degradation" due to various factors. To get a better idea about how to compute and measure performance of the scheme in general (and therefore also how big the described performance degradations may be) kindly refer to chapter 7.

Chapter 7

Performance Evaluation

Our scheme was designed to provide more fine-grained trade-off between security and space efficiency compared to the "classic" deduplication scheme exploiting convergent encryption [3]. Specifically, deduplication efficiency is decreased for the benefit of increased security of unpopular files. This section presents both theoretical equations and practical measurements to demonstrate scheme performance and efficiency both in terms of space reduction and in computation and communication cost. We compare the results to those of classic deduplication and to other secure data deduplication solutions.

First, we modify the deduplication ratio (DR) definition, used for classic deduplication efficiency evaluation, to formulate a space reduction ratio (SRR) that can be used to easily compare the efficiency of our scheme to that of classic deduplication, using the dataset properties only, without the need to apply deduplication to the dataset. Next, we present evaluation of the SRR efficiency of our scheme on artificial and real datasets, to demonstrate which factors influence it, and how. Second, we focus on resources required by the scheme in terms of computation and communication split between the individual algorithms and their phases. Additionally, we comment on the expected user-perceived delay, comparing to storage services without deduplication or with classic convergently secured deduplication.

7.1 Storage Space Reduction Ratio

A classic deduplication scheme uses a simple metric called *deduplication (or duplicity) ratio* (DR) to evaluate the space-saving efficiency of deduplication applied to a concrete dataset. Deduplication ratio is defined as "size of dataset before deduplication" divided by "size of dataset after deduplication" or, in a simpler way, as DR = *bytes in*/*bytes out* [2]. Technically, deduplication ratio is applicable to every deduplication scheme (ours included), however it is necessary to first deduplicate the whole dataset. To prevent this necessity

of the actual dataset deduplication process (which can be quite resource-intensive), we present the *Space Reduction Ratio* (SRR) that can be used to compute deduplication efficiency based solely on known properties of the individual dataset files. Specifically, for each unique file F in the dataset we only need to know its size |F| and its popularity p_F (*i.e.* in how many copies it is present in the dataset).

Having a dataset described using unique files, their sizes and popularities, a classic deduplication scheme removes all excessive copies of a unique file, keeping only one. Therefore we denote classic deduplication as *perfect deduplication* for which it holds that a per-file space reduction ratio for file F is equal to its popularity p_F (since the non-deduped size $p_F \times |F|$ is reduced to $|F|$ by deduplication). 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 4.3, $t \geq p_{\text{lim}} + n_{\mathcal{A}}$). The concrete deduplication efficiency of our scheme thus depends directly on two factors – the threshold t (tunable parameter of the scheme) and the popularity distribution of files in the dataset (determined by the dataset, non-settable).

To define formally, 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 $\mathcal{F}_p = \{F_i \ | p_{F_i} \geq t\}$ and a set of unpopular files $\mathcal{F}_u = \{F_i \ | p_{F_i} < t\}$. The space reduction ratio SRR is then computed as:

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

For simpler comparison (to directly see how much space was saved, percentually) we define the space reduction percentage SRP as

$$\mathsf{SRP} = (1 - 1/\mathsf{SRR}) \times 100$$

Note that $\mathsf{SRR} = 1$ ($\mathsf{SRP} = 0$) means no deduplication occurs (zero efficiency).

Note that same as DR, SRR is computed over the actual file contents only, not counting associated metadata (such as when the file was accessed, modified etc.). For deduplication to be efficient, it is expected that metadata are marginal in size compared to actual file contents. Since our scheme generates some additional metadata that are not being counted by the SRR, we also address the metadata overhead later in this section and show that it is likewise marginal to the actual file contents size.

7.2 Analysis of Space Reduction Efficiency

As described, the space reduction efficiency of our scheme is directly dependent on two factors – the threshold t and the popularity distribution of files in the dataset. We first present an analysis of space reduction efficiency of our scheme using simple artificial distributions to demonstrate the influence of choice of threshold t and then perform the analysis using two real-world datasets, to show the influence in practice.

7.2.1 Artificial Datasets

To demonstrate the influence of the factors on storage space efficiency of the scheme without concrete data we use artificial datasets with files of uniform size $|F| = 1$. This simplifies the SRR equation to

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

First we use a very basic example where popularity is constant *i.e.* for every file F in the dataset, the popularity $p_F = c$, where c is a natural number greater than zero (we deliberately prevent a dataset without duplicates). With constant popularity, the SRR can be easily computed, based on the value of t as:

1. $t \leq c$ perfect deduplication (maximum efficiency) is achieved since all files are popular *i.e.* $\mathcal{F}_u = \emptyset$ thus

$$\mathsf{SRR} = \sum_{i=1}^{N}(p_{F_i})/(\sum_{F\in\mathcal{F}_p} 1) = p_F$$

2. $t > c$ no deduplication (zero efficiency) is achieved since all files are unpopular *i.e.* $\mathcal{F}_p = \emptyset$ thus

$$\mathsf{SRR} = \sum_{i=1}^{N}(p_{F_i})/(0 + \sum_{F\in\mathcal{F}_u}(p_F)) = 1$$

The constant-popularity example clearly shows the basic limits but does not demonstrate the gradual changes of SRR. To show these we use a uniform distribution.

Using a discrete uniform distribution as the dataset popularity distribution nicely demonstrates the influence of step-by-step increasing t on the SRR. Considering a discrete uniform distribution with lower bound a and upper bound b described with a probability mass function as $f(x) = 1/n$ for $x \in \langle a, b\rangle$ and $n = b - a + 1$ (i.e. having a set of discrete values $a, a+1, .., b-1, b$ that are equally likely to be observed), the efficiency of the

proposed scheme based on the value of t can be expressed as:

1. $t \leq a$, perfect deduplication (maximum efficiency) is achieved since all files are popular *i.e.* $\mathcal{F}_u = \emptyset$ and thus

$$\mathsf{SRR} = \sum_{i=1}^{N}(p_{F_i})/(\sum_{F\in\mathcal{F}_p} 1) = p_F$$

2. $a < t \leq b$, less efficient than perfect deduplication is achieved since files with popularity lower than t are popular and those with popularity higher or equal to t are not and thus the general equation must be used

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

3. $t > b$, no deduplication (zero efficiency) is achieved since all files are unpopular *i.e.* $\mathcal{F}_p = \emptyset$ and thus

$$\mathsf{SRR} = \sum_{i=1}^{N}(p_{F_i})/(0 + \sum_{F\in\mathcal{F}_u}(p_F)) = 1$$

For better illustration how the SRR changes let us use a simple example of a storage containing 5 base files with popularity distribution $\mathsf{Unif}(2,6)$ so 20 files in total. SRR of perfect deduplication in this example is 4, SRR and SRP values of our scheme for different values of t are listed in Tab. 7.1 along with number of files remaining in the dataset after deduplication. Note that the number of files remaining in the dataset after deduplication does not increase linearly with increasing t, but grows faster with higher t. Based on this observation, using our scheme with a dataset containing a few files with very high popularity ($p_F >> t$) and many files with very low popularity ($p_F << t$) could still have reasonable deduplication efficiency.

Table 7.1: Scheme Efficiency for Discrete Uniform Popularity Distribution, $\mathsf{Unif}(2,6)$, 20 Files

t	SRR	SRP	files after dedup.
2	4.00 (4:1)	75	5
3	3.34 (10:3)	70	6
4	2.50 (5:2)	60	8
5	1.82 (20:11)	45	11
6	1.34 (4:3)	25	15
7	1.00 (1:1)	0	20

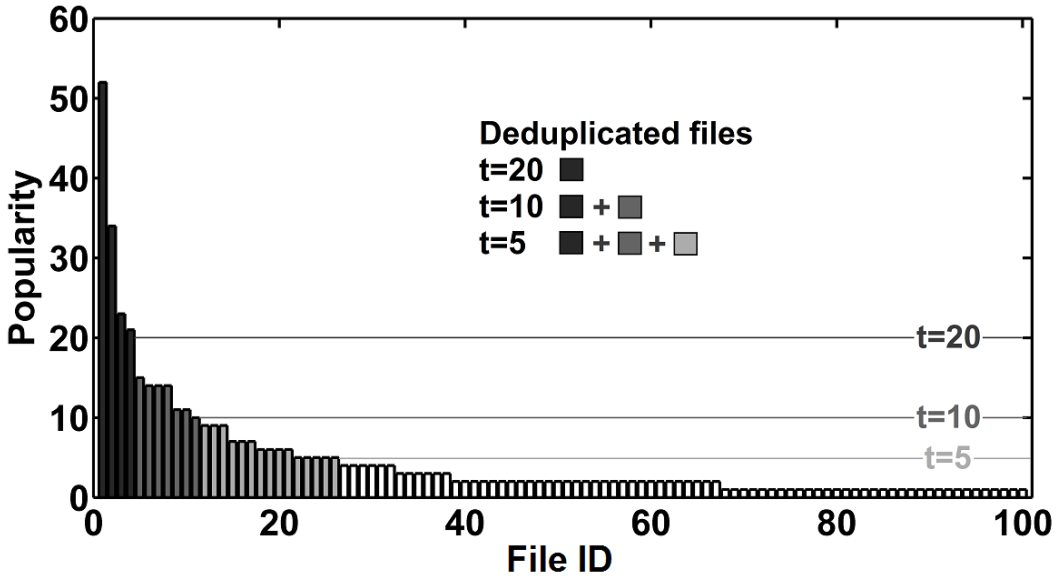

Figure 7.1: Graphical example: Deduplication over dataset with Pareto popularity distribution. Values for t=5, t=10 and t=20 are emphasized. Values for larger data sets and t values are in Table 7.3.

While it is generally impossible to model popularity distribution with a clean simple function-based distribution, analyses of different empirical data show that power-law distributions appear in most of the human-generated and human-managed data from different areas [9]. Based on these observations, we used Matlab to analyze the efficiency of the proposed scheme for a power-law popularity distribution. We used the generalized Pareto distribution with all parameters (*i.e.* shape, scale and threshold) equal to 1 as an illustrative example. First we generated a random vector $\vec{x}$ of length 100, sampling the chosen distribution. The impact of various choices of t in one generated example is illustrated in Fig. 7.1. Note that the concrete numbers will differ per every new measurement since the filesize distribution is obtained by finite sampling from an infinite distribution, with an infinite expected value. To smoothen the results, we repeated the experiment 100 times and computed the arithmetic averages that are presented in Tab 7.2. It is important to take these results as basis for general implications about their relation and dependency instead of considering them to be "hard numbers". To demonstrate how the situation scales up from the very small dataset, we varied the vector length and t values, average results over 100 samples are available in Tab. 7.3.

The above illustrative examples demonstrate how threshold t and the file popularity distribution in the dataset influence the resulting efficiency of the scheme (not very good in either case). Indeed, if the datasets really did have constant popularity per file, or uniform or Pareto distributions, our scheme would not be a very good fit for them. Note that we specifically stressed this in the scheme overview section 4, stating that *"outsourced*

Table 7.2: Average SRR for Generalized Pareto Popularity Distribution, 100 Experiments, Dataset with 100 Files

t	Our Scheme	Perfect Deduplication
5	5.9	9.9
10	4.3	
20	3.4	

dataset contains few instances of some data items and many instances of others". Neither of the classic distributions used so far satisfy this requirement, yet they were useful to demonstrate the interdependence of factors, specifically the threshold t, file popularity p_F and its distribution, all influencing efficiency of our our scheme. In the rest of this section we focus our analysis on real-world examples and discuss scheme efficiency for them.

7.2.2 Real Datasets

To analyse 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 [39], representing an example of user data backup from multiple users, and the *UPC dataset*, similar to the one used by Liu *et al.* [23], consisting of data provided by the Ubuntu Popularity Contest [40] (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

Table 7.3: Average SRR for Generalized Pareto Popularity Distribution, 100 Samples

	t	Number of files	
		10 000	1 000 000
Our Scheme	20	6.24	5.52
	50	4.78	4.24
	100	4.08	3.6
Perfect ded.		18.6	16.46

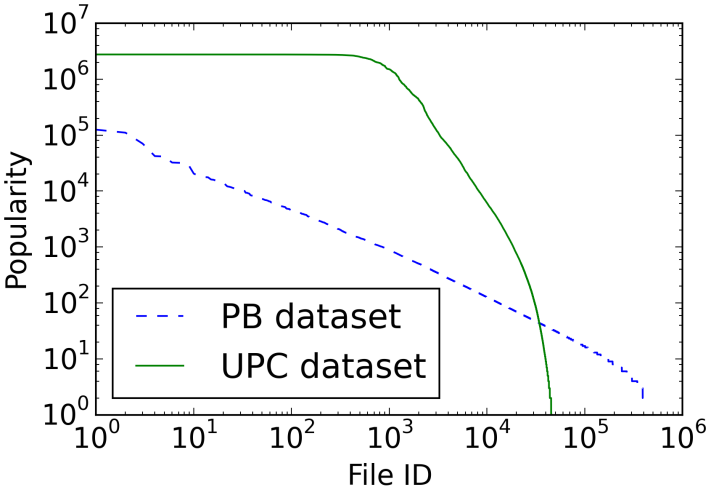

Figure 7.2: File popularity distributions in the evaluated datasets, logarithmic scale.

and "leechers" *i.e.* peers having only part of the file at the moment, but intending to get the whole file in near future. File 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 petabytes.

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 provided by the Ubuntu Popularity Contest[40] 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 ranging from 1 to 2 755 245. There are 3 641 060 666 files in total in the dataset, with the total size of 2,282 petabytes.

Popularity distributions of both datasets are shown in Figure 7.2. To compare the efficiency of our scheme to perfect deduplication, we provide SRP comparison for both datasets in Figure 7.3.

As the SRPs demonstrate, our scheme offers very good reduction for the *UPC dataset*, even for quite high values of threshold t (99,68% 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% 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 petabytes, the *PB dataset* contains only two files with popularity larger than 100 000 whereas the *UPC dataset* contains 3267 such files. 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 obser-

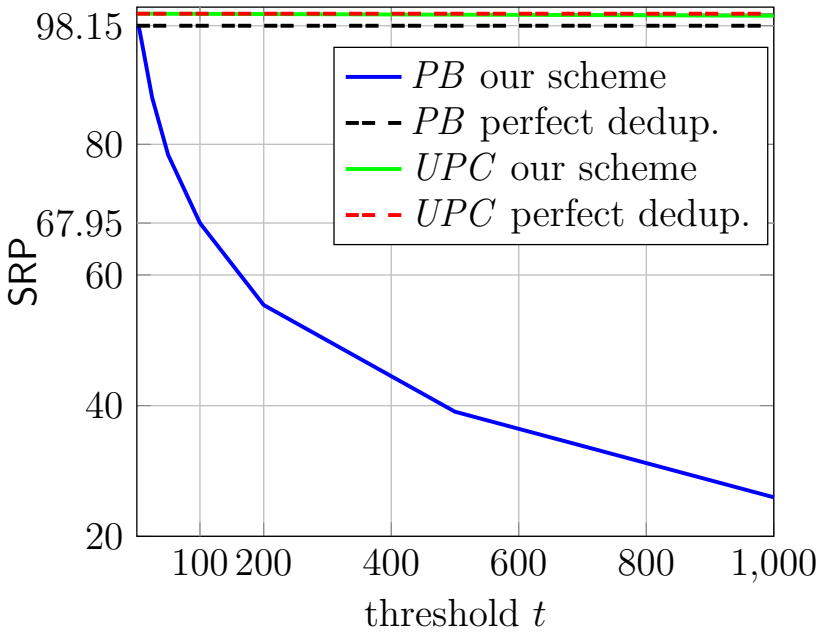

Figure 7.3: Space reduction percentage (SRP) comparison of our scheme and perfect deduplication schemes for the *PB dataset* and the *UPC dataset*.

vations regarding efficiency of our scheme:

1. 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;

2. 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.* SRP = 67,95 for $t = 100$ in the *PB dataset*);

3. 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.

These observations fit well with the original goal of targeting *outsourced datasets containing few instances of some data items and many instances of others*.

7.3 Metadata Overhead Analysis

As described in the SRR definition (Section 7.1), this metric works over file contents only, ignoring metadata as insignificant and marginal in size. To make sure our scheme does not introduce any new metdata of significant size, we do a metadata size analysis,

	Popular File	Unpopular File
user's local storage	$\|\text{index}\| + \|\text{key}\|$	$2 \times (\|\text{index}\| + \|\text{key}\|)$
IRS storage	$\|\text{index}\| + \|\text{ctr}\|$	$(p_F + 1) \times \|\text{index}\| + p_F \times \|\text{dshare}\| + \|\text{ctr}\|$

Table 7.4: General metadata overhead analysis.

providing equations to compute metadata size and evaluate its respective concrete sizes for the real-world *UPC* and *PB* datasets.

In our scheme, metadata are stored in the users local storage and in the IRS storage only. Table 7.4 summarizes the equations to compute a per-file metadata overhead for both users local storage and for IRS storage. Note that we do not consider the metadata stored in the storage providers space S (specifically information about owners and deduplication index required by our scheme) since these have to be an integral part of the storage provider file metadata anyway, independent on whether or not a deduplication scheme is used.

To evaluate the metadata overhead concretely we can use the datasets from Section 7.2 and set $\lambda = 128$ (thus |key| is 8 bytes), index size |index| to 32 bytes, popularity counter size |ctr| to 2 bytes and |dshare| size to 64 bytes (all settings that we actually use for practical measurements analysis later). Considering the highest measured value $t = 1\,000$ to obtain the highest unpopular to popular file ratio (and thus the highet metadata overhead), we computed the metadata overheads listed in Table 7.5.

Note that metadata overhead is independent on the actual file size, though for very small files the overhead could still be seen as significant. From the user's perspective, for deduplication to make sense the file must be larger than 80 bytes (size of metadata needed to be stored locally per unpopular files, not affected by the value of t). From the IRS perspective, the metadata size is highly influenced by the value of t – unpopular files with the highest popularity take up the most space having to store the index mappings and decryption shares. For the considered highest value of $t = 1000$ we can take a worst-case example file with popularity $p_F = 999$ causing corresponding IRS metadata to be almost 96 kB. Thankfully, due to the nature of dataset popularity distribution and the fact that also huge part of the metadata-taken space is reclaimed when the file gets popular, the

Table 7.5: Metadata overhead analysis for the *PB* and *UPC* datasets (in MB).

	PB dataset	*UPC dataset*
user's local storage	25,86	2,94
IRS storage	803,12	503,71

IRS metadata also remains insignificant in case of the *PB* and *DPC* datasets 7.5.

7.4 Computation and Communication Cost Analysis

In this section we analyse the computation and communication cost of our scheme, by dissecting scheme algorithms to individual components and analyzing their cost based on parameter variables. For practical measurements, we implement a prototype of our scheme and measure the respective components. Knowledge from this section can be used to evaluate computation and communication cost of our scheme for any dataset. Finally we provide comparative measurements of our prototype to implementations of other secure deduplication solutions.

7.4.1 Analysis Setup

To measure consumption of computational and communication resources in practice and provide comparison to other schemes, we implement a prototype of our scheme consisting of a client program that performs file upload and download operations, a server IdP program that sets up the system parameters and user share generation, a helper server deduplication application and an IRS that creates secure connection (TLS) with the user and generates secure indexes. For comparison with other schemes, we use the publicly available prototype of DupLESS [5], a prototype kindly provided by Liu *et al.* [23] and our prototype implementation of ClearBox [22] (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 the *Crypto* and *hashlib* Python libraries for symmetric cryptography operations and hashing, and a wrapper for the C-implemented PBC library [34] for pairing-based cryptography operations. 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 implementations were tested on an Intel Xeon E3-1220 machine with 4 CPU cores 3.1 GHz, and 16GB 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 [34] 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.* [41] 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 $t = 1\,000$. We use SHA-256 as the indexing function and also whenever hash functions are needed.

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.* [22].

7.4.2 Network Communication

Network communication consists of relations between the scheme participants. We split and sort the individual data transfers by scheme algorithms in Table 7.6. Note that GetIdx is part of each Upload (*i.e.* Upload.Unpopular and Upload.Popular).

Considering that scheme deployment only makes sense in cases where actual data contents are much larger than the corresponding metadata (see Section 7.3), we observe that most of the bandwidth is consumed by transfer of the actual encrypted file contents *i.e.* the unpopular file upload and (both popular and unpopular) file download operations between user and S. Compared to $|file|$, all other sizes are marginal.

Note that while the interaction between user and IRS consists of very small messages, it occurs quite frequently and, moreover, it must be secure (at least in the "user to IRS" direction) for the scheme security properties to hold. Implementation-wise this could impose additional communication cost (*e.g.* the TLS handshake) and we recommend that the client application should create batch requests or use the same secure connection for more requests to reduce this unaccounted-for communication cost.

7.4.3 Computational Resources

Consumption of computational resources is split among the scheme participants as follows:

IdP performs computation during scheme initialization and upon new user credentials generation

IRS computes a pseudorandom function (PRF) per each first "unique" request (unique combination of user identity and deduplication index; the index is the input to the PRF)

S uses most computational resources during the actual file deduplication process

Table 7.6: Data Transfers per Scheme Algorithm

Algorithm	From	To	Payload Size
GetIdx (Upload)	user	IRS	$\|index\| + \|dshare\|$
	IRS	user	$\|index\|$
Upload.Unpopular	user	S	$\|index\| + \|file\|$
Upload.Popular	user	S	$\|index\|$
Download	user	S	$\|index\|$
	S	user	$\|boolean\| + \|file\|$
Delete	user	S	$\|index\|$
	S	user	$\|index\|$
	user	IRS	$\|index\| + \|r_i\|$
	S	IRS	$\|index\|$ or 0
Deduplicate	IRS	S	$t \times (\|dshare\| + \|index\|)$

user U_i spends most computational resources during encryption and decryption of data

All participants are required also to do some database lookups but these are implementation-specific and should be quite fast and simple, thus we do not include them in our analysis.

Since the PRF computation is always performed over a short hash only, the processing cost is negligible. Therefore, we focus on the more interesting Init, Upload, Download and Deduplicate algorithms in more detail.

Init

We evaluate the time required to initialize the scheme using the $\mathcal{E}_\mu$.Setup implementation consisting of two logically independent processes – "scheme parameters generation and initialization" and "user share generation".

The first process includes the generation of $\{\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, g, \bar{g}, \bar{g}_{pub}\}$ and of the secret sk. Using the PBC library for pairing implementation, most of the time consumed by $\mathcal{E}_\mu$.Setup is taken by the process of pairing parameter generation (*i.e.* finding suitable groups and pairing based on the value of the security parameter λ). The results for varied values of the security parameter λ are available in Table 7.7, value of threshold t plays no role in this phase. Note that the scheme initialization cost is very low compared to the parameters generation cost.

The second process implements secret user-share generation in the most straightforward way – master secret sk is the zeroth coefficient in a polynomial of order t, user share is generated by polynomial evaluation using the Horner scheme. The number of users is practically unlimited, a new secret share can be generated anytime by evaluating the polynomial at the next point (if the last shared secret was evaluated at n, the next one would be evaluated at $n+1$ etc.). Depending on the t value, the generation process takes

from 3 ns for $t = 50$ to 1.04 ms for $t = 1000$. These values are negligible compared to other measured processes, since share generation occurs only once per new user.

Undoubtedly, the processing time of almost 30 seconds that the $\mathcal{E}_\mu.\mathsf{Setup}$ operation has when $1^\lambda = 2048$ would not be acceptable if the operation was to occur frequently during scheme operation. Thankfully, the $\mathcal{E}_\mu.\mathsf{Setup}$ operation is performed by the IdP only once, at the moment of the initial system deployment, and is therefore not significant for regular scheme runtime.

File Upload and Download

We analyze the upload operation first. We split the cost of the operation into smaller isolated components that we analyze separately, and then compose into the total operation cost.

$\mathsf{Upload}(F, \mathsf{U}_i)$ (see Figure 5.4) of an unpopular file can be decomposed into the following operations:

UP1 Convergent encryption and tag generation

UP2 Decryption share generation

UP3 Secure index obtaining

UP4 Symmetric encryption of the convergent ciphertext

UP5 Threshold encryption of the symmetric key

UP6 Data transfer

For a popular file upload, **UP4** and **UP5** are missing and **UP6** transfers only an index instead of file contents. **UP1** is common for all deduplication schemes exploiting convergent encryption and its cost is filesize dependent. **UP2**, **UP3** and **UP5** represent the cost of operations present only in our scheme and are filesize independent. **UP4** represents the cost of operation present only of our scheme and is filesize dependent. **UP6** is both filesize and bandwidth dependent and is present in all deduplication schemes.

Table 7.7: Scheme Parameters Generation and Initialization (in seconds)

λ	Parameter Generation	Scheme Initialization
512	0.45	0.023
1024	4.25	0.023
2048	28.40	0.023

To evaluate the cost of operations independently of the filesize, we used a 1 KB file and repeated the upload process 100 times (without **UP6**). Table 7.8 lists the results. Since the value of **UP3** highly depends on the distance and link quality between IRS and the user (the computation takes 4 ms only), we present three values – the first value corresponds to IRS and user in the same country (CZ), direct distance appx. 200 km, the second corresponds to the same continent (CZ-IT), appx. 1 000 km, and the third to different continents (CZ-JPN), appx. 9 100 km. Note that **UP3** can be split further into subcomponents **UP3.1** secure channel establishment, **UP3.2** over-the channel data transmission and **UP3.3** secure index generation. To minimize cost, we recommended to do **UP3.1** only once for batch uploads and re-use the established secure channel.

To evaluate the cost of operations **UP1** and **UP4** (dependent on the filesize) we split the operations further into subcomponents. $\mathbf{UP1}(|F|) = \{\text{SHA256}(|F|)+\text{AES128.E}(|F|)+\text{SHA256}(|F|)\}$ and $\mathbf{UP4}(|F|) = \{\text{AES128.K}+\text{AES128.E}(F)\}$ (AES128.K is just a random number generation). Since both hashing and symmetric encryption should scale almost linearly with the filesize we have used two test files of size 1 MB and 64 MB and repeated the upload procedure for each of them 100 times (without **UP6**) to compute the average throughput. The resulting approximate throughput for **UP1** is 100 MBps and for **UP4** 500 MBps.

Since **UP6** represents the actual data upload, we can perceive **UP1** to **UP5** as an unpopular file upload overhead incurred by our scheme compared to a plaintext remote storage scheme without any deduplication and encryption. The overhead is composed of constant cost $\mathcal{T}_{\text{const}} = \mathcal{T}(\mathbf{UP2}) + \mathcal{T}(\mathbf{UP3}) + \mathcal{T}(\mathbf{UP5})$ and filesize-dependent relative cost of $\mathcal{T}_{\text{rel}}(|F|) = \mathcal{T}(\mathbf{UP1}(|F|)) + \mathcal{T}(\mathbf{UP4}(|F|))$. From the measurements, it is clear that the constant cost would be the major overhead for small files and with the increasing filesize, the relative cost would become predominant. To obtain concrete numbers we adopted the approach suggested by Bellare *et al.* [5] and generated a set of random content files of size 2^{2i} kB for $i \in \{0, 1, .., 8\}$ (*i.e.* from 1 kB to 64 MB) and uploaded them using only the Dropbox API (*i.e.* plain non-encrypted upload, only **UP6**) and then using our scheme (*i.e.* using **UP1** to **UP6** for unpopular file and **UP1** to **UP3** + **UP6** [index

	Mean	Standard Deviation
$\mathcal{T}(\mathbf{UP2})$	5.14	0.01
$\mathcal{T}(\mathbf{UP3})$	26.3; 112.5; 1 090	0.93; 2.18; 17.45
$\mathcal{T}(\mathbf{UP5})$	36.48	2.55

Table 7.8: Cost of operations independent of the filesize (in milliseconds). The three $\mathcal{T}(\mathbf{UP3})$ values correspond to different geographical settings (same country; same continent; different continents).

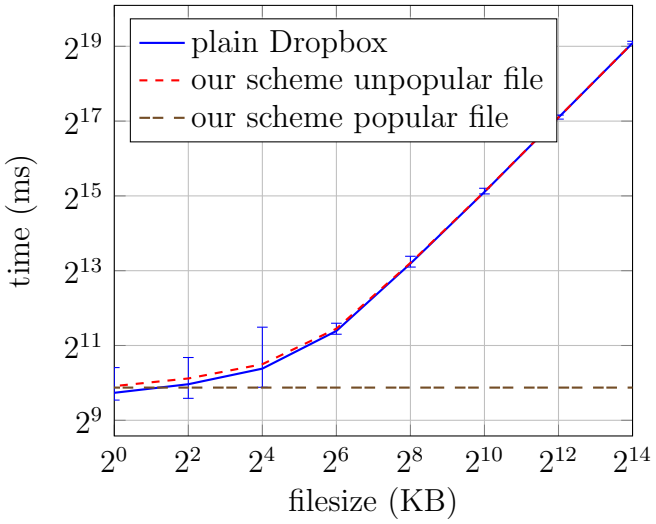

Figure 7.4: Upload duration comparison for plain Dropbox and Dropbox with our scheme.

only] for popular file; using $\mathcal{T}(\mathbf{UP3}) = 112.5$ ms). We repeated the experiment 10x to avoid single-measurement errors, the results are available in Figure 7.4.

While Figure 7.4 demonstrates that the overhead incurred by our scheme for unpopular file upload is minimal (especially for bigger files), it is caused mainly by the very low upload speed to the Dropbox storage. If the upload speed was higher then the cost would be proportionally bigger since the time to upload the actual data would shorten. Due to the constant cost $\mathcal{T}_{\text{const}}$ it is not possible to compute throughput of our scheme, but we can compute relative throughput per filesize considering sequential upload of uniform-sized files like $\mathsf{THR}(|F|) = (1/(\mathcal{T}_{\text{const}} + \mathcal{T}_{\text{rel}}(|F|)))$. Using concrete values, the $\mathsf{THR}(1\text{kB}) = 6.45\text{kBps}$, $\mathsf{THR}(1\text{MB}) = 6\text{MBps}$ and $\mathsf{THR}(64\text{MB}) = 69.4\text{MBps}$. To conclude, if the user would upload files of average size 1 MB he would not perceive any notable delay if the upload speed to the storage provider would be lower or equal to 6 MBps (respectively 48 Mbps). Note that the computation does not include the per-file storage provider upload initialization cost so the actual real speeds would be even higher.

The cost of the $\mathsf{Download}(F, \mathsf{U}_i)$ operation is very easy to analyse since it corresponds either to 1x AES128.$\mathsf{D}(|F|)$ for popular file or 2x AES128.$\mathsf{D}(|F|)$ for unpopular file plus the actual data transfer from the storage to the user. Since throughput of AES128.D on our testing machine is over 1 GBps (thanks to parallelization of the decryption process) it is unlikely that the cost would be noticeable compared to the actual data transfer.

Deduplication

The $\mathsf{Deduplicate}(\text{indexes}, \text{shares})$ algorithm can be decomposed into the following operations:

DE1 IRS sends a set of indexes and decryption shares to S

DE2 S threshold-decrypting symmetric key(s)

DE3 S symmetrically-decrypting unpopular file(s)

DE4 S discards unpopular files and stores one popular file

Since **DE1** is transfer of filesize-independent sets of indexes and shares, it is of near-constant cost and, assuming reasonably good connection of 1 MBps, $t = 1\,000$ and ping RTT from S to IRS as 200ms, the $\mathcal{T}(\mathbf{DE1}) = 600$ ms. **DE2** is filesize-independent and has a measured constant cost $\mathcal{T}(\mathbf{DE2}) = 266.3$ ms (standard deviation 1.42). **DE3** is filesize dependent and corresponds to AES128.D which reaches 1 GBps throughput in our test setting. **DE4** is the cost incurred by S implementation.

While the cost of deduplication is substantial, it occurs only once per file transiting states and the computationally-intensive part can be scheduled by the storage provider upon need. A greater limitation is the need to implement the functionality itself in the storage provider S. While the implementation is straightforward and should not be difficult to be performed by S "inside", it is almost impossible to achieve it by modifications done "from the outside".

7.4.4 Performance Comparison of Different Solutions

Since scheme initialization and user registration procedures are varied among the different solutions and are relatively rare (compared to file manipulation operations), we leave them out of the comparison. For completeness we stress that in 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 cost 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. The aggregate results plotted in Figure 7.5 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

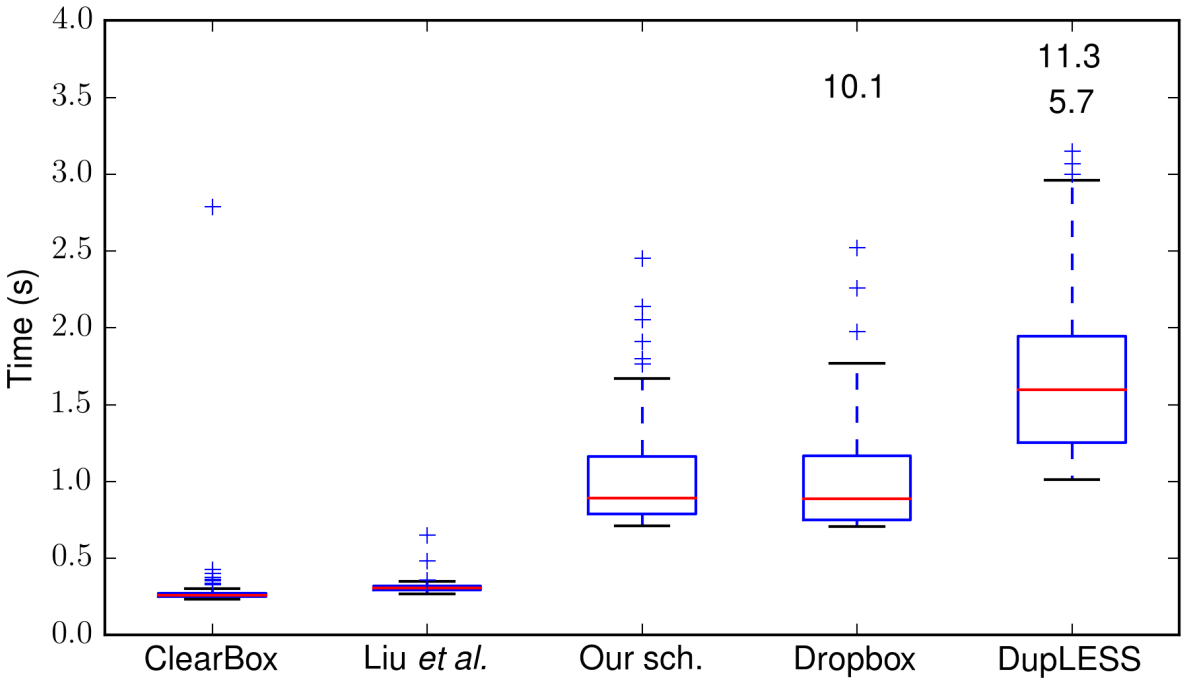

Figure 7.5: Average processing time of a Put request for a deduplicable file from a randomly chosen 100-file sample from the *UPC dataset*.

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.

To provide a more detailed analysis, we have chosen two Put requests – one with the lowest processing time and one with the highest processing time (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 Figure 7.6.

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 $i \in \{0, 1, .., 8\}$ (*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 Figure 7.7. 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 cost. 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.

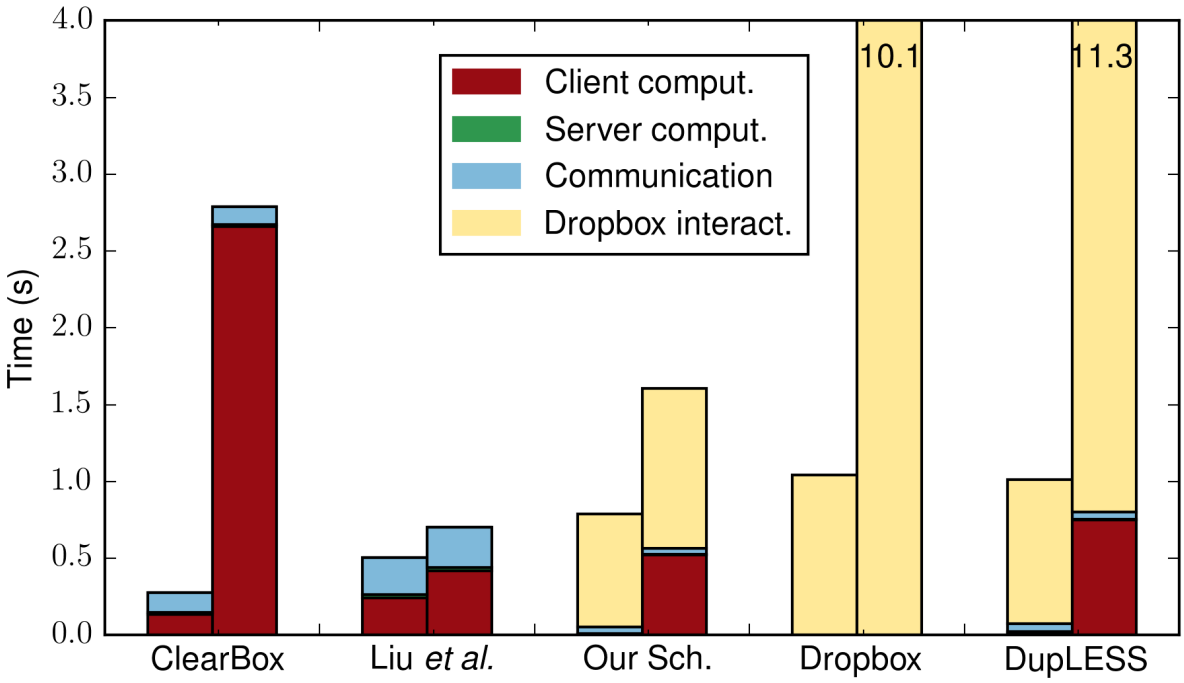

Figure 7.6: Put request with the lowest (left; 157 KB file) and the highest (right, 33 MB file) processing time split into individual costs (same sample as Figure 7.5).

Communication cost: We use *tcpdump* to measure communication on the network interface for a deduplicable 1MB file (cost is file-size-independent, provided that PoW in ClearBox is capped for 64 MB buffer as suggested). We repeated the Put operation 100x 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 cost depends on the PoW parameters, for the tested settings it did not reach 10 KB. The proposal of Liu *et al.* contains significant cost due to the PAKE processing (set to default 30 requests) and averaged at 110 KB per Put.

Non-deduplicable Put request: All prototypes inherit the Dropbox interaction cost of DupLESS (see Figure 7.7, 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 cost). DupLESS retains the cost as it does not differ between Put for deduplicable and non-deduplicable file. Even though ClearBox has the highest cost, 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 cost will be considerably higher than the processing cost once the storage "fills in" reasonably.

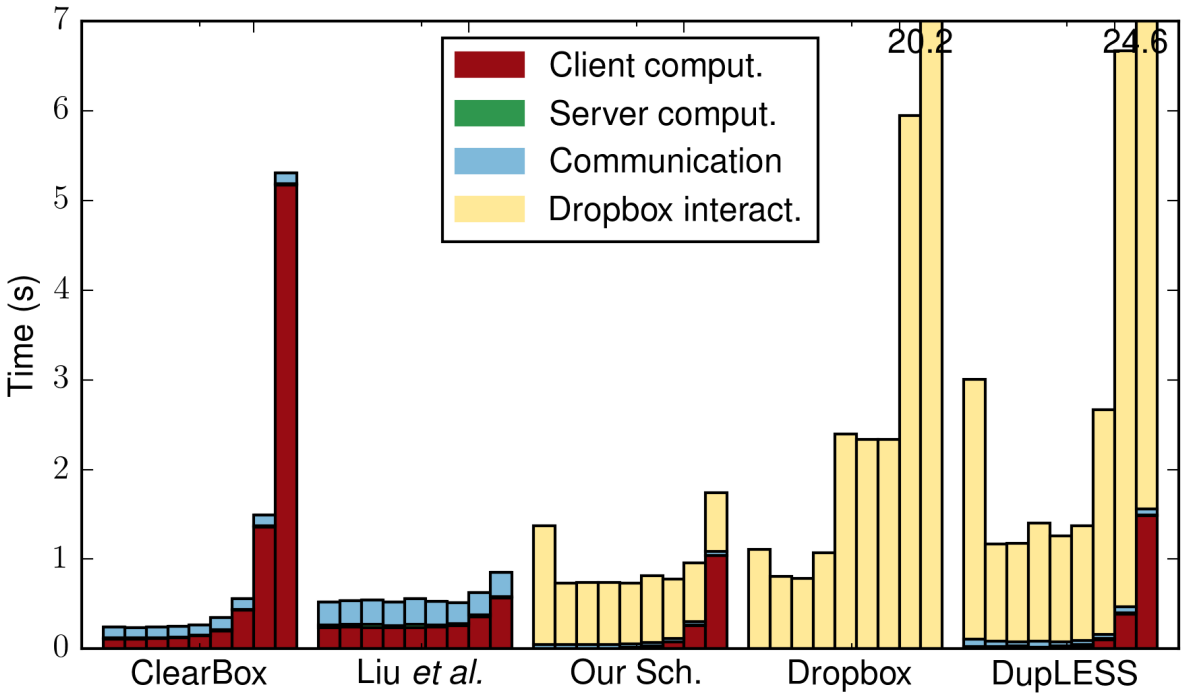

Figure 7.7: Processing time of Put requests for deduplicable files of size 2^{2i} KB for $i \in \{0, 1, .., 8\}$ (plotted left to right per prototype).

Get request: The comparison of Get requests processing reveals that all schemes perform near-equally. ClearBox adds computational cost 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 allow 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 costs 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. 850ms (considering 1MBps 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 1s for a 100MB file. This measurement was done on a separate cloud application as Dropbox does not support it.

7.5 Summary

Chapter 7 presented a metric to measure the space reduction efficiency of our scheme – the Space Reduction Ratio (SRR), and covered an analysis of the computation and

communication cost of our scheme and of representative examples of other secure data deduplication schemes.

With respect to the space reduction efficiency, ClearBox and DupLESS implement a perfect deduplication scheme and thus achieve the highest possible SRR for any dataset. Our scheme and the scheme proposed by Liu *et al.* have varying efficiency, based on the parameters of the scheme and properties of the dataset. Plain Dropbox was used only for comparison and does not offer any deduplication, thus no space reduction.

Considering the computation and communication cost, all analyzed schemes 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 analysed schemes 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 cost for Get requests. Considering the offered storage space reduction, all solutions create a win-win situation for both users and storage providers.

The results of the comparative analysis of performance of the different schemes from the computation and communication cost point of view demonstrate that there is no "winner" in this respect, each analyzed scheme has its pros and cons. Using the results of Put measurements, we demonstrate that, depending on the cloud back-end performance, our scheme is on par with ClearBox for smaller files and outperforms it for bigger files, the scheme of Liu *et al.* outperforms other schemes for bigger files but is ineffective for small files. However, the increased cost of ClearBox for bigger files is largely caused by the included Proof of Ownership mechanism that is not integrated in the other schemes and the ineffectiveness of the scheme by Liu *et al.* for small files is caused by the need to process the request also by other client(s) than only the uploading one, which, on the other hand, enabled the scheme to limit the amount of potentially exploitable information stored in the server application. Considering the ease of deployment, apart from ClearBox, all schemes require modification of the cloud back-end. This is cumbersome, but viewed from a different perspective, it enables the possibility to download files even if the server component of these schemes is down (which is impossible in the case of ClearBox).

Despite the fact that all the analysed schemes offer secure data deduplication, they are very different in many aspects. We recommend potential adopters to evaluate their needs and choose the scheme that best fits their requirements and environment from functional, security and performance perspectives. In Table 7.9 we list a few of the differentiating features we identified, that might help in the decision process. We stress that each feature can be considered as an advantage from one perspective and as a disadvantage from another perspective and the schemes could be modified to support some of the features

they do not include in their original design. The table should therefore serve more as a basic view of what features each scheme offers out of the box as it was originally published and designed.

Table 7.9: Feature comparison of different deduplication schemes; x - feature present; o - feature not present; x* - not measured in our performance analysis, based on information from the source paper

Feature	Clearbox	Liu *et al.*	Our scheme	DupLESS	ClouDedup
Perfect deduplication	x	o	o	x	x
No Indexing Server-like component	o	x	o	o	o
No active participation of users on deduplication	x	o	x	x	x
Transparent deduplication pattern attestation	x	o	o	o	o
Automatic differentiation of popular and unpopular files	o	o	x	o	o
Inherent resilience to storage-service-based side-channel attacks	o	o	x	o	x
Limited file-size influence on deduplicable `Put` cost	o	x	x	o	x*
Incorporated Proof of Ownership mechanism	x	o	o	o	o

Chapter 8

Conclusion

This thesis deals with the inherent tension between well established storage optimization methods and end-to-end encryption in a practical example of a cloud storage service scenario. Concretely, we focus on the issue of secure data deduplication, analyzing different views of security versus deduplication. Based on the analysis we build a secure deduplication scheme implementing the idea of popularity where different data require different protection based on how much they are shared (*i.e.* popular) among users. We present a construction of our scheme, evaluate it from both security and performance views, discuss it's limitations and possible ideas for their alleviation and compare our scheme with other state of the art secure data deduplication schemes.

Differently from the approach of related works that assume all files to be equally security-sensitive, we vary the security level of a file based on how popular that file is among the users of the system. This is a major switch in view of file security – the traditional view that takes file contents as the major factor when considering file sensitivity and potential level of protection of the file is replaced by a view that instead of contents considers file popularity and argues that once a file is "widely known", there is no point in keeping it "heavily protected". This novel view may be of independent interest.

Our proposed secure data deduplication scheme has two major advantages (provided the popularity-based classification is acceptable for the scheme users) – the users no longer need to manually classify sensitive files (since all files are first unpopular and thus protected using a semantically secure cryptosystem), and the transition between unpopular and popular state is automatic and does not require active user participation. The advantages come at the cost of disadvantages – our scheme has lower deduplication ratio than perfect deduplication schemes and the computation cost of scheme operations is not negligible.

To ease possible adoption of a secure deduplication scheme by cloud storage service providers we provide an extensive performance and security evaluation of our scheme and

compare it with other state of the art schemes. Our evaluation shows that there is no clear "winner" (*i.e.* the best secure deduplication scheme) among the proposals – each scheme has some pros and cons. Performance-wise, none of the evaluated 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).

Admittedly, secure data deduplication schemes (ours included) are not perfect and their concrete performance highly depends on the underlying dataset. However, the steeply increasing amount of data being stored in cloud storage services calls for usage of storage optimization methods and deduplication seems a viable candidate. This thesis can help the readers to understand the risks related to data deduplication and make decisions regarding potential secure data deduplication scheme adoption based on their concrete requirements and setup.

Despite existence of multitude of secure data deduplication scheme proposals, all of them use the deduplication index. The deduplication index is, by its nature, a leakage of information about file contents. However, without the deduplication index, it is not possible to find out that two files are the same (and thus can be deduplicated). All the schemes analysed in this work, including ours, solve this index-caused security weakness by handling it by a trusted component or requiring active participation of users storing the respective indexes locally. Finding a solution that would not use the deduplication index but rather some other innovative approach, without the inherent security weakness introduced by the deduplication index, is still an open problem.

Appendix A

PhD Studies – Overview and Results

Due to quite changing topics of interest during my PhD studies there is a lot of research that is not included in this thesis but was part of my PhD studies. For completeness I present a short time-ordered summary of my PhD studies and the respective publication results.

I started my PhD studies in 2011 with the original topic of SIP (Session Initiation Protocol), respectively VoIP (Voice over IP) security. The initial research showed that SIP servers (core building blocks of SIP infrastructure) are very prone to Denial of Service (DoS) attacks and there is no suitable protection available. To attract attention to the issue we first published (together with my supervisor) a paper describing the issue [42], including a simple SIP DoS attack tool as a proof of concept that the threat is real. Afterwards I designed and evaluated a few possible approaches how to defend against such attacks. The result of this research was another publication describing design of a DDoS protection solution tailored specifically for SIP servers [43]. During the SIP-oriented research, I found myself constantly lacking sources of real or at least real-like traffic. Therefore we decided to collaborate with a testing-oriented free SIP network iptel.org, represented by Jiri Kuthan, and prepared an analysis of real SIP traffic [44]. As the next step, we implemented a tool to anonymize SIP traffic such that it can be shared with public without risking potential disclosure of client/confidential data, and a portal where such anonymized traffic can be shared. To prove that anonymization does not remove interesting factors from the data we published an analysis of anonymized data yielding useful results [45]. When preparing the infrastructure and doing processing of the collected data, I became more familiar with cloud technologies and noted a new vast field of potentially very interesting security problems. After playing with the provided cloud solutions a bit, we decided with my supervisor that it might be worth switching topics to cloud security as this field was rapidly expanding and offering quite a few novel security challenges.

In the beginning of my "cloud security" PhD era I did an exploratory work to see what was already done, what is available and what is a likely candidate to pursuit. This preliminary work resulted in a short review summarizing fully homomorphic encryption (FHE) and multiple sources regarding the current state of the art research in the cloud security research area. Despite being very interesting, the FHE topic was already being researched by multiple teams around the world and the amount of "familiarization work" required to become an active part of this research was too high to be feasible for me at that, already advanced, stage of PhD studies. Not to waste the concise FHE survey that resulted from the exploratory work, I discussed with my colleagues at the university and we decided that it might be useful to attract students to the topic. Thus I have re-formed, simplified and translated the survey to Czech language and provided it is a study material for students of the "Introduction to Information Security" course. Next to the FHE topic, I also ended up with many other viable research options. We discussed them with my supervisor and formulated a few most promising topics to be pursued in further research. To evaluate the "attractiveness" of the topics to the research community I joined an open call offering student internships at IBM Research – Zurich and discussed the topics with their research team during the applicant interviews. As a result I was accepted for the internship with the topic focusing on secure deduplication. I started my 6-months internship and secure deduplication-oriented research in mid-2012 and continued till 2016, having a chance to do another 6-months internship at IBM Research – Zurich in 2014. The results of this research are described in this thesis and were also published in form of one patent [46], one conference publication [47] and one journal publication [48]. As part of the work at the internships I also participated in the backup and storage research area and, together with my IBM colleagues, we formulated a mechanism how to efficiently implement secure deduplication also for tape-based storages. This work also resulted in patent application [49]. Despite being very-well perceived by the research community, secure deduplication did not get much practical applications yet, since the solutions providing reasonable security levels are also more resource-requiring. At the time of the research, companies either didn't move to public cloud at all, preferring isolated private clouds, or used legal-based forms of security (Service Level Agreements with legally binding clauses) rather than technical security solutions. A few big companies created dominant cloud platforms that "are supposedly secure", though noone really explained how exactly this security is assured and didn't formalize (or disclose) the threat models and actors. This attitude is slowly changing now and secure deduplication is, among other cloud-security related technical solutions, very likely to become also practically adopted (or at least it seems from the growing number of patents being issued in the field). However, in 2016 it seemed the practical adoption will take a few years and after

a more theoretical research I was striving for some hands-on security that will likely be practically adopted quickly. A chance came in form of the emerging Internet of Things (IoT) – hitting the market very fast means very likely very poor security which inherently means a lot of potential for applied security research.

My research towards IoT security started as a joint work between university and a private company that started to replace their traditional production with smart (*i.e.* connected) devices. The first results of research were documents evaluating security of design of an Over the Air Update solution proposed for the new smart devices and security evaluation of a prototype smart device. Unfortunately, due to the confidential nature of the results (since security flaws, planned patches and new, more-secure, system designs must not be published to keep the company secure and business-like ahead of competitors) publication of these results is currently not possible. Also, a lot of the results of this work are concerning applied security, which is very interesting for in-field deployment, but not as much interesting to the research community. Evaluating my current research results, we concluded with my supervisor that the most impactful research that is already published is the work that focuses on the secure deduplication and so I decided to finish my PhD studies (*i.e.* write this thesis) focusing on this topic. It is very likely that in the future, the situation will change and IoT secure system designs will be widely published whereas the for-now mostly-theoretical field of secure deduplication will become more applied. But waiting for that time might well prove to be too long and since some new topic might gain my interest in the meantime, it seems a much better idea to finish PhD now, not to be doing it "virtually forever" *[smileys are not to be included in a serious research work].*

Appendix B

List of Publications

This list covers all publications, that I authored or co-authored during my PhD studies. Publications are sorted according to their citation count reported by Google Scholar (GS) and Web of Science (WoS). Data snapshot taken in February 2018.

Publication	Citations GS (WoS)
J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A secure data deduplication scheme for cloud storage", in *International Conference on Financial Cryptography and Data Security, 2014*	161 (30)
J. Stanek and L. Kencl, "SIPp-DD: SIP DDoS flood-attack simulation tool", in *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN), 2011*	16 (4)
J. Stanek, L. Kencl, and J. Kuthan, "Characteristics of real open SIP-server traffic", in *International Conference on Passive and Active Network Measurement, 2013*	7
J. Stanek and L. Kencl, "SIP protector: Defense architecture mitigating DDoS flood attacks against SIP servers", in *IEEE International Conference on Communications (ICC), 2012*	5 (1)
J. Stanek and L. Kencl, "Enhanced secure thresholded data deduplication scheme for cloud storage", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2016	4
J. Stanek, L. Kencl, and J. Kuthan, "Analyzing anomalies in anonymized SIP traffic", in *IFIP Networking Conference, 2014*	3
R. Cideciyan, J. Jelitto, S. Sarafijanovic, and J. Stanek, *US patent application 20140358871*, 2014. [Online]. Available: `http://www.freepatentsonline.com/y2014/0358871.html`	3
J. Jelitto, T. Mittelholzer, S. Sarafijanovic, A. Sorniotti, and J. Stanek, *US PATENT 9292532: Remote data storage*, 2016. [Online]. Available: `http://www.freepatentsonline.com/9292532.html`	1

Appendix C

Publications Annotated with University Requirements

This list covers all publications, that I authored or co-authored during my PhD studies. Publications are sorted according to the university requirements. Where author participation is not specifically noted then participation of all co-authors of the work was equal.

C.1 Publications Related to Thesis Topic

Publication Author participation (if not equal)	Type
J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A secure data deduplication scheme for cloud storage", in *International Conference on Financial Cryptography and Data Security, 2014* Stanek 40%, Kencl 15%, Androulaki 20%, Sorniotti 25%	Indexed by ISI*
J. Stanek and L. Kencl, "Enhanced secure thresholded data deduplication scheme for cloud storage", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2016	Journal with Impact Factor
R. Cideciyan, J. Jelitto, S. Sarafijanovic, and J. Stanek, *US patent application 20140358871*, 2014. [Online]. Available: `http://www.freepatentsonline.com/y2014/0358871.html`	Patent application
J. Jelitto, T. Mittelholzer, S. Sarafijanovic, A. Sorniotti, and J. Stanek, *US PATENT 9292532: Remote data storage*, 2016. [Online]. Available: `http://www.freepatentsonline.com/9292532.html`	Patent

* ISI - Institute for Scientific Information

C.2 Other Publications

Publication Author participation (if not equal)	Type
J. Stanek and L. Kencl, "SIPp-DD: SIP DDoS flood-attack simulation tool", in *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN), 2011* Stanek 60%, Kencl 40%	Indexed by ISI*
J. Stanek and L. Kencl, "SIP protector: Defense architecture mitigating DDoS flood attacks against SIP servers", in *IEEE International Conference on Communications (ICC), 2012* Stanek 60%, Kencl 40%	Indexed by ISI*
J. Stanek, L. Kencl, and J. Kuthan, "Characteristics of real open SIP-server traffic", in *International Conference on Passive and Active Network Measurement, 2013* Stanek 50%, Kencl 25%, Kuthan 25%	Other publication
J. Stanek, L. Kencl, and J. Kuthan, "Analyzing anomalies in anonymized SIP traffic", in *IFIP Networking Conference, 2014* Stanek 50%, Kencl 40%, Kuthan 10%	Indexed by ISI*

* ISI - Institute for Scientific Information

Bibliography

[1] C. Gentry, "A fully homomorphic encryption scheme", PhD thesis, Stanford University, 2009.

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

[3] 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 *ICDCS*, 2002, pp. 617–624. DOI: `10.1109/ICDCS.2002.1022312`. [Online]. Available: `http://dx.doi.org/10.1109/ICDCS.2002.1022312`.

[4] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage", *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010. DOI: `10.1109/MSP.2010.187`. [Online]. Available: `http://doi.ieeecomputersociety.org/10.1109/MSP.2010.187`.

[5] S. Keelveedhi, M. Bellare, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage", in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 179–194. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bellare`.

[6] A. Shamir, "How to share a secret", *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979. DOI: `10.1145/359168.359176`. [Online]. Available: `http://doi.acm.org/10.1145/359168.359176`.

[7] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-locked encryption and secure deduplication", in *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, 2013, pp. 296–312. DOI: `10.1007/978-3-642-38348-9_18`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-38348-9_18`.

[8] R. Housley, RSA Laboratories, W. Polk, NIST, W. Ford, VeriSign, D. Solo, and Citigroup, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile", RFC Editor, RFC 3280, 2002, pp. 1–129. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc3280.txt`.

[9] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985. DOI: `10.1109/TIT.1985.1057074`. [Online]. Available: `http://dx.doi.org/10.1109/TIT.1985.1057074`.

[10] *Hash of plaintext as key?*, Cypherpunks mail archive, https://groups.google.com/forum/msg/mail.cypherpunks/RLHsypJfd7o/CjGOsOxqwT0J, 1996.

[11] *Tahoe-lafs*, Tahoe-LAFS webpage, https://tahoe-lafs.org/trac/tahoe-lafs, 2007.

[12] *Dropbox*, Dropbox webpage, https://www.dropbox.com/, 2007.

[13] S. Goldwasser and S. Micali, "Probabilistic encryption", *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984. DOI: 10.1016/0022-0000(84)90070-9. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(84)90070-9.

[14] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik, "Estimation of deduplication ratios in large data sets", in *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, 2012, pp. 1–11. DOI: 10.1109/MSST.2012.6232381. [Online]. Available: http://dx.doi.org/10.1109/MSST.2012.6232381.

[15] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario", in *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference 2009, Haifa, Israel, May 4-6, 2009*, 2009, p. 8. DOI: 10.1145/1534530.1534541. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534541.

[16] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication", in *Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008, Companion Proceedings*, 2008, pp. 12–17. DOI: 10.1145/1462735.1462739. [Online]. Available: http://doi.acm.org/10.1145/1462735.1462739.

[17] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system", in *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference 2009, Haifa, Israel, May 4-6, 2009*, 2009, p. 6. DOI: 10.1145/1534530.1534539. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534539.

[18] 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, 2014. DOI: 10.1109/TPDS.2013.173. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.173.

[19] M. W. Storer, K. M. Greenan, D. D. E. Long, and E. L. Miller, "Secure data deduplication", in *Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS 2008, Alexandria, VA, USA, October 31, 2008*, 2008, pp. 1–10. DOI: 10.1145/1456469.1456471. [Online]. Available: http://doi.acm.org/10.1145/1456469.1456471.

[20] 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, 2014. DOI: 10.1109/TPDS.2013.284. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.284.

[21] P. Puzio, R. Molva, M. Önen, and S. Loureiro, "Cloudedup: Secure deduplication with encrypted data for cloud storage", in *IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, Bristol, United Kingdom, December 2-5, 2013, Volume 1*, 2013, pp. 363–370. DOI: 10.1109/CloudCom.2013.54. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2013.54.

[22] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 886–900.

[23] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015, pp. 874–885. DOI: 10.1145/2810103.2813623. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813623.

[24] P. Meye, P. R. Parvédy, F. Tronel, and E. Anceaume, "A secure two-phase data deduplication scheme", in *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*, 2014, pp. 802–809. DOI: 10.1109/HPCC.2014.134. [Online]. Available: http://dx.doi.org/10.1109/HPCC.2014.134.

[25] Y. Duan, "Distributed key generation for encrypted deduplication: Achieving the strongest privacy", in *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, 2014, pp. 57–68. DOI: 10.1145/2664168.2664169. [Online]. Available: http://doi.acm.org/10.1145/2664168.2664169.

[26] J. Xu, E. Chang, and J. Zhou, "Weak leakage-resilient client-side deduplication of encrypted data in cloud storage", in *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, 2013, pp. 195–206. DOI: 10.1145/2484313.2484340. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484340.

[27] J. Li, Y. K. Li, X. Chen, P. P. Lee, and W. Lou, "A hybrid cloud approach for secure authorized deduplication", *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1206–1216, 2015.

[28] C.-M. Yu, "Xdedup: Efficient provably-secure cross-user chunk-level client-side deduplicated cloud storage of encrypted data", 2016, http://eprint.iacr.org/2016/1041.

[29] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems", in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 491–500. DOI: 10.1145/2046707.2046765. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046765.

[30] R. D. Pietro and A. Sorniotti, "Boosting efficiency and security in proof of ownership for deduplication", in *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, 2012, pp. 81–82. DOI: 10.1145/2414456.2414504. [Online]. Available: http://doi.acm.org/10.1145/2414456.2414504.

[31] R. Di Pietro and A. Sorniotti, "Proof of ownership for deduplication systems: A secure, scalable, and efficient solution", *Computer Communications*, vol. 82, pp. 71–82, 2016.

[32] J. R. Douceur, "The sybil attack", in *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, 2002, pp. 251–260. DOI: 10.1007/3-540-45748-8_24. [Online]. Available: http://dx.doi.org/10.1007/3-540-45748-8_24.

[33] 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. DOI: 10.1007/s00145-004-0311-z. [Online]. Available: http://dx.doi.org/10.1007/s00145-004-0311-z.

[34] B. Lynn, *The pairing-based crypto. library*, PBC webpage, http://crypto.stanford.edu/pbc/, 2007.

[35] G. Ateniese, J. Camenisch, S. Hohenberger, and B. de Medeiros, "Practical group signatures without random oracles", *IACR Cryptology ePrint Archive*, vol. 2005, p. 385, 2005. [Online]. Available: http://eprint.iacr.org/2005/385.

[36] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data", in *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, 2006, pp. 89–98. DOI: 10.1145/1180405.1180418. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180418.

[37] J. Camenisch, S. Hohenberger, and A. Lysyanskaya, "Balancing accountability and privacy using e-cash (extended abstract)", in *Security and Cryptography for Networks, 5th International Conference, SCN 2006, Maiori, Italy, September 6-8, 2006, Proceedings*, 2006, pp. 141–155. DOI: 10.1007/11832072_10. [Online]. Available: http://dx.doi.org/10.1007/11832072_10.

[38] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf, "Pseudonym systems", in *Selected Areas in Cryptography, 6th Annual International Workshop, SAC'99, Kingston, Ontario, Canada, August 9-10, 1999, Proceedings*, 1999, pp. 184–199. DOI: 10.1007/3-540-46513-8_14. [Online]. Available: http://dx.doi.org/10.1007/3-540-46513-8_14.

[39] *The pirate bay 2008-12 dataset*, University Of Zurich, Department of Informatics webpage, http://www.csg.uzh.ch/publications/data/piratebay.html, 2008.

[40] *Debian popularity contest*, Debian Popularity Contest webpage, http://popcon.debian.org/, 2015.

[41] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López, "Faster explicit formulas for computing pairings over ordinary curves", in *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, 2011, pp. 48–68. DOI: 10.1007/978-3-642-20465-4_5. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20465-4_5.

[42] J. Stanek and L. Kencl, "SIPp-DD: SIP DDoS flood-attack simulation tool", in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, IEEE, 2011, pp. 1–7.

[43] ——, "SIP protector: Defense architecture mitigating DDoS flood attacks against SIP servers", in *2012 IEEE International Conference on Communications (ICC)*, IEEE, 2012, pp. 6733–6738.

[44] J. Stanek, L. Kencl, and J. Kuthan, "Characteristics of real open SIP-server traffic", in *International Conference on Passive and Active Network Measurement*, Springer, 2013, pp. 187–197.

[45] ——, "Analyzing anomalies in anonymized SIP traffic", in *Networking Conference, 2014 IFIP*, IEEE, 2014, pp. 1–9.

[46] J. Jelitto, T. Mittelholzer, S. Sarafijanovic, A. Sorniotti, and J. Stanek, *US PATENT 9292532: Remote data storage*, 2016. [Online]. Available: `http://www.freepatentsonline.com/9292532.html`.

[47] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A secure data deduplication scheme for cloud storage", in *International Conference on Financial Cryptography and Data Security*, Springer, 2014, pp. 99–118.

[48] J. Stanek and L. Kencl, "Enhanced secure thresholded data deduplication scheme for cloud storage", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2016, ISSN: 1545-5971. DOI: `10.1109/TDSC.2016.2603501`.

[49] R. Cideciyan, J. Jelitto, S. Sarafijanovic, and J. Stanek, *US patent application 20140358871*, 2014. [Online]. Available: `http://www.freepatentsonline.com/y2014/0358871.html`.