Ref. Ares(2015)6069002 - 31/12/2015

Project title: Enforceable Security in the Cloud to Uphold Data Ownership
Project acronym: ESCUDO-CLOUD
Funding scheme: H2020-ICT-2014
Topic: ICT-07-2014
Project duration: January 2015 – December 2017

D2.1

Report on data protection techniques

Editors: Stefano Paraboschi (UNIBG)
Sara Foresti (UNIMI)
Giovanni Livraga (UNIMI)
Reviewers: Andrew Byrne (EMC)
Carlos Rodrigo Rubia Marcos (WT)

Abstract

Cloud Computing introduces a wide variety of benefits to customers looking to reduce the cost and increase the efficiency of their IT infrastructure. In particular for smaller organisations, moving workloads, applications and data to the cloud can bring with it improved management and security services. However, these benefits come at a cost. As customer data and workloads move to the cloud, there is a loss of control and visibilty over how they are accessed, processed and stored. This introduces several security and privacy issues that need to be properly address to protect data in the cloud. To address these issues, the first fundamental consideration requires the protection of the data itself stored at a single cloud provider. This requires the availability of techniques for protecting data confidentiality from any unauthorized party (including the cloud provider itself), and for assessing data integrity and availability when the cloud provider cannot be considered trustworthy for correctly managing the data it stores. In this document, we illustrate encryption-based and fragmentation-based techniques aiming at protecting the confidentiality of data stored and managed in the cloud, as well as solutions allowing data owners to assess the integrity and availability of their data outsourced to cloud providers. Attention must also be paid to attacks on confidentiality that derives from the ability of the cloud provider to monitor access to the data.

Type	Identifier	Dissemination	Date
Deliverable	D2.1	Public	2015.12.31

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644579. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract No 150087. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission or the Swiss Government.

ESCUDO-CLOUD Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	British Telecom	BT	United Kingdom
3.	EMC Corporation	EMC	Ireland
4.	IBM Research GmbH	IBM	Switzerland
5.	SAP SE	SAP	Germany
6.	Technische Universität Darmstadt	TUD	Germany
7.	Università degli Studi di Bergamo	UNIBG	Italy
8.	Wellness Telecom	WT	Spain

Disclaimer: The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The below referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2015 by Università degli Studi di Milano, IBM Research GmbH, Università degli Studi di Bergamo.

Versions

Version	Date	Description
0.1	2015.11.23	Initial Release
0.2	2015.12.14	Second Release
1.0	2015.12.31	Final Release

List of Contributors

This document contains contributions from different ESCUDO-CLOUD partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Stefano Paraboschi (UNIBG), Sara Foresti (UNIMI), Giovanni Livraga (UNIMI)
Chapter 1: Cryptographic techniques for data confidentiality	Elli Androulaki (IBM), Angelo de Caro (IBM), Christian Cachin (IBM), Nikola Knezevic (IBM)
Chapter 2: Fragmentation techniques for data confidentiality	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Giovanni Livraga (UNIMI), Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 3: Techniques for data integrity	Christian Cachin (IBM), Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Giovanni Livraga (UNIMI), Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 4: Techniques for access confidentiality	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 5: Conclusions	Stefano Paraboschi (UNIBG)

Contents

Executive Summary **11**

1 Cryptographic techniques for data confidentiality **13**
1.1 Encryption in storage systems and cloud storage 13
1.1.1 Traditional storage stack 13
1.1.2 Encryption in the storage stack 14
1.1.3 Encryption for networked storage and cloud storage 16
1.2 Client-side encryption and key management 18
1.3 Deduplication and cloud storage 19

2 Fragmentation techniques for data confidentiality **22**
2.1 Protection requirements and techniques 22
2.1.1 Confidentiality constraints 22
2.1.2 Fragmentation and encryption 23
2.2 Two can keep a secret 24
2.2.1 Fragmentation model 24
2.2.2 Fragmentation metrics 26
2.2.3 Computing an optimal fragmentation 27
2.3 Multiple fragments 28
2.3.1 Fragmentation model 28
2.3.2 Fragmentation metrics 30
2.3.3 Computing an optimal fragmentation 32
2.4 Keep a few 33
2.4.1 Fragmentation model 33
2.4.2 Fragmentation metrics 34
2.4.3 Computing an optimal fragmentation 36

3 Techniques for data integrity **37**
3.1 Deterministic approaches 37
3.1.1 Digital signatures 37
3.1.2 Authenticated data structures 38
3.2 Probabilistic approaches 40
3.3 Proof of possession and retrievability 41
3.4 Integrity assurance towards multiple verifiers 42

4 Techniques for access confidentiality **43**
4.1 Motivation 43
4.2 Violations of access privacy 44
4.3 Shuffle index 46
4.4 Security analysis 48
4.5 Overhead analysis 50

5 Conclusions **55**

Bibliography **56**

List of Figures

1.1 A traditional storage stack as found in operating systems today 14
1.2 An example of full-disk encryption through an encryption proxy that is directly connected to the host (SCSI) or over the network (SAN, iSCSI), and performs encryption immediately above the drive layer. Gray parts denote encrypted data . 15
1.3 File-system encryption inside the file-system layer itself generally performs well and has access to relevant metadata in the file system. Gray parts denote encrypted data 15
1.4 A virtual file system, layered on top of an ordinary file system, may provide transparent encryption to applications without the constraints of being integrated into the operating system. Gray parts denote encrypted data 16
1.5 Schema of a networked or a cloud-based storage system, where the client-side application accesses a storage resource over a network or the Internet, respectively 16
1.6 The four different ways of realizing encryption for a cloud storage system. Gray parts denote encrypted data 17

2.1 An example of a relation (a) and of confidentiality constraints over it (b) 23
2.2 Two can keep a secret 25
2.3 An example of a correct fragmentation of relation MEDICALDATA in Figure 2.1(a) in the two can keep a secret scenario 26
2.4 An example of affinity matrix 27
2.5 Multiple fragments 28
2.6 An example of correct fragmentation of relation MEDICALDATA in Figure 2.1(a) in the multiple fragments scenario 30
2.7 An example of affinity matrix in the multiple fragments scenario 31
2.8 Keep a few 33
2.9 An example of a correct fragmentation of relation MEDICALDATA in Figure 2.1(a) in the keep a few scenario 34
2.10 An example of size of attributes (a) and query workload (b) for relation MEDICALDATA in Figure 2.1(a) 35

3.1 A Merkle hash tree over attribute `Name` of relation MEDICALDATA in Figure 2.1 . 38
3.2 A skip list for set $\mathscr{S}=\{5,6,8,9,10\}$ with three levels (a), search process for key value 9 (b), and verification object for a query searching for value 9 (c) 39

4.1 An example of abstract (a) and logical (b) representation of an unchained $B+$-tree, and of the corresponding view of the server (c) 47
4.2 Evolution of the shuffle index 49

4.3 Access time of the shuffle index in a switched Fast Ethernet LAN as a function of the number of covers (a) and of the size of the cache (b). The dashed line (baseline) represents the service time observed using a plain encrypted index with the same height. 51
4.4 Number of KiB exchanged as a function of the number of covers (a), and of the size of the cache (b) 53
4.5 Access time of the shuffle index in a WAN configuration as a function of the number of covers (a), and of the size of the cache (b). The dashed line (baseline) represents the service time observed using a plain encrypted index with the same height. 53

List of Tables

2.1 Classification of the metrics in the multiple fragments scenario 30
2.2 Classification of the metrics in the keep a few scenario 36

Executive Summary

Moving services to external cloud providers for data storage and management inevitably raises major concerns about the security and privacy of the outsourced data that now falls outside the direct control of their owner. The first fundamental step for protecting data in the cloud requires ensuring adequate protection to the stored data itself, outsourced at a single provider. This, in turn, requires different problems to be tackled, ensuring data confidentiality, integrity, and availability. The goal of this document is to illustrate basic protection techniques that can be adopted to ensure protection of data at rest, stored at a single cloud provider.

In most scenarios, the cloud provider storing the data is considered *honest-but-curious*, that is, it is trusted to correctly manage the data but not trusted to respect its confidentiality. Data confidentiality represents therefore a key property to be guaranteed, meaning that neither the cloud provider, nor unauthorized users, can access the (plaintext) values of the outsourced data. Applying a level of *encryption* to the data before outsourcing represents a natural approach that can be adopted by data owners to protect data confidentiality. However, encrypting the entire data collection before outsourcing might be excessive in scenarios in which the sensitive information to be protected is represented by associations among data values, rather than the values themselves. For instance, while the information that a given individual has been hospitalized at a given clinic, and the fact that the same clinic treats a given rare condition, might not be considered sensitive, there is a global consensus that the association between the identity of a patient and his/her condition is not to be disclosed. In this case, a promising solution is represented by replacing or complementing encryption with *data fragmentation*, splitting data in different (non-linkable) vertical views over the original data collection so that sensitive associations are broken (e.g., name and condition of patients are split in two different fragments). Encryption and fragmentation can be combined in several ways, leading to different protection models and techniques to ensure adequate protection to data confidentiality.

As well as our so called honest-but-curious cloud providers, within the cloud market there also exists a number of providers that might not be considered trustworthy for even correctly managing the data they store. In this case, data integrity and availability should also be adequately protected. This, in turn, ensures that neither the cloud provider, nor unauthorized users, can improperly tamper with the outsourced resources without being detected. Cryptographic techniques can be adopted to provide data owners with means to verify that their outsourced resources have not been improperly tampered with. Digital signatures and authenticated data structures can in fact be adopted to build practical techniques ensuring the integrity of data in storage with deterministic guarantees (i.e., providing full confidence). Complementing digital signatures with other techniques (such as controlled data replication and/or fake data insertion) can also lead to the design of a family of solutions providing integrity guarantees in a probabilistic way, while data availability can be assessed exploiting Proofs of Retrievability and Proofs of Data Possession techniques.

An additional threat is represented by the ability of the cloud provider to monitor access requests to the data, building in this way some information about the content of the data (even when encrypted) and the behavior of the user. To mitigate this exposure, techniques are being developed by the research community [SS13, SvS+13, WSC08, WSS09] that aim at voiding or significantly reducing the information that the storage provider can derive from their control over the physical representation of the data. A technique that received specific interest in ESCUDO-CLOUD is the *shuffle index* [DFP+15], a variant of B-tree that offers the ability to efficiently access large collections of encrypted data while providing protection against a cloud provider attempting to extract information through the monitoring of the access to the data. The basic mechanism adopted consists of a continuous reorganization of the data, creating additional access requests which both hide the real target among a group of potential covers and provide physical locations to be shuffled after the execution of the access request.

The remainder of this document is organized as follows. Chapter 1 illustrates how data confidentiality can be ensured through the use of encryption-based techniques. Chapter 2 discusses fragmentation-based solutions that, either complementing or departing from encryption, can be effectively adopted to protect the confidentially of data in storage. Chapter 3 addresses the problem of guaranteeing data integrity and availability, illustrating techniques providing both deterministic and probabilistic guarantees. Chapter 4 discusses access confidentiality and the design of the shuffle index.

1. Cryptographic techniques for data confidentiality

This chapter illustrates the basic approach of using encryption to keep data confidential, with respect to typical storage systems and cloud-storage services. Section 1.1 contains a model for networked storage systems that explains where encryption operations may be applied in practical systems. Section 1.2 focuses on the client side encryption and key-management aspects related to cloud storage. Section 1.3 addresses the problem of compressing data and eliminating duplicate data. Modern storage systems can benefit tremendously from compressing the stored data by eliminating duplicate copies of the same data; this technique is known as *deduplication*. Deduplication of encrypted data poses interesting problems, which are addressed in here.

1.1 Encryption in storage systems and cloud storage

Encrypting data prior to outsourcing it and sending it away represents perhaps the oldest and most effective solution for protecting data confidentiality. However, this straightforward approach may quickly face difficulties during realization, when confronted with constraints imposed by practical environments.

As storage services are typically provided by distributed systems, and the cloud-computing model by definition is distributed, we consider here networked storage systems as our primary focus.

Encryption for stored data (*data-at-rest*) in contrast to data that travels over a network (*data-in-transit*) may use symmetric cryptosystems. Today, most bulk encryption systems primarily rely on the AES cipher, for which fast software and hardware implementations are widely available [Xu10]. As asymmetric cryptosystems are generally slower than symmetric ones, bulk data is almost always encrypted using symmetric cryptosystems. In such cases, best security practices suggest using a key whose scope is limited to the immediate context of the encryption step, and only key-management operations may use asymmetric cryptosystems.

1.1.1 Traditional storage stack

As shown in Figure 1.1, a traditional storage system without encryption contains at least the following four elements, roughly: (1) *Application* layer that accesses the file system through standard system interfaces, such as Unix' *POSIX* file-system API; (2) the *file system* is the most complex part in the storage stack, as it is responsible for organizing the files into directories and adding attributes, while also providing operations to the application, and translating these operations into the fixed-size blocks that normally reside on the block device; (3) a *block device* is an abstraction provided by the operating system that interfaces the disk-drive layer; (4) at the bottom of this stack there is a *disk drive* or *storage volume*, where the stored data ultimately resides. It handles only

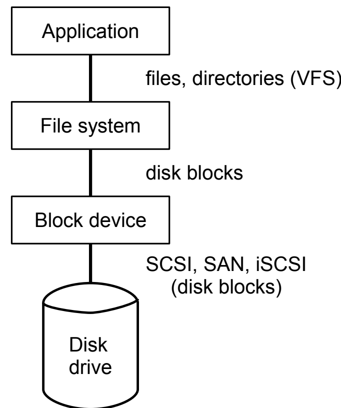


Figure 1.1: A traditional storage stack as found in operating systems today

fixed-size blocks and knows nothing about directories, attributes, or files. Also called the *block layer*, this is accessed over a standardized interface (for example SATA or SCSI) in a local computer, and such access may also be provided over a network (see Section 1.1.3). The "disk" may contain further layers, such as those found in solid-state drives (SSD).

1.1.2 Encryption in the storage stack

Storage encryption aims at providing transparent access to data that is stored in encrypted form, offering this to an application at the top of the storage stack. The application may not be aware that encryption takes place at some lower level. Hence, the main design feature of storage encryption methods lies in the *choice* of the storage layer where encryption takes place. According to the model of Figure 1.1, encryption may take place *inside* most of the shown components (file system, block device, disk drive) or *between* any two components. In the latter case the encryption service is provided by an additional component whose only goal is to encrypt data transparently.

The following sub-sections will consider some prominent examples how this technology has been used in practice.

Full-disk encryption

As illustrated in Figure 1.2, with the *full-disk encryption* (FDE) technology, data is encrypted at the level of logical drives. This usually takes place inside the disk drive's enclosure. In the case where the host's block device is connected over a network to some storage target (for example, a generic iSCSI target, or a SAN Volume Controller [Cor15]), then the encryption function may also be performed by a transparent network proxy, in the same place of the storage stack. Such solutions are very popular in the market [Gro09]. However, they cannot distinguish higher-level attributes of the stored data, and therefore they protect all data with the same key. Thus the protection guarantees are coarse-grained, essentially preventing only against unauthorized access to the disks.

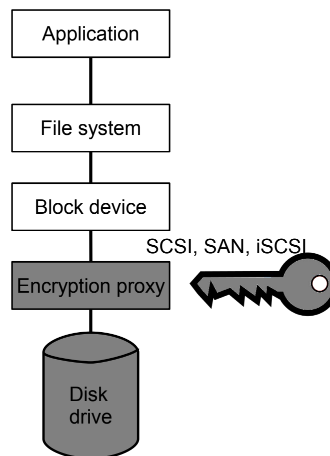


Figure 1.2: An example of full-disk encryption through an encryption proxy that is directly connected to the host (SCSI) or over the network (SAN, iSCSI), and performs encryption immediately above the drive layer. Gray parts denote encrypted data

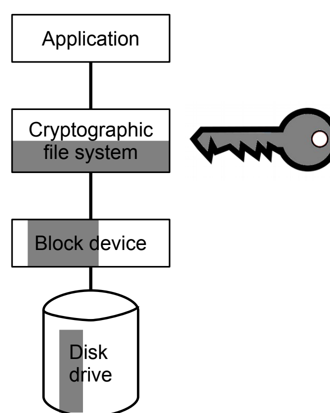


Figure 1.3: File-system encryption inside the file-system layer itself generally performs well and has access to relevant metadata in the file system. Gray parts denote encrypted data

Integrated file-system encryption

As depicted in Figure 1.3, with *integrated file-system encryption*, the traditional file system is enhanced to perform encryption in the data path. This means that files, directories and metadata may be encrypted, effectively also hiding file sizes and directory structure from an adversary that may access the lower layers. There are many popular solutions following this model, including *Microsoft Windows EFS*, the *IBM AIX JFS2 Encrypted File System*, or *Oracle ZFS encryption*. Due to the close integration with the native file system, the performance degradation is usually only minimal and only a little more noticeable than with encryption at the block layer (e.g., full-disk encryption). A single disk could be used by multiple filesystems, as depicted on Figure 1.3, where only a part of the disk contains encrypted data.

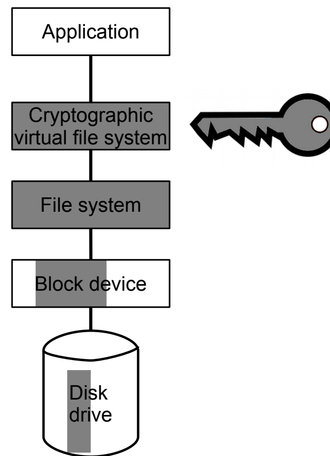


Figure 1.4: A virtual file system, layered on top of an ordinary file system, may provide transparent encryption to applications without the constraints of being integrated into the operating system. Gray parts denote encrypted data

Encryption in a virtual file system

Figure 1.4 shows a *virtual cryptographic file system*, which sits transparently between the file-system interface of the operating system (e.g., POSIX) and an application. This can be provided inside the operating system kernel (e.g., *eCryptFS* in Linux) or outside the kernel in user space (e.g., *encfs* using *FUSE* in Linux). Such systems typically expose the directory structure, the length of files, and so on because they are not taking care of space allocation. On the other hand, they are simpler to realize and more modular than encryption inside the file-system.

1.1.3 Encryption for networked storage and cloud storage

At any layer in the storage stack a network may be inserted, in the sense that the operating system accesses the layer below over a network. This results in a *networked storage system*, as shown in

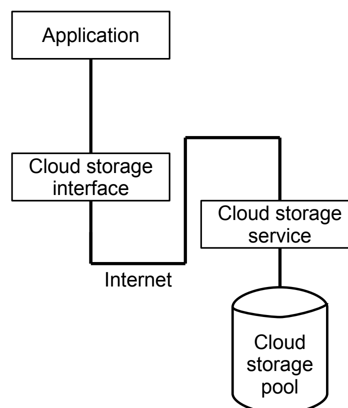
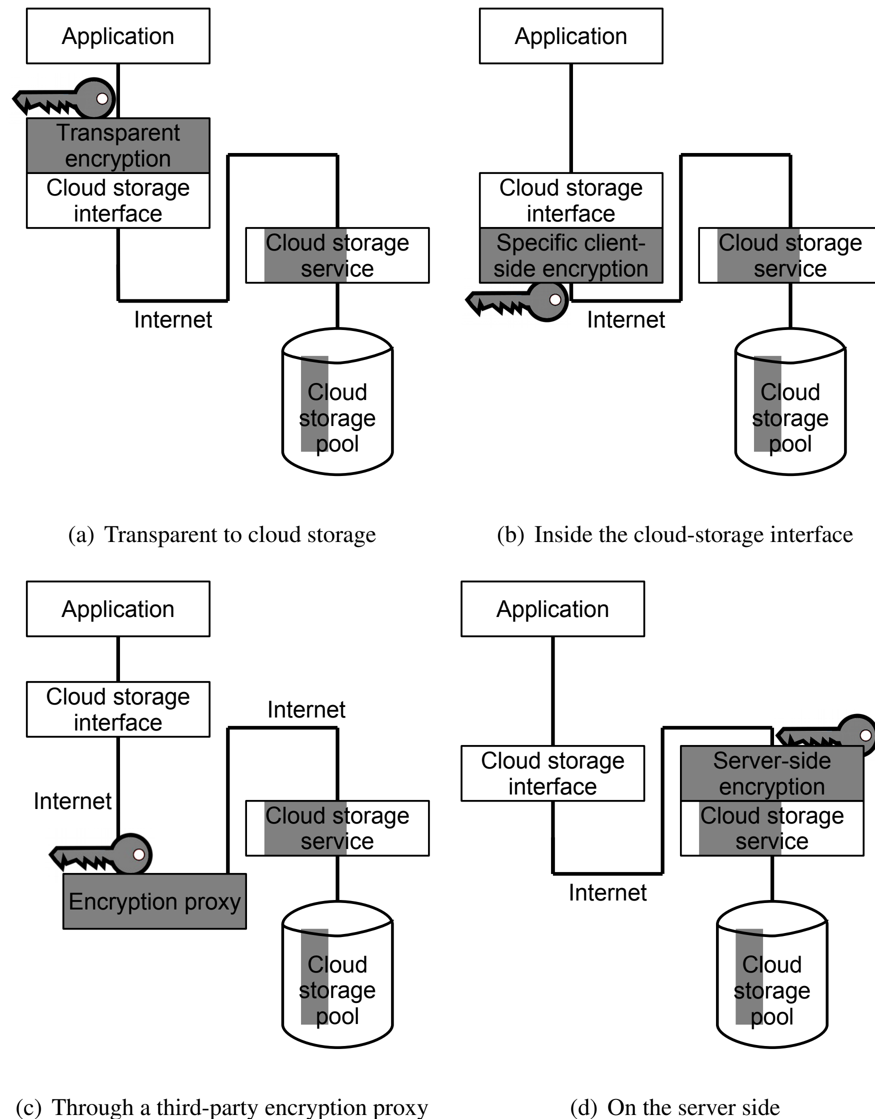


Figure 1.5: Schema of a networked or a cloud-based storage system, where the client-side application accesses a storage resource over a network or the Internet, respectively



(a) Transparent to cloud storage

(b) Inside the cloud-storage interface

(c) Through a third-party encryption proxy

(d) On the server side

Figure 1.6: The four different ways of realizing encryption for a cloud storage system. Gray parts denote encrypted data

Figure 1.5. Cloud storage is a special case of this model, where the storage resource is often accessed over the Internet on the application level, e.g., through a REST interface [FT02]. However, other forms of networked storage access also exist in cloud computing environments, for example, when a virtual machine hosted in a cloud data-center accesses the virtual disk images provided by a block-storage service; in this case, the network access to cloud storage would occur at the block-device layer.

Exploiting the multiple options for encrypting data in a storage system shown in the previous section, it is possible to combine these with the layer on which the remote access in networked storage occurs. For example, an encryption layer can be inserted on the client side *above* the cloud-storage network interface or *below*. Moreover, most of the possible combinations are found in today's cloud-computing infrastructure.

In Figure 1.6 the different options for placing an encryption layer into a cloud-storage system are shown in more detail. In the first variant (Figure 1.6(a) data is encrypted *before* it enters into

the layer that accesses the network. This means that the communication over the network and all specific functions of the networked storage system treat the encrypted data transparently and need no modifications. For example, a cloud-hosted virtual machine that allocates an encrypted disk volume (in the operating system) and consumes the raw storage data from a networked block store falls under this model.

Alternatively, the encryption layer can be entered *below* the local network-storage module or also *inside* that module, so that client-side encryption occurs transparently to the application running on top of the cloud-storage interface (see Figure 1.6(b)). This has the advantage that the application does not need to be changed. In the example where a virtual machine accesses a virtual block device emulated by the underlying hypervisor, and encryption is added inside the block-device driver in the hypervisor, the virtual machine does not know its data is being encrypted locally. Yet the virtual machine can benefit from the protection provided by data encryption, as its machine images are stored in encrypted form. An example implementation of this approach can be found in the encryption option of the Cinder volume driver of OpenStack.

The first two options place the encryption operation and control over the keys on the client side. Hence, they fully support the goal that the clients retain control over their data in the cloud.

Figure 1.6(c) denotes a variant where data is encrypted by a third-party encryption proxy. In this scenario, the client offloads the cryptographic operations to a trusted third-party. This could benefit from avoiding expensive cryptographic operations at the client side, but relies on trusting the third-party, and the use of secure channels between the client and the third-party. Nevertheless, the client still benefits from the protection provided by data encryption, as the storage provider only receives encrypted data.

A fourth form of encryption found in cloud storage places the cryptographic operations under the control of the storage provider on the server side, as illustrated in Figure 1.6(d). In this case, the provider runs the encryption operations and, in principle, may get access to the keys. However such a model is still beneficial because it defends against insider attacks in the provider's infrastructure, which is typically large, networked, and much more complex than shown here. In this model two different options have been explored for managing the encryption keys: It is possible that encryption keys are managed by the provider as well, in a "more secure" environment than the data itself. On the other hand, the client may supply encryption keys (on a protected channel) together with the requests to the cloud storage service, such that the service (by default) never stores client encryption keys; it only uses a key during access of the client. This model provides arguably more security, as the time window during which keys are exposed on the provider's side is limited.

1.2 Client-side encryption and key management

Section 1.1.3 has introduced the concept of the *client-side* encryption and contrasted it with *server-side* encryption. This section discusses the trade-offs related to the client-side encryption and key management.

In essence, cloud computing and storage consists of many interconnected components. Storing data or outsourcing computation to clouds involves additional risks, as a vulnerability in one of these components may introduce a risk to another component. In addition to attacking the components directly, the attackers could also exploit weaknesses between components, processes, or human control. Thus, the infrastructure security contains fundamental challenges in providing security guarantees. On the other hand, client-side encryption provides a most effective solution

for protecting data and computation confidentiality.

Nevertheless, security guarantees coming from client-side encryption carry a hidden cost in performance and scalability, as more computation has to be done on the client side and processes of key-management become more involved. As these two factors are the main drivers of cloud adoption, client-side encryption with client-side key-management is a viable solution only for clients (or companies) that need the highest level of security. In addition, as discussed in Section 1.3, encryption on the client side adds challenges in efficient data-deduplication, a powerful method of reducing the overall cost of running a cloud. Otherwise, server-side encryption provides some benefits, like ease of use, scalability, lower cost, and simpler server-side computation.

When it comes to cloud storage encryption, there are three main factors to consider with each approach: the encryption point (client- or server-side), the unit of encryption (none or document based), and the key holder/provider (client- or server-provided keys). The first factor has been discussed in the previous section. The factor relates to the control point of key management and offers only a narrow set of choices. Usually the encryption of a storage system is more robust when it uses per-document or per-object keys, since when using the same key for a large scope, the risk associated with the compromise of that key is greater. Specifically, if all documents are encrypted with a single key, this would give the attacker a large advantage in the event that they can compromise that key. Thus, a recommended practice for security and privacy is to provide a per-document key, or even possibly a hierarchy of keys. This level of encryption provides better guarantees against common cloud risks, such as rogue administrators, complicit service providers, or hackers. One could argue that the combination of client-side document-based encryption paired with client-provided key-management offers a holistic, end-to-end approach to security and privacy.

The last factor (client- versus server-provided keys) might look like the first factor, that is, whether encryption is done by the server or by the client, however, there are subtle differences. For example, key-management itself is difficult, and having each client implement their own set of processes, tools, and overall integration is costly. Nevertheless, paired with client-side encryption it offers the highest level of security of all options. However, having server-provided key management might offer certain benefits (like ease of use or better integration) if the provider for key-management is different from the storage provider. It is worth noting that there are two options in this space; using a physical hosting environment for Hardware Security Modules (HSMs), or using a cloud key-management solution. The latter solution carries the same risks as the ones for data storage, however, with even greater repercussions.

1.3 Deduplication and cloud storage

Storage efficiency functions such as compression and deduplication afford storage providers better utilization of their storage backends and the ability to serve more customers with the same infrastructure. Compression is a process of reducing the size of the stored data at either the storage provider side and/or at the client side, by using several well-known algorithms. Storing the compressed data at the storage provider does not differ much from storing uncompressed data (apart from the used space and bandwidth). In the following text we focus on data deduplication, and it is to note that most of the comments on deduplication apply to compression as well.

Data deduplication is the process by which a storage provider only stores a single copy of a

file owned by several of its users[1]. There are four different deduplication strategies, depending on whether deduplication happens at the client side (i.e. before the upload) or at the server side, and whether deduplication happens at a block level or at a file level. Deduplication is most rewarding when it is triggered at the client side, as it saves upload bandwidth (in addition to being cost-effective, due to reduced storage costs). For these reasons, deduplication is a critical enabler for a number of popular and successful storage services (e.g. Dropbox, Memopal) that offer cheap, remote storage to the broad public by performing client-side deduplication, thus having both the network bandwidth and storage costs. Indeed, data deduplication is arguably one of the main reasons why the prices for cloud storage and cloud backup devices have dropped so sharply.

Unfortunately, deduplication loses its effectiveness in conjunction with end-to-end encryption. End-to-end encryption in a storage system is the process by which data is encrypted at its source prior to ingress into the storage system. It is becoming an increasingly prominent requirement due to both the number of security incidents linked to leakage of unencrypted data and the tightening of sector-specific laws and regulations. Clearly, if semantically secure encryption is used, file deduplication is impossible, as no one, apart from the owner of the decryption key can decide whether two ciphertexts correspond to the same plaintext. Trivial solutions, such as forcing users to share encryption keys or using deterministic encryption, fall short of providing acceptable levels of security. As a consequence, storage systems are expected to undergo major restructuring to maintain the current disk/customer ratio in the presence of end-to-end encryption.

The design of storage efficiency functions in general and of deduplication functions in particular that do not lose their effectiveness in the presence of end-to-end security is still an open problem.

Several deduplication schemes have been proposed by the research community [MB09] showing how deduplication allows very appealing reductions in the usage of storage resources [DF09, HMN+12]. Most works do not consider security as a concern for deduplicating systems; recently however, Harnik et al. [HPSP10] presented a number of attacks that can lead to data leakage in storage systems in which client-side deduplication is in place.

To thwart such attacks, the concept of proof of ownership has been introduced [HHPSP11, PS12]. None of these works, however, can provide real end-user confidentiality in presence of a malicious or honest-but-curious cloud provider. Convergent encryption is a cryptographic primitive introduced by Douceur et al. [DAB+02, SGLM08], attempting to combine data confidentiality with the possibility data deduplication. Convergent encryption of a message consists of encrypting the plaintext using a deterministic (symmetric) encryption scheme with a key which is deterministically derived solely from the plaintext. Clearly, when two users independently attempt to encrypt the same file, they will generate the same ciphertext which can be easily deduplicated. Unfortunately, convergent encryption does not provide semantic security as it is vulnerable to content-guessing attacks. Later, Bellare et al. [BKR12] formalized convergent encryption under the name message-locked encryption. As expected, the security analysis presented in [BKR12] highlights that message-locked encryption offers confidentiality for unpredictable messages only, clearly failing to achieve semantic security. Xu et al. [XCZ13] present a Proof of Ownership (PoW) scheme allowing client-side deduplication in a bounded leakage setting. They provide a security proof in a random oracle model for their solution, but do not address the problem of low min-entropy files.

[1] It is to note that this has no impact on reliability. We consider here the files at a "logical" level and assume that adequate reliability services will keep redundant copies of each object at the "physical" level, in order to offer availability and persistence.

Recently, Bellare et al. presented DupLESS [BKR13], a server-aided encryption for deduplicated storage, that uses a modified convergent encryption scheme with the aid of a secure component for key generation. Stanek et al [SSAK14] introduced the concept of file popularity as the distinct number of users who have uploaded a particular file, and designed an encryption scheme that guarantees semantic security for unpopular data and provides weaker security and better storage and bandwidth benefits for popular data.

2. Fragmentation techniques for data confidentiality

As illustrated in Chapter 1, encryption represents an effective solution for protecting data confidentiality when relying on external honest-but-curious (i.e., trusted to correctly manage the stored data but not to access their content) cloud providers for storage. An alternative solution, possibly to be used in conjunction with encryption, is *data fragmentation*. When it is the associations between pieces of data that is sensitive, rather than the individual data themselves, confidentiality can be provided by splitting the data into separate non-linkable fragments. In this chapter, we illustrate fragmentation-based approaches that can be adopted for protecting data confidentiality in the cloud.

2.1 Protection requirements and techniques

In this section, we first discuss how confidentiality requirements that need to be satisfied when moving the data to the cloud can be expressed by data owners (Section 2.1.1). We then illustrate the protection techniques that can be adopted for the enforcement of such confidentiality constraints (Section 2.1.2).

2.1.1 Confidentiality constraints

Protection requirements express what is sensitive and should be therefore kept confidential when storing data with external providers. For simplicity and concreteness, most existing proposals assume data to be organized as a relation r over relational schema $R(a_1,\ldots,a_n)$, where a_i, $i = 1,\ldots,n$, are the different attributes of the relation, with the note that the proposed protection techniques could be applied to different data models. Similarly, they assume protection requirements to be defined at the level of the relational schema, meaning over attributes, in contrast to specific attribute values. This assumption simplifies the management and the application of the protection techniques, ensuring the applicability of the solutions.

Operating at the level of schema, we can distinguish the following two kinds of confidentiality requirements that can apply to the data, corresponding to the fact that a given attribute is sensitive or that the association among some attributes is sensitive.

- *Sensitive attributes.* Attributes that are considered sensitive should have their values be kept confidential. Simple examples of such attributes are SSNs, credit card numbers, emails or telephone numbers and similar attributes whose values should not be disclosed.
- *Sensitive associations.* In some cases, what is sensitive is the association among attribute values, rather than the values of a single attribute. For instance, the names of patients in a

MEDICALDATA

SSN	Name	Race	Job	Disease	Treatment	Ins
123-45-6789	Alice	white	teacher	flu	paracetamol	160
234-56-7890	Bob	while	farmer	asthma	bronchodilators	100
345-67-8901	Carol	asian	nurse	gastritis	antacids	100
456-78-9012	David	black	lawyer	angina	nitroglycerin	200
567-89-0123	Eric	black	secretary	flu	aspirin	100
678-90-1234	Fred	asian	lawyer	diabetes	insulin	180
789-01-2345	Greg	white	clerk	stroke	nitroglycerin	150
890-12-3456	Hal	black	surgeon	broken leg	surgery	200

(a)

$\mathscr{C}$

c_1 = {SSN}
c_2 = {Name, Disease}
c_3 = {Name, Ins}
c_4 = {Disease, Ins}
c_5 = {Race, Job, Ins}

(b)

Figure 2.1: An example of a relation (a) and of confidentiality constraints over it (b)

hospital may be considered not sensitive, and so are the conditions treated by the hospital; however the specific association between individual patients and their illnesses is sensitive and should be maintained confidential.

A simple, yet conveniently expressive, way to capture the confidentiality requirements of sensitive attributes/associations is the specification of *confidentiality constraints* as set of attributes whose joint visibility should be avoided [ABG+05a]. Singleton sets (e.g., $\{a\}$) correspond to sensitive attributes; non-singleton sets (e.g., $\{a_i, \dots, a_j\}$) correspond to sensitive associations. A confidentiality constraint c over relation $R(a_1, \dots, a_n)$ is then defined as a subset of attributes in R (i.e., $c \subseteq \{a_1, \dots, a_n\}$).

As an example, consider relation MEDICALDATA in Figure 2.1(a), reporting the information about hospitalized patients. Figure 2.1(b) illustrates an example of a set of confidentiality constraints over it stating that:

- c_1: the Social Security Numbers of the patients are sensitive and should be maintained confidential (sensitive attribute);
- c_2, c_3: the associations between the name of patients and their condition, and between their names and the insurance they pay, are sensitive and should be maintained confidential (sensitive associations);
- c_4: the association between the disease of a patient and the medical insurance they pay is sensitive (sensitive association);
- c_5: the association among the race of a patient, their job, and the insurance they pay is confidential (sensitive association).

Note that the protection of a confidentiality constraint c_i implies the protection of any confidentiality constraint c_j such that $c_i \subset c_j$ (if observers do not have visibility of the attribute/association c_i they clearly do not have visibility of the association including it); making the consideration of c_j redundant. A set $\mathscr{C}$ of confidentiality constraints over R is *well-defined* if it does not include *redundant* constraints, that is, $\forall c_i, c_j \in \mathscr{C}$, $i \neq j$: $c_i \not\subset c_j$. The set of constraints in Figure 2.1(b) is well-defined.

2.1.2 Fragmentation and encryption

A natural protection technique that has been proposed for satisfying confidentiality requirements is represented by *fragmentation*, possibly coupled with *encryption* (see Chapter 1).

Essentially, fragmentation consists of splitting the attributes of a relation r producing different vertical views (fragments) in such a way that these views stored at external providers do not violate confidentiality constraints (neither directly nor indirectly). Intuitively, fragmentation protects the sensitive association represented by an association constraint c when the attributes in c do not appear all in the same (publicly available) fragment, and fragments cannot be joined by non authorized users. Note that singleton constraints are correctly enforced only when the corresponding attributes do not appear in any fragment that is stored at a cloud provider.

Depending on whether and how fragmentation is coupled with encryption, and how fragmentation itself operates on data, several fragmentation strategies have been proposed in the literature, as summarized in the following and illustrated in more detail in Section 2.2, Section 2.3, and Section 2.4.

- *Two can keep a secret*. This strategy assumes the availability of two independent and non-communicating providers, each storing a portion of the data. Whenever possible, sensitive associations are protected by partitioning the involved attributes among the two providers (in such a way that the two providers do not have complete visibility of all the attributes in a sensitive association). Sensitive attributes are always encrypted. Other attributes may be stored in encrypted form whenever storing them in the clear at any of the two fragments would violate at least one confidentiality constraint. The two fragments that have a key attribute in common, making them joinable by the owner and by authorized users, are the only parties who have access to both providers.

- *Multiple fragments*. This strategy employs encryption for protecting sensitive attributes and fragmentation for protecting sensitive associations. It does not make assumptions on the nature/number of providers and on the number of fragments. Employing an arbitrary number of fragments allows sensitive associations to always be satisfied with fragmentation. Fragments are complete (all attributes are stored in each fragment in either encrypted or plaintext form) and not linkable (they have no attribute in common). Since the fragments are unlinkable, there is no need to assume that the providers cannot communicate among them.

- *Keep a few*. This strategy assumes the involvement of a trusted party (typically the owner) for storing, and hence participating in the processing of, a limited amount of data. No encryption is applied. Sensitive attributes are stored at the owner side. Sensitive associations are protected by storing at least one of their attributes at the owner side (trying to minimize storage/computation required to the owner).

2.2 Two can keep a secret

We now present a solution based on encryption and fragmentation where data are split into two fragments, with each fragment stored at a different provider (see Figure 2.2, where encrypted attributes are denoted as gray columns). The two providers in this case are assumed to be non-communicating [ABG$^+$05b].

2.2.1 Fragmentation model

The satisfaction of confidentiality constraints is guaranteed by proper combination of vertical fragmentation and encryption, and relies on the assumption that the two cloud providers storing frag-

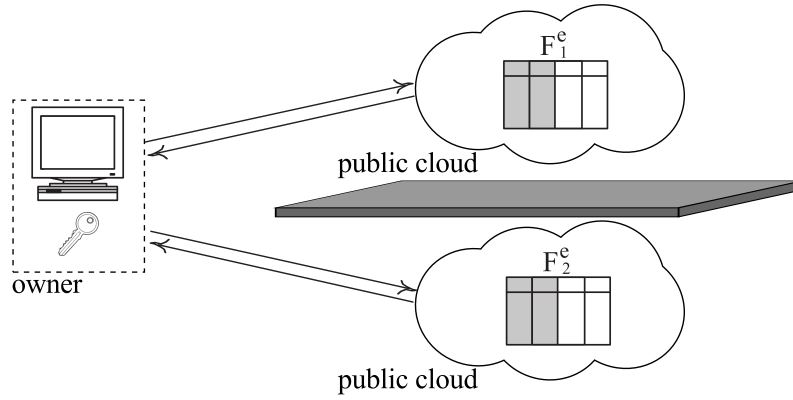


Figure 2.2: Two can keep a secret

ments do not communicate with each other (see Figure 2.2).

According to the proposal in [ABG$^+$05b], the original relation R is fragmented generating a fragmentation $\mathscr{F}$=$\{F_1$,F_2,$E\}$, where F_1 and F_2 are two fragments that are stored at two providers, and E is the set of encrypted attributes. Note that in the original proposal [ABG$^+$05b] encryption is considered as one of the techniques that can be used for encoding (i.e., obfuscating) attributes. Given an attribute a, the encoding of a consists of splitting its value into two (or more) attributes, say a^i and a^j, whose combined knowledge is necessary to reconstruct a (i.e., $a = a^i \otimes a^j$ with $\otimes$ a non-invertible composition operator). Encoding an attribute using encryption means therefore that a^i (a^j, resp.) contains the ciphertext, a^j (a^i, resp.) contains the encryption key, and $\otimes$ is the encryption function adopted by the data owner. Attribute a^i is stored at one provider and attribute a^j is stored at the other. For the sake of readability, in the remainder of this section we will consider encryption as the specific technique adopted to enforce encoding.

Singleton constraints are satisfied by encrypting sensitive attributes. Association constraints are satisfied by splitting the involved attributes between the two providers. Since relation r can be split in two fragments only, it may happen that an attribute cannot be stored at any of the two fragments without violating a confidentiality constraint. In this case, the confidentiality constraint can be satisfied by encrypting one (or more) of its attributes. A fragmentation $\mathscr{F}$=$\{F_1$,F_2,$E\}$ is *correct* if it satisfies all the confidentiality constraints defined by the data owner. Formally, a fragmentation $\mathscr{F} = \{F_1, F_2, E\}$ is *correct* iff:

- $\forall c \in \mathscr{C}$: $c \not\subseteq F_1$, $c \not\subseteq F_2$ (confidentiality),
- $F_1 \cup F_2 \cup E = R$ (completeness).

The first condition requires that neither F_1 nor F_2 store all the attributes in a confidentiality constraint in plaintext. Since the two fragments are stored at different providers, and these providers do not communicate with each other, sensitive associations as well as sensitive attribute values cannot be reconstructed by non authorized users. The second condition instead demands that the fragments store (either plaintext or encrypted) all the attributes in the original relation. This guarantees that the content of the original relation can always be reconstructed starting from $\mathscr{F}$. For instance, a correct fragmentation $\mathscr{F}$ of relation MEDICALDATA in Figure 2.1(a) with respect to the confidentiality constraints in Figure 2.1(b) is $\mathscr{F}$=$\{F_1$,F_2,$E\}$, with F_1={Name,Race,Job}, F_2={Disease,Treatment}, and E={SSN,Ins}.

At the physical level, fragments F_1 and F_2 are represented by *physical fragments* F^e_1 and F^e_2, respectively. Each physical fragment F^e_i stores the attributes in F_i in plaintext, and all the attributes

F_1^e

tid	Name	Race	Job	SSN[1]	Ins[1]
1	Alice	white	teacher	Enc(123-45-6789,$k_{\texttt{SSN}}$1)	Enc(150,$k_{\texttt{Ins}}$1)
2	Bob	while	farmer	Enc(234-56-7890,$k_{\texttt{SSN}}$2)	Enc(100,$k_{\texttt{Ins}}$2)
3	Carol	asian	nurse	Enc(345-67-8901,$k_{\texttt{SSN}}$3)	Enc(100,$k_{\texttt{Ins}}$3)
4	David	black	lawyer	Enc(456-78-9012,$k_{\texttt{SSN}}$4)	Enc(200,$k_{\texttt{Ins}}$4)
5	Eric	black	secretary	Enc(567-89-0123,$k_{\texttt{SSN}}$5)	Enc(100,$k_{\texttt{Ins}}$5)
6	Fred	asian	lawyer	Enc(678-90-1234,$k_{\texttt{SSN}}$6)	Enc(180,$k_{\texttt{Ins}}$6)
7	Greg	white	clerk	Enc(789-01-2345,$k_{\texttt{SSN}}$7)	Enc(150,$k_{\texttt{Ins}}$7)
8	Hal	black	surgeon	Enc(890-12-3456,$k_{\texttt{SSN}}$8)	Enc(200,$k_{\texttt{Ins}}$8)

F_2^e

tid	Disease	Treatment	SSN[2]	Ins[2]
1	flu	paracetamol	$k_{\texttt{SSN}}$1	$k_{\texttt{Ins}}$1
2	asthma	bronchodilators	$k_{\texttt{SSN}}$2	$k_{\texttt{Ins}}$2
3	gastritis	antacids	$k_{\texttt{SSN}}$3	$k_{\texttt{Ins}}$3
4	angina	nitroglycerin	$k_{\texttt{SSN}}$4	$k_{\texttt{Ins}}$4
5	flu	aspirin	$k_{\texttt{SSN}}$5	$k_{\texttt{Ins}}$5
6	diabetes	insulin	$k_{\texttt{SSN}}$6	$k_{\texttt{Ins}}$6
7	stroke	nitroglycerin	$k_{\texttt{SSN}}$7	$k_{\texttt{Ins}}$7
8	broken leg	surgery	$k_{\texttt{SSN}}$6	$k_{\texttt{Ins}}$8

Figure 2.3: An example of a correct fragmentation of relation MEDICALDATA in Figure 2.1(a) in the two can keep a secret scenario

in E encrypted. The two physical fragments representing relation r must have a common attribute, to allow authorized users to correctly reconstruct the content of r (lossless join property). Therefore, physical fragment F_i^e representing fragment F_i has schema $F_i^e(\underline{tid}, a_{i_1}, \ldots, a_{i_n}, a^i_{e_1}, \ldots, a^i_{e_m})$, where:

- tid is a randomly generated tuple identifier;
- $\{a_{i_1}, \ldots, a_{i_n}\}$ is the set of attributes composing fragment F_i;
- $\{a^i_{e_1}, \ldots, a^i_{e_m}\}$ is the set of attributes resulting from the encryption of the attributes in E= $\{a_{e_1}, \ldots, a_{e_m}\}$, that is, for each a_{e_i} in E, either $a^1_{e_i}$ represents encrypted attribute a_{e_i} and $a^2_{e_i}$ represents the corresponding encryption key, or viceversa.

Formally, each tuple t in r is represented by a tuple t_1^e in F_1^e and a tuple t_2^e in F_2^e, where: $t_1^e[tid]$=$t_2^e[tid]$ is a randomly generated value; $t_1^e[a]$=$t[a]$, $\forall a \in F_1$ and $t_2^e[a]$=$t[a]$, $\forall a \in F_2$; and attributes $t_1^e[a^1]$, $t_2[a^2]$ are the encrypted version and the encryption key of attribute $t[a]$, $\forall a \in E$ (i.e., $Enc(t[a], t_1^e[a^1]) = t_2^e[a^2]$ or $Enc(t[a], t_2^e[a^2]) = t_1^e[a^1]$).

Figure 2.3 illustrates the physical fragments representing fragmentation $\mathscr{F}$=$\{F_1, F_2, E\}$, with F_1=$\{$`Name`,`Race`,`Job`$\}$, F_2=$\{$`Disease`,`Treatment`$\}$, and E=$\{$`SSN`,`Ins`$\}$ of relation MEDICALDATA in Figure 2.1(a). In this example, for simplicity, we assume that F_1^e stores the encrypted attribute values and F_2^e stores the corresponding encryption keys for all the tuples in r and for both attributes `SSN` and `Ins`.

2.2.2 Fragmentation metrics

Given a relation schema R and a set $\mathscr{C}$ of confidentiality constraints over it, the data owner needs to compute a correct fragmentation. However, there may exist different correct fragmentations that satisfy all the constraints. As a simple example, fragmentation $\mathscr{F}$=$\{F_1, F_2, E\}$ with E=R and F_1=F_2=$\emptyset$ is clearly correct but undesirable, since no data appears in the clear in F_1 and F_2, thereby causing a high overheard due to the need to encrypt all attributes (nullifying the benefits

	SSN	Name	Race	Job	Disease	Treatment	Ins
SSN	10	20	22	18	17	25	10
Name		30	12	25	10	15	40
Race			20	18	32	40	10
Job				10	14	23	17
Disease					10	30	40
Treatment						20	40
Ins							15

Figure 2.4: An example of affinity matrix

of fragmentation) and also to the fact that no query can be evaluated by the providers storing F_1^e and F_2^e. In order to place as much of the computational effort on the cloud providers, it is then necessary to define a metric to measure the *quality* of a fragmentation in terms of the overhead required to users for evaluating their queries over the fragmentation $\mathscr{F}$. The metric proposed in [ABG+05b] is based on the knowledge of the query workload $\mathscr{Q}$ (i.e., a set of representative queries that are expected to be frequently executed) characterizing the system. In fact, the query workload describes how frequently attributes appear together in queries, which enables the estimation of the computational overhead that a fragmentation splitting these attributes may cause to users. To assess the quality of a fragmentation, the query workload is modeled as an *affinity matrix*, which is a symmetric matrix with a row and a column for each attribute in R, and where each cell $M[a_i,a_j]$=$M[a_j,a_i]$ ($i \neq j$), represents the cost (i.e., the computation overhead for users) of having attributes a_i and a_j stored in different fragments. Each cell $M[a,a]$ (i.e., cells along the diagonal) instead represents the cost of having attribute a encrypted. For instance, the affinity matrix in Figure 2.4 states that the cost of having attributes Name and Disease stored in two different fragments is $M[\texttt{Name},\texttt{Disease}] = 10$, and that of encrypting attribute Ins is $M[\texttt{Ins},\texttt{Ins}] = 15$. The cost of a fragmentation $\mathscr{F}$ is computed by summing the costs of the attributes encrypted in $\mathscr{F}$, and the costs of the pairs of attributes not stored together in a fragment in $\mathscr{F}$. Formally, the cost of a fragmentation $\mathscr{F}$={F_1,F_2,E} is defined as:

$$\sum_{a_i \in F_1, a_j \in F_2} M[a_i,a_j] + \sum_{a_i \in E} M[a_i,a_i]$$

As an example, consider relation MEDICALDATA in Figure 2.1(a), the fragmentation in Figure 2.3, and the affinity matrix in Figure 2.4 (since the matrix is symmetric, we report the values only for the cells in the upper half of the matrix). The quality of $\mathscr{F}$ is computed as: $M[\texttt{Name},\texttt{Disease}] + M[\texttt{Name},\texttt{Treatment}] + M[\texttt{Race},\texttt{Disease}] + M[\texttt{Race},\texttt{Treatment}] + M[\texttt{Job},\texttt{Disease}] + M[\texttt{Job},\texttt{Treatment}] + M[\texttt{SSN},\texttt{SSN}] + M[\texttt{Ins},\texttt{Ins}] = 10+15+32+40+14+23+10+15 = 159$.

2.2.3 Computing an optimal fragmentation

The problem of computing a fragmentation that minimizes the cost of query evaluation is NP-hard (the minimum hypergraph coloring problem reduces to it in polynomial time [ABG+05b]). Hence, in [ABG+05b] the authors propose to adopt an heuristic approach to compute a good, although non optimal, solution. The proposed solution is based on a graph modeling of the fragmentation problem, where each attribute in R is represented as a vertex in a *complete graph* G whose edges and vertices are weighted according to M (i.e., *weight*(a)=$M[a,a]$ and *weight*(a_i,a_j)=$M[a_i,a_j]$). The graph has an additional set H of hyperarcs, modeling the confidentiality constraints in $\mathscr{C}$. The proposed heuristic combines two approximation techniques, traditionally used to find a good solution to the following well known hard problems.

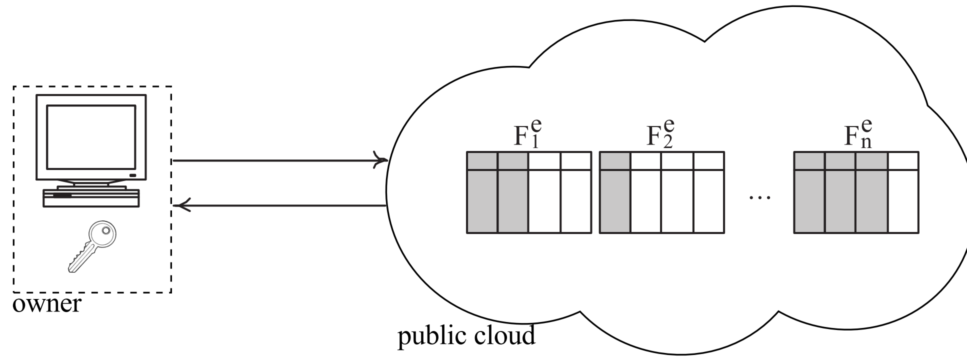


Figure 2.5: Multiple fragments

- *Min-Cut*. Assuming that $\mathscr{C}$ is empty, the problem of computing an optimal fragmentation can be translated into the problem of computing a minimum cut for G. A minimum cut for a graph G is a partitioning of the set of vertices in G in two subsets, V_1 and V_2, that minimizes the weight of the edges with one vertex in V_1 and one vertex in V_2. Intuitively, the heuristic approaches proposed for the Min-Cut problem can be used to compute different cuts that are nearly minimal, and then we can choose the one that satisfies the highest number of confidentiality constraints.
- *Weighted Set Cover*. If we do not consider the cost of splitting attributes between F_1 and F_2, the problem of computing a correct fragmentation can be translated into the minimum set cover problem. Intuitively, each confidentiality constraint is a set whose elements are the attributes composing it. The weight of each attribute a is $M[a,a]$ and the minimum set cover is the set C' of attributes with minimum weight that includes (at least) one attribute for each constraint. Confidentiality constraints can be satisfied by encrypting all the attributes in C'.

By combining these two approaches, it is possible to compute a good fragmentation in polynomial time. In fact, the Min-Cut heuristic algorithm guarantees to compute a good split of the attributes between F_1 and F_2, while the weighted set cover guarantees constraint satisfaction. The corresponding heuristic algorithm works as follows. First, it computes a minimum set cover E through a greedy strategy, to guarantee that all the constraints are satisfied by encrypting the attributes in E. Then, it computes a minimum cut for the attributes in $R \setminus E$, to split them between F_1 and F_2 minimizing the cost of the fragmentation. Finally, for each attribute a in E, it moves a to F_1 (F_2, respectively) if no confidentiality constraint is violated.

2.3 Multiple fragments

We present a solution based on encryption and fragmentation where data can be split among an arbitrary number of fragments [CDF+07, CDF+10], which may be possibly stored at the same provider (Figure 2.5).

2.3.1 Fragmentation model

The goal of the proposal in [CDF+07] is to remove the limiting assumption of absence of collusion between the two providers characterizing the solution in [ABG+05a]. The idea is therefore to compute a fragmentation (with no limits on the number of fragments composing it) in such a way

that its fragments are not linkable, meaning that it is not possible for parties different from the data owner and authorized users to reconstruct the original relation and then also the sensitive values and associations. Being non linkable, fragments can be stored at the same provider with no confidentiality risk.

The approach in [CDF+07] couples vertical fragmentation with encryption to satisfy confidentiality constraints. In particular, each singleton constraint c={a} is satisfied by encrypting the involved attribute a. Each association constraint $c=\{a_1,\ldots,a_n\}$ can instead be satisfied by either encrypting at least one among $a_1,\ldots,a_n$, or by storing these attributes in different fragments. To prevent indirect violation of confidentiality constraints by joining fragments, fragments must be disjointed (i.e., no attribute can appear in more than one fragment). More formally, a fragmentation $\mathscr{F} = \{F_1,\ldots,F_m\}$ is *correct* iff:

- $\forall c \in \mathscr{C}, \forall F \in \mathscr{F}$: $c \not\subseteq F$ (confidentiality);
- $\forall F_i,F_j \in \mathscr{F}, i \neq j$: $F_i \cap F_j = \emptyset$ (unlinkability).

The first condition states that a fragment in $\mathscr{F}$ cannot contain all the attributes composing a confidentiality constraint. The second condition states that fragments must be disjointed. This approach has two advantages: *i)* being disjointed, all fragments $F_1,\ldots,F_n$ composing a fragmentation $\mathscr{F}$ are not linkable and can therefore be stored at the same provider; and *ii)* not imposing any limit on the number of fragments, association constraints can always be satisfied without encryption, thus increasing the *visibility* over data, with clear advantages for query evaluation. In fact, the plaintext representation of an attribute a in a fragment F permits the evaluation of conditions over a at the cloud provider storing F. For this reason, the approach in [CDF+07] aims at computing fragmentations that maximize visibility. A fragmentation maximizes visibility if each attribute a in R not appearing in a singleton constraint is plaintext represented in *at least* one fragment. Note however that, to satisfy the unlinkability condition, each attribute not appearing in a singleton constraint can belong to *at most* one fragment in a correct fragmentation. For instance, $\mathscr{F}$={{Name,Job}, {Disease,Treatment}, {Race,Ins}} is a correct fragmentation of relation MEDICALDATA in Figure 2.1(a) with respect to the constraints in Figure 2.1(b). This fragmentation maximizes visibility as all the attributes but SSN, which is sensitive per se (c_1), are plaintext represented in exactly one fragment.

At a physical level, each fragment $F_i = \{a_{i_1},\ldots,a_{i_n}\}$ of a fragmentation $\mathscr{F}$ is represented by a *physical fragment* $F_i^e(\underline{salt},enc,a_{i_1},\ldots,a_{i_n})$, where:

- *salt* is the primary key of the relation and contains a randomly chosen value;
- *enc* is an attribute storing the encrypted attributes in $R \setminus \{a_{i_1},\ldots,a_{i_n}\}$;
- $\{a_{i_1},\ldots,a_{i_n}\}$ is the set of attributes composing fragment F_i.

Each tuple t in r is represented by a tuple in each of the physical fragments $\{F_1^e,\ldots,F_m^e\}$ corresponding to the fragments $\{F_1,\ldots,F_m\}$ in $\mathscr{F}$. Tuple t^e representing t in F_i^e is such that: $t^e[salt]$ is a random value; $t^e[enc]$ is computed as $Enc(t[R \setminus F_i] \oplus t[salt],k)$, with $\oplus$ the binary XOR operator; and $t^e[a]$=$t[a]$, $\forall a \in F$. Note that the attributes not appearing in plaintext in F^e are combined with a random salt before encryption to prevent frequency attacks [Sch96]. Since each physical fragment stores (either plaintext or encrypted) all the attributes in R, every query can be evaluated on a single fragment. Figure 2.6 illustrates the physical fragments representing fragmentation $\mathscr{F}$={{Name,Job}, {Disease,Treatment}, {Race,Ins}} of relation MEDICALDATA in Figure 2.1(a).

F_1^e

salt	enc	Name	Job
s_1^1	xTb:	Alice	teacher
s_1^2	o;!G	Bob	farmer
s_1^3	Ap'L	Carol	nurse
s_1^4	.u7t	David	lawyer
s_1^5	y"e3	Eric	secretary
s_1^6	(l1!	Fred	lawyer
s_1^7	K"f 4	Greg	clerk
s_1^8	#p8r	Hal	surgeon

F_2^e

salt	enc	Disease	Treatment
s_2^1	hg5=	flu	paracetamol
s_2^2	mB71	asthma	bronchodilators
s_2^3	:k?2	gastritis	antacids
s_2^4	Ql4,	angina	nitroglycerin
s_2^5	-kGd	flu	aspirin
s_2^6	p[Mz	diabetes	insulin
s_2^7	Ji(6	stroke	nitroglycerin
s_2^8	l&r1	broken leg	surgery

F_3^e

salt	enc	Race	Ins
s_3^1	b%P5	white	160
s_3^2	&C*x	white	100
s_3^3	1Bny	asian	100
s_3^4	Oj)6	black	200
s_3^5	vT7/	black	100
s_3^6	l1fY	asian	180
s_3^7	f%r"	white	150
s_3^8	Lp3(	black	200

Figure 2.6: An example of correct fragmentation of relation MEDICALDATA in Figure 2.1(a) in the multiple fragments scenario

Metric	Quality function
Number of fragments	$card(\mathscr{F})$
Affinity	$\sum_{k=1}^{n} aff(F_k)$ where $aff(F_k) = \sum_{a_i,a_j \in F_k, i<j} M[a_i,a_j]$, $k = 1,\ldots,n$
Query evaluation cost	$\sum_{i=1}^{m} freq(q_i) \cdot cost(q_i,\mathscr{F})$ where $cost(q_i,\mathscr{F})=Min(cost(q_i,F_j)$, $j = 1,\ldots,n)$ and $cost(q_i,F_j)=S(q_i,F_j) \cdot \lvert r \rvert \cdot size(t_j)$, $i = 1,\ldots,m$ and $j = 1,\ldots,n$

Table 2.1: Classification of the metrics in the multiple fragments scenario

2.3.2 Fragmentation metrics

Given a relation schema R and a set $\mathscr{C}$ of confidentiality constraints over it, there may be different correct fragmentations that maximize visibility. As an example, a fragmentation $\mathscr{F}$ where each attribute in R that does not appear in a singleton constraint is stored in a different fragment is correct and maximizes visibility. However, this solution causes an excessive fragmentation that should be avoided. We now present different metrics (see Table 2.1) that can be used to assess the quality of a fragmentation in terms of the query evaluation overhead caused to users.

- *Minimal fragmentation* (e.g., [CDF+07]). A simple metric for evaluating the quality of a fragmentation consists of minimizing the number of fragments thus avoiding excessive fragmentation. Intuitively, a fragmentation with a lower number of fragments is likely to store more attributes in the same fragment, clearly favoring the evaluation of queries that involve these attributes (also together). For instance, both $\mathscr{F}$={{`Name`,`Job`}, {`Disease`,`Treatment`}, {`Race`,`Ins`}} and $\mathscr{F}'$={{`Name`,`Job`}, {`Disease`},{`Treatment`}, {`Race`,`Ins`}} are correct fragmentations of relation MEDICALDATA in Figure 2.1(a). However, $\mathscr{F}$ is preferable to $\mathscr{F}'$ because it efficiently supports the evaluation of queries involving, both `Disease` and `Treatment`. Two different notions of minimality have been proposed: *minimality* (i.e., composed of the minimum number of fragments [BBL12, CDF+12]) and *local minimality* (i.e., composed of fragments that cannot be merged without violating constraints [CDF+07, CDF+12]). We note that, while a locally minimal fragmentation might not be composed of the minimum number of fragments, a minimal fragmentation is indeed also locally minimal (i.e., merging any of its fragments violates at least a constraint).

	Name	Race	Job	Disease	Treatment	Ins
Name		10	30	10	10	10
Race			10	10	10	40
Job				10	10	10
Disease					25	10
Treatment						10
Ins						

Figure 2.7: An example of affinity matrix in the multiple fragments scenario

- *Maximum affinity* (e.g., [CDF+10]). A more precise assessment of the quality of a fragmentation is based on the *affinity* between attributes. The affinity between two attributes quantifies the performance advantage in query evaluation that can be obtained by storing them in the same fragment [OV99]. Attributes with high affinity are expected to be frequently involved together in queries. Therefore, the higher the affinity, the higher the advantage in query evaluation of having the attributes stored in the same fragment. Attribute affinity can be modeled by an *affinity matrix* M, which is a symmetric matrix with a row and a column for each attribute that do not appear in a singleton constraint. Each cell $M[a_i,a_j]$, with $i \neq j$, represents the benefit obtained by storing attributes a_i and a_j in the same fragment. For instance, Figure 2.7 illustrates an example of affinity matrix for relation MEDICALDATA in Figure 2.1(a). Fragmentations that keep in the same fragment attributes with high affinity are to be preferred over fragmentations that split them in different fragments. The quality of a fragmentation $\mathscr{F}$ is measured as the sum of the affinity of the fragments composing it, where the affinity of a fragment F is obtained by summing the affinities of the pairs of attributes in F. As an example, consider relation MEDICALDATA in Figure 2.1(a), the fragmentation in Figure 2.6, and the affinity matrix in Figure 2.7. The quality of $\mathscr{F}$ is computed as: $M[\texttt{Name},\texttt{Job}] + M[\texttt{Disease},\texttt{Treatment}] + M[\texttt{Race},\texttt{Ins}] = 30+25+40 = 95$.

- *Minimum query evaluation cost* (e.g., [CDF+09a]). Another possible metric is based on the definition of a query cost model, which can be used to evaluate the cost of executing a representative set of queries over fragments. This metric, compared with the affinity metric, has the advantage of taking into consideration also the benefit of storing in the same fragment arbitrary sets of attributes (instead of pairs thereof). The adoption of this metric requires the availability of the query workload $\mathscr{Q}$ of the system, which is a set $\{q_1,\ldots,q_m\}$ of queries along with their execution frequency $freq(q_i)$, $i = 1,\ldots,m$. The proposal in [CDF+09a] assumes that queries in $\mathscr{Q}$ are of the form "SELECT $a_{i_1},\ldots,a_{i_n}$ FROM R WHERE $\bigwedge_{j=1}^{n}$ (a_j IN V_j)" with V_j a set of values in the domain of attribute a_j. The quality of a fragmentation $\mathscr{F}$ then depends on the cost of executing the queries in $\mathscr{Q}$, properly weighted by their frequency, over the fragments in $\mathscr{F}$. Since each physical fragment stores, either plaintext or encrypted, all the attributes in R, the cost of evaluating a query q over $\mathscr{F}$ is the minimum among the costs of evaluating q over each physical fragment F^e in $\mathscr{F}$. The cost of evaluating q on F^e is estimated by the size of the result returned to the user, because the costs of communication, decryption, and evaluation of conditions on encrypted attributes at the user side are more expensive than the computational costs at the provider side. Hence, the cost $cost(q_i,F_j)$ of executing q_i on F_j is computed as $S(q_i,F_j) \cdot |r| \cdot size(t_j)$, where $S(q_i,F_j)$ is the selectivity of query q_i, $|r|$ is the number of tuples in r, and $size(t_j)$ is the size of the attributes appearing in the SELECT clause of q_i and the size of attribute *enc* if there is the need of accessing attributes not appearing in plaintext in F_j. The selectivity of condition a IN $V = \{v_1,\ldots,v_n\}$ is an estimate

of the ratio of the number of tuples in F that satisfy the condition over the total number of tuples in r. If attribute a does not appear in plaintext in F, the selectivity is set to 1. Consider, as an example, a query workload composed of two queries: q_1="SELECT * FROM MedicalData WHERE Job='teacher' AND Race='asian'", with frequency 30; and q_2="SELECT* FROM MedicalData WHERE Job='lawyer' AND Disease='flu'", with frequency 70. The fragmentation in Figure 2.6 implies a query evaluation cost $cost(\mathcal{Q},\mathcal{F}) = cost(q_1,\mathcal{F}) \cdot freq(q_1) + cost(q_2,\mathcal{F}) \cdot freq(q_2) = 1/8 \cdot 8 \cdot 1 \cdot 30 + 1/4 \cdot 8 \cdot 1 \cdot 70 = 170$. In fact, assuming that the size of the tuples is the same for all the fragments and is equal to 1, the fragment that minimizes query evaluation cost for q_1 is F_1, and are both F_1 and F_2 for q_2. Indeed, the most selective condition in q_1 operates on attribute Job, while the two conditions in q_2 are equally selective.

2.3.3 Computing an optimal fragmentation

Regardless of the metric adopted to evaluate the quality of a fragmentation, the problem of computing an optimal fragmentation is NP-hard (the minimum hypergraph coloring problem reduces to it in polynomial time [CDF+10]). Hence, the time complexity of any algorithm able to compute an optimal fragmentation is exponential in the number of attributes in R. In the following, we briefly survey exact and heuristic algorithms proposed for efficiently computing a correct and optimal (according to a chosen metric) fragmentation.

- *Minimal fragmentation* (e.g., [BBL12, CDF+07, CDF+12]). Both exact and heuristic algorithms have been proposed with the aim of avoiding an excessive fragmentation and producing minimal or locally minimal fragmentations. The exact algorithms in [BBL12, CDF+12], proposed to produce a minimal fragmentation, rely on a logical modeling of the problem. The attributes in R are interpreted as Boolean variables, and each confidentiality constraint c=$\{a_1,\ldots,a_n\}$ in $\mathcal{C}$ as a Boolean formula representing the conjunction $a_1 \wedge \ldots \wedge a_n$ of the attributes composing it. A fragment F of R is a truth assignment that assigns *true* to the variables representing the attributes in the fragment, and *false* to the other variables. A fragmentation $\mathcal{F}$ is a set of truth assignments that satisfy all the constraints in $\mathcal{C}$, and such that each variable a is assigned *true* in at most one fragment F in $\mathcal{F}$. Two approaches have been studied to compute a set of truth assignments representing a correct fragmentation that use a SAT (SATisfiability) and an OBDD (Ordered Binary Decision Diagram) formulation of the fragmentation problem. The adoption of SAT solvers has been proposed in [BBL12] to compute a fragmentation composed of the minimum the number of fragments. To achieve this, a SAT solver able to compute a correct fragmentation composed of n fragments is iteratively invoked. At the first iteration, n is set to 1. It is then incremented by 1 at each iteration. The iteration stops when the SAT solver finds a correct fragmentation. The adoption of the OBDD data structure to represent confidentiality constraints and efficiently compute fragments (i.e., truth assignments) satisfying constraints has been proposed in [CDF+12]. The problem of computing a fragmentation composed of the minimum number of fragments is translated into the problem of computing a maximum weighted clique over a *fragmentation graph*. The fragmentation graph models fragments, efficiently computed using OBDDs, that satisfy all the confidentiality constraints and a subset of the visibility constraints (i.e., required views over the data) defined in the system. Another heuristic approach for computing a locally minimal fragmentation has been proposed in [CDF+07]. The algorithm starts from an empty fragmentation $\mathcal{F}$ and tries to insert each attribute a in R (not involved in a

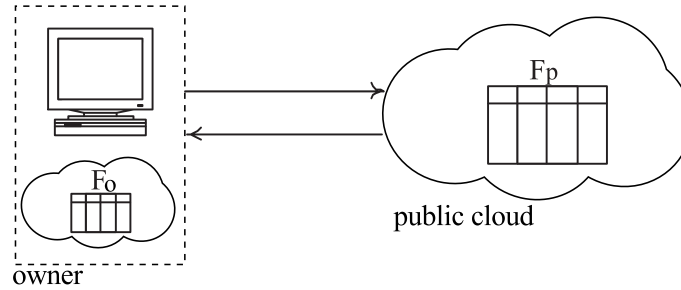


Figure 2.8: Keep a few

singleton constraint) into a fragment $F \in \mathscr{F}$. If a cannot be inserted into any fragment in $\mathscr{F}$ without violating constraints, a new fragment $F'=\{a\}$ is created and inserted into $\mathscr{F}$. The attributes are considered in decreasing order of the number of constraints in which they are involved (i.e., attributes appearing in a higher number of constraints are considered first).

- *Maximum affinity* (e.g., [CDF+10]). The greedy approach proposed in [CDF+10] takes advantage of the affinity matrix M previously illustrated in this section to compute a fragmentation that maximizes affinity. The proposed technique starts with a fragmentation $\mathscr{F}$ where each attribute a that does not appear in a singleton constraint belongs to a different fragment $F \in \mathscr{F}$. At each iteration, the algorithm merges the pair of fragments $\langle F_i, F_j \rangle$ with highest affinity according to M, provided no constraint is violated. The algorithm terminates when no further merge is possible.
- *Minimum query evaluation cost* (e.g., [CDF+09a]). The exact algorithm in [CDF+09a] for minimizing the cost of query execution is based on an efficient visit of the solution space of the fragmentation problem, which is represented through a lattice $(S_F, \preceq)$. Set S_F includes all the fragmentations of relation R composed of disjointed fragments; $\preceq$ is a dominance relationship between fragmentations where $\mathscr{F} \preceq \mathscr{F}'$ iff $\mathscr{F}$ can be obtained by merging fragments in $\mathscr{F}'$. The visit of the fragmentation lattice is based on two nice properties of the dominance relationship: *i)* given a non-correct fragmentation $\mathscr{F}'$, any fragmentation $\mathscr{F}$ such that $\mathscr{F} \preceq \mathscr{F}'$ is not correct; and *ii)* given two fragmentations $\mathscr{F}$ and $\mathscr{F}'$ such that $\mathscr{F} \preceq \mathscr{F}'$, the query evaluation cost of $\mathscr{F}'$ is higher than the cost of $\mathscr{F}$ (i.e., the cost is monotonic with the dominance relationship). A heuristic algorithm exploiting the fragmentation lattice has also been proposed [CDF+09a].

2.4 Keep a few

We present a solution completely departing from encryption where a trusted party (the owner) is involved in storing a portion of the data (Figure 2.8). Specifically, data are split into two fragments, one stored at the data owner side, and one stored at a cloud provider so that the fragment managed by the provider does not violate the confidentiality constraints [BPW11, CDF+09b, CDF+11].

2.4.1 Fragmentation model

Sensitive associations are protected by the approaches discussed in previous sections by encrypting (a portion of) the original relation and/or by splitting its content into non-linkable fragments. The approach in [CDF+09b] departs from encryption, and protects sensitive associations relying on

F^e_o

tid	SSN	Name	Ins
1	123-45-6789	Alice	160
2	234-56-7890	Bob	100
3	345-67-8901	Carol	100
4	456-7 8-9012	David	200
5	567-89-0123	Eric	100
6	678-90-1234	Fred	180
7	789-01-2345	Greg	150
8	890-12-3456	Hal	200

F^e_p

tid	Race	Job	Disease	Treatment
1	white	teacher	flu	paracetamol
2	while	farmer	asthma	bronchodilators
3	asian	nurse	gastritis	antacids
4	black	lawyer	angina	nitroglycerin
5	black	secretary	flu	aspirin
6	asian	lawyer	diabetes	insulin
7	white	clerk	stroke	nitroglycerin
8	black	surgeon	broken leg	surgery

Figure 2.9: An example of a correct fragmentation of relation MEDICALDATA in Figure 2.1(a) in the keep a few scenario

owner-side storage to satisfy confidentiality constraints. According to this proposal, relation R is fragmented generating a pair $\mathscr{F} = \langle F_o, F_p \rangle$ of fragments, with F_o stored at the data owner and F_p stored at a cloud provider. This solution satisfies singleton constraints c=$\{a\}$ by storing a at the owner. Similarly, it satisfies association constraints c=$\{a_1,\ldots,a_n\}$ by storing at least one among $\{a_1,\ldots,a_n\}$ at the owner. Formally, a fragmentation $\mathscr{F} = \langle F_o, F_p \rangle$ is *correct* iff:

- $\forall c \in \mathscr{C}, c \not\subseteq F_p$ (confidentiality);
- $F_o \cup F_p = R$ (losslessness).

The first condition states that fragment F_p cannot contain all the attributes composing a confidentiality constraint. This condition must hold only for F_p, since F_o is stored at the data owner and is therefore accessible to authorized users only. The second condition demands that all attributes in R are represented at the data owner or at the cloud provider, thus guaranteeing losslessness of the fragmentation. Although, in principle, F_o might include attributes appearing in F_p, this redundancy is not necessary and might be expensive for the data owner (both in terms of storage and computation). Fragments are then required to be disjointed (i.e., $F_o \cap F_p = \emptyset$). For instance, $\mathscr{F} = \langle F_o, F_p \rangle$ with F_o=`{SSN,Name,Ins}` and F_p=`{Race,Job,Disease,Treatment}` represents a correct fragmentation of relation MEDICALDATA in Figure 2.1(a) with respect to the confidentiality constraints in Figure 2.1(b).

At the physical level, fragments F_o and F_p must have a common key attribute to permit authorized users to correctly reconstruct the content of relation r. This attribute can be either the primary key of relation R, if it is not sensitive, or an attribute that does not belong to the schema of R and that is added to both F_o and F_p to this purpose. Assuming that the primary key of R cannot be publicly released, a fragmentation $\mathscr{F} = \langle F_o, F_p \rangle$, with $F_o = \{a_{o_1},\ldots,a_{o_i}\}$ and $F_p = \{a_{p_1},\ldots,a_{p_j}\}$, is translated into physical fragments $F^e_o(\underline{tid}, a_{o_1},\ldots,a_{o_i})$ and $F^e_p(\underline{tid}, a_{p_1},\ldots,a_{p_j})$, where *tid* is a randomly generated tuple identifier. Figure 2.9 illustrates the physical fragments representing fragmentation $\mathscr{F} = \langle F_o, F_p \rangle$ with F_o=`{SSN,Name,Ins}` and F_p=`{Race,Job,Disease,Treatment}` of relation MEDICALDATA in Figure 2.1(a). Note that at least one attribute of each constraint in Figure 2.1(b) is in F_o.

2.4.2 Fragmentation metrics

Given a relation schema R and a set $\mathscr{C}$ of confidentiality constraints over it, there may exist different correct fragmentations that are non-redundant. For instance, consider a fragmentation

Attribute a	*size*(a)
SSN	10
Name	20
Race	5
Job	18
Disease	18
Treatment	30
Ins	8

(a)

Query q	*freq*(q)	*Attr*(q)	$Cond_q$
q_1	20	Job, Disease	⟨Job⟩, ⟨Disease⟩
q_2	30	Disease, Treatment	⟨Disease⟩, ⟨Treatment⟩
q_3	40	Job, Ins	⟨Job⟩, ⟨Ins⟩
q_4	10	SSN, Ins, Disease	⟨SSN⟩, ⟨Ins⟩, ⟨Disease⟩

(b)

Figure 2.10: An example of size of attributes (a) and query workload (b) for relation MEDICALDATA in Figure 2.1(a)

$\mathcal{F} = \langle F_o, F_p \rangle$ where $F_o = R$ and $F_p = \emptyset$. This fragmentation is correct and non-redundant, but it does not take advantage of outsourcing as no storage and/or computation is delegated to the cloud provider. To maximize the advantages for the data owner, they must push as much of the storage and compute workloads as possible to the cloud provider for the management of their data. To achieve this, it is necessary to properly measure the storage, computation, and communication overhead caused to the data owner by the storage and management of fragment F_o. In the following, we illustrate the metrics that can be adopted to assess the quality of a fragmentation, depending on the resource that the data owner values more and on the information available about the system workload at initialization time [CDF+09b].

- *Minimal fragmentation.* The most straightforward metric consists of counting the number of attributes in F_o. Intuitively, a fragment composed of a lower number of attributes is likely to be smaller (reducing the storage occupation), and to be involved in a lower number of queries (reducing the computation and communication overhead).

- *Minimal size of attributes.* If the data owner wants to limit the storage occupation at the provider side, the most effective metric to assess the quality of a fragmentation measures the size of F_o. The storage occupation of F_o is computed as the sum of the size of all the attributes composing it. For instance, suppose that the size of the attributes of relation MEDICALDATA in Figure 2.1(a) is as summarized in Figure 2.10(a). The size of fragment F_o in Figure 2.9 is *size*(SSN)+*size*(Name)+*size*(Ins) $= 10+20+8 = 38$.

- *Minimal number of queries.* The computation and communication overhead at the data owner side can be measured as the number of queries whose evaluation requires the owner's intervention (i.e., queries involving at least one attribute in F_o). The adoption of this metric requires the knowledge of the query workload $\mathcal{Q}$ characterizing the system that, in this scenario, is a set $\{q_1, \ldots, q_m\}$ of representative queries, along with their frequency *freq*(q_i), $i = 1, \ldots, m$. For instance, the first three columns in Figure 2.10(b) represent a query workload for relation MEDICALDATA in Figure 2.1(a). The cost of a fragmentation $\mathcal{F} = \langle F_o, F_p \rangle$ is computed as the sum of the frequencies of the queries including at least one attribute in F_o. With respect to the workload in Figure 2.10(b), the fragmentation in Figure 2.9 requires the evaluation of *freq*(q_3)+*freq*$(q_4) = 50$ queries at the data owner, since q_3 and q_4 involve attributes in F_o.

- *Minimal number of conditions.* An alternative metric measuring the computation overhead of the data owner considers, instead of the number of queries, the number of conditions

	Metric	Quality function
Storage	Number of attributes	$card(F_o)$
	Size of attributes	$\sum_{a \in F_o} size(a)$
Computation and communication	Number of queries	$\sum_{q \in \mathscr{Q}} freq(q)$ s.t. $Attr(q) \cap F_o \neq \emptyset$
	Number of conditions	$\sum_{cond \in Cond(\mathscr{Q})} freq(cond)$ s.t. $cond \cap F_o \neq \emptyset$

Table 2.2: Classification of the metrics in the keep a few scenario

he/she should evaluate. In fact, the presence of multiple conditions in the same query operating on F_o causes a higher computation overhead for the data owner. To adopt this metric, it is necessary to know, besides the frequency $freq(q_i)$ of each query q_i in the query workload $\mathscr{Q}$, also the conditions, denoted $Cond(q_i)$, composing it. The quality of $\mathscr{F} = \langle F_o, F_p \rangle$ is then computed as the sum of the frequencies of the conditions in $\mathscr{Q}$ involving attributes in F_o. For instance, the first, second, and fourth column of Figure 2.10(c) represent a possible workload profile for relation MEDICALDATA in Figure 2.1(a). With respect to this workload, the fragmentation in Figure 2.9 requires the evaluation of $freq(\langle \texttt{SSN} \rangle)+freq(\langle \texttt{Ins} \rangle) = 10 + (40 + 10) = 60$ conditions at the data owner side.

Table 2.2 summarizes the formal definition of the metrics illustrated above. Note that the adoption of each metric is subject to the knowledge of different information about relation r and the query workload expected for the system.

2.4.3 Computing an optimal fragmentation

The problem of computing an optimal fragmentation that minimizes the storage or the computation and communication costs for the data owner is NP-hard (the minimum hitting set problem reduces to it in polynomial time [CDF+09b]). The heuristic approach proposed to compute a good fragmentation is based on the nice property that all the metrics illustrated above are monotonic in the number of attributes in F_o (i.e., the cost of a fragmentation $\mathscr{F}$ increases when an attribute is moved from F_p to F_o). Hence, the same heuristics applies to all the four metrics in Table 2.2. The algorithm proposed in [CDF+09b] aims at computing a *locally minimal fragmentation* $\mathscr{F} = \langle F_o, F_p \rangle$, which is defined as a fragmentation where no attribute can be moved from F_o to F_p without violating confidentiality constraints. The algorithm first inserts into F_o all the attributes that are considered sensitive per se (i.e., the attributes involved in singleton constraints). The remaining attributes, which initially belong to F_p, are organized in a priority queue. The priority of an attribute a depends on: *i)* the number of constraints that would be solved moving a to F_o, and *ii)* the cost that the data owner would pay to move a to F_o. The algorithm iteratively extracts from the queue the attribute a with highest priority (i.e., the attribute with minimum cost per solved constraint) and inserts it into F_o. The iteration stops when either all the constraints are satisfied or the queue is empty (i.e., F_o=R and F_p=$\emptyset$). The algorithm finally tries to move each attribute in F_o to F_p, to guarantee minimality of the computed fragmentation.

3. Techniques for data integrity

The techniques for ensuring data confidentiality illustrated in the previous chapters assume cloud providers to be trustworthy in managing the data they store. However, the rich cloud market features the availability of providers that might not be considered fully trusted for correctly managing the outsourced data. In this case, besides data confidentiality, data integrity and availability are two additional critical elements that should be guaranteed, meaning that neither the cloud provider, nor unauthorized parties, can improperly tamper with data in storage without being detected. In this chapter, we illustrate some of the most well-known solutions aiming at guaranteeing integrity and availability of data stored in the cloud.

3.1 Deterministic approaches

The first class of solution we will examine for ensuring data integrity relies on *deterministic* strategies, which have the advantage of providing integrity evidences with full confidence. These solutions are typically based on the adoption of digital signatures and of authenticated data structures, as illustrated in the remainder of this section.

3.1.1 Digital signatures

Digital signatures represent a traditional solution for guaranteeing data integrity (e.g., [HIM03]). Each data owner has its own public key pair. Each tuple is signed with the private key of its owner, and the signature is concatenated to the actual tuple. This concatenated chunk is then encrypted and sent to the cloud provider for storage. Unauthorized modifications to a tuple can be immediately detected by checking the signature associated with it. This basic approach, while effective, has the disadvantage that the cost associated with integrity verification linearly grows with the number of (accessed) tuples.

To limit the burden of signature verification, multiple digital signatures (related to multiple tuples) can be combined in a single signature by adopting *condensed RSA*, *BGLS*, or batch *DSA signature aggregation* [MNT06]. Condensed RSA is an extension of the traditional RSA encryption scheme that permits the aggregation of signatures generated by the same signer. BGLS [BGLS03] is an encryption scheme based on bilinear mappings that supports the aggregation of signatures even when they have been generated by different signers. Batch DSA is an extension of traditional DSA signature schema that permits the aggregation of the signatures of different tuples, which can be verified as a whole. The verification of a batch DSA signature aggregation is based on the multiplicative homomorphic property of these signatures. The signature verification processes for condensed RSA and BGLS schemas are more efficient than the verification process of batch DSA. However, both condensed RSA and BGLS are *mutable*, meaning that the knowledge of multiple aggregated signatures allows to compute their composition, thus obtaining a valid signature that

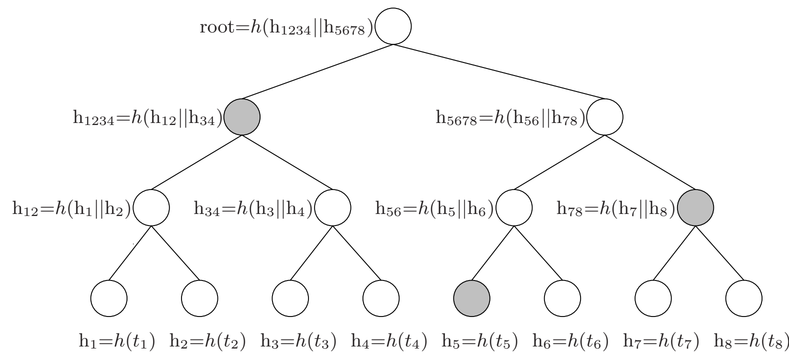


Figure 3.1: A Merkle hash tree over attribute `Name` of relation MEDICALDATA in Figure 2.1

may correspond to the aggregate signature of an arbitrary set of tuples. This might represent a threat to the integrity guarantees of the data.

3.1.2 Authenticated data structures

This family of solutions guarantees data integrity by building an *authenticated data structure* on the outsourced relation or data [MND+04, Tam03]. Since authenticated data structures are defined over the whole data collection, they provide integrity of data in storage (as well as of query results over them). Unauthorized modifications can be immediately detected when checking the integrity of a subset of tuples in the outsourced relation, such as the result of a query evaluation (e.g., [DGMS00, LHKR06, Mer89, PJRT05, PT04, YPPK09]). Essentially, a *verification object* (VO) extracted from the authenticated data structure is returned with any accessed subset of tuples, and can then be used by the client to verify data integrity. If the VO is consistent with the data structure, this guarantees that the set of tuples is correct and complete, and that the outsourced relation has not been improperly modified.

Merkle hash trees. Integrity of an outsourced relation (as well as of query computations over it) can be provided relying on a Merkle hash tree built over the relation [Mer89, PJRT05]. Given a relation r, a Merkle hash tree is a binary tree that stores, in each leaf, the result of a one-way hash function h over a tuple of r, and in each internal node the hash of the concatenation of its children. The tuples in the leaves of the tree are ordered according to the values of an attribute a, and the root of the Merkle hash tree is signed by the data owner. Figure 3.1 illustrates an example of a Merkle hash tree defined over attribute `Name` of relation MEDICALDATA in Figure 2.1(a). Given a set of tuples in r with contiguous values for a (e.g., resulting from the evaluation of a query over a), the server returns to the data owner also a VO with the values of the nodes needed by the client to compute the value of the root. To verify the correctness and completeness of the set of tuples returned by the server, the owner computes the value of the root using the VO and the tuples received from the server. It then checks if the computed value is the same as the root initially computed on the data before outsourcing [DGMS00]. The computation of VO depends on the set of tuples returned. For instance, in case of a specific tuple, the VO contains the values of all the nodes being sibling of those in the path from the root to the leaf corresponding to the returned tuple. With reference to relation MEDICALDATA in Figure 2.1 and the Merkle hash tree in Figure 3.1, consider the tuple of the patient with name *Fred*. The server returns tuple t_6 and its VO contains the gray nodes in the figure. The data owners can then compute hashes over the t_6

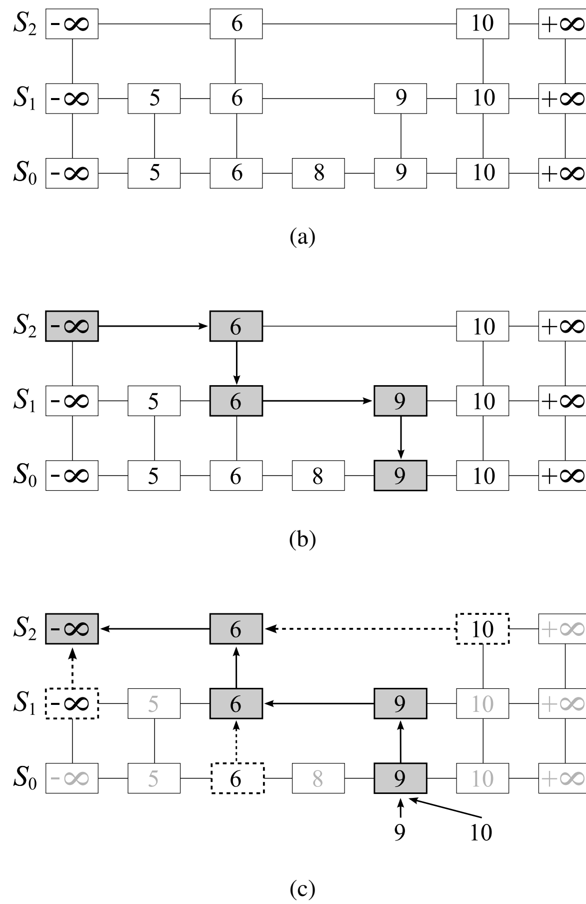


Figure 3.2: A skip list for set $\mathscr{S}$={5,6,8,9,10} with three levels (a), search process for key value 9 (b), and verification object for a query searching for value 9 (c)

and VO, as illustrated in the figure, to compute the root of the tree which is then compared to the original one.

The original technique illustrated in [DGMS00] has been extended to improve the efficiency of the verification processes (e.g., [LHKR06, PT04]), and to support integrity verification of multiple (joined) relations [YPPK09].

Skip lists. Another authenticated structure that can be used to verify the integrity of an outsourced relation (and of queries over it), is represented by *skip lists* [AGT01, DP07]. A skip list for a set $\mathscr{S}$ of distinct key values is a set of lists $S_0, S_1, \ldots, S_k$ such that: *i)* S_0 contains all keys in $\mathscr{S}$ in non-decreasing order, together with sentinels $-\infty$ and $+\infty$; and *ii)* list $S_i, i = 1, \ldots, k$, contains an arbitrary subset of the keys included in S_{i-1} that always includes sentinels $-\infty$ and $+\infty$. Figure 3.2(a) illustrates a skip list with three levels for $\mathscr{S}$={5,6,8,9,10}.

Skip lists support search operations for the definition of VOs for sets of tuples in r. The search operation for a key value v in a skip list starts from sentinel $-\infty$ in the top list (i.e., S_k). It then operates through operations *hop forward*, moving right along the current list until the visited key value v_i is the largest value lower than or equal to v, and *drop down*, moving down a list (i.e., from S_j to S_{j-1}). The search iteratively hops forward and drops down until it reaches the bottom list S_0. For instance, with reference to the skip list in Figure 3.2(a), Figure 3.2(b) illustrates a search for value 9, where accessed nodes are denoted in gray.

Skip lists can be efficiently used to verify the integrity of tuples with key v in $\mathscr{S}$ (obtaining, at the same time, guarantees over the integrity of the outsourced overall relation). To achieve this, the skip list defined for $\mathscr{S}$ is authenticated adopting a *commutative* and *collision-resistant* hash function (i.e., a hash function such that $h(x,y)$=$h(y,x)$). Each node in the skip list is associated with a label, computed through the commutative and collision-resistant hash function, that depends on the elements on its right and below it. For the nodes in the bottom list S_0, the label $f(v,S_0)$ of node v is computed as the hash of its value v and: the value of the node w on its right, if w also belongs to S_1 (i.e., $f(v,S_0) = h(v,w)$); the label $f(w,S_0)$ of the node w on its right (i.e., $f(v,S_0) = h(v, f(w,S_0))$, otherwise. For instance, with reference to Figure 3.2(a), $f(9,S_0) = h(9,10)$ while $f(6,S_0) = h(6, f(8,S_0))$. For the nodes in S_i, $i > 0$, the label $f(v,S_i)$ of node v is: the same as the label of v at S_{i-1}, if the node w on its right also belongs to S_{i+1} (i.e., $f(v,S_i) = f(v,S_{i-1})$); the hash of the labels of the node below v and on of the node w on the right of v (i.e., $f(v,S_i) = h(f(v,S_{i-1}), f(w,S_i))$, otherwise. For instance, with reference to S_1 in Figure 3.2(a), $f(5,S_1) = f(5,S_0)$ while $f(6,S_1) = h(f(9,S_1), f(6,S_0))$. The label of the starting node s of the skip list (i.e., the first sentinel node in the top list) is signed by the data owner and maintained locally for future integrity checks.

If a search for element v over a skip list returns a positive answer (i.e., $v \in \mathscr{S}$), the integrity verification process checks the existence of the value itself. Otherwise, it verifies the existence of two elements v' and v'', consecutive in list S_0, such that $v' < v < v''$. To this aim, the owner receives a verification object that includes the label of the nodes on the right and below the nodes forming the path to v, which are necessary and sufficient to compute the label of the received nodes. For instance, consider the skip list in Figure 3.2(a) and suppose to search key value 9. Figure 3.2(c) highlights the visited nodes (gray nodes) and the node included in the verification object (dashed nodes). The verification object then corresponds to the list $\langle$9, 10, $f(6,S_0)$, $f(-\infty,S_1)$, $f(10,S_2)\rangle$. The data owner verifies the answer by hashing the values in the verification object and comparing the result with the label $f(s)$ of the starting node s of the skip list, locally stored at outsourcing time.

A modification to $\mathscr{S}$ due to insertion/deletion of a value v translates to an update, efficiently performed in $O(\log(|\mathscr{S}|))$, of the associated skip list for inserting/deleting v.

3.2 Probabilistic approaches

All the techniques described in Section 3.1 can assess the integrity of tuples with contiguous values for (or resulting from the evaluation of queries over) the attribute(s) over which the authenticated structures have been built. While ensuring integrity with full confidence, no guarantee is provided for queries operating over other attributes. *Probabilistic* approaches are not limited to operate on specific subsets of attributes, but ensure integrity with a certain degree of confidence. Current probabilistic approaches are based on the insertion of *fake tuples* in the outsourced relation, on the *controlled replication* of a subset of tuples, or on a combination of these two techniques. Fake and replicated tuples are guaranteed to be indistinguishable from the original ones, and can be detected only by authorized users. Unauthorized modifications to the original relation will, with a certain degree of probability, affect these control tuples and can be witnessed during the verification process.

Fake tuples [LW09, XWYM07]. Fake tuples are inserted in the relation before storing it at the cloud provider, and are built in such way to appear indistinguishable, to the eyes of the cloud

provider, from original tuples. The insertion of fake tuples is driven by the data owner according to a deterministic function f operating over the domains of the attributes in the relation. Users authorized to check query integrity know this function. Given the result of a query q returned to the requesting client, the client checks whether all the expected fake tuples belong to the query result. Absence of one or more expected fake tuples satisfying the query signals incompleteness of the query result. As proved in [XWYM07], even a limited number of fake tuples ensures high probabilistic guarantee of completeness.

Controlled replication [WYPY08]. An alternative probabilistic approach to verify data integrity consists in replicating all the tuples in the relation to be outsourced that satisfy a *replication condition* C_r. The original tuples in the outsourced relation are then encrypted with a key k_1, and the tuples satisfying the replication condition are duplicated and encrypted with a different key k_2. The relation stored at the provider then includes two copies of each tuple satisfying C_r, one encrypted with k_1 (i.e., $E_{k_1}(t)$), and one encrypted with k_2 (i.e., $E_{k_2}(t)$). The provider, knowing neither k_1 nor k_2, cannot identify pairs of replicated tuples. Given a query q formulated by the user over the original relation, the client transforms it into two queries q_1 and q_2 equivalent to q. One of these queries operates on the original data collection (i.e., on tuples encrypted with k_1), while the other operates on replicated tuples only (i.e., on tuples encrypted with k_2). To verify the completeness of the query result, the client checks the presence of two copies of each tuple in the query result that satisfy the replication condition C_r. The presence of one copy only of these tuples signals the incompleteness of the query result.

We note that the two strategies illustrated above can also be applied in combination, ensuring as a consequence complementarity of controls and stronger integrity guarantees (e.g., [DFJ+14a, DFJ+13, DFJ+14b]).

3.3 Proof of possession and retrievability

Proofs of retrievability (PORs) and *Proofs of data possession* (PDPs) were introduced by Juels and Kaliski [JKJ07] and by Ateniese et al. [ABC+07], respectively, in 2007. A POR or PDP scheme allows a data storage center to produce a concise proof that a user (called also *verifier*) can retrieve a target file F that he/she previously uploaded. In other words, a successful execution of POR assures the verifier that F is retrievable. Moreover, a POR scheme permits detection of tampering or deletion of a remotely located file, but does not protect against loss of file contents. In more details, a PDP/POR scheme consists of *setup phase* and a sequence of *verification phases*. In the setup phase, user's data is pre-processed using secret information to generate some authenticating information. Then, the user sends the data together with authenticating information to the data storage center. Data and authenticating information are then locally removed and user is left with only his/her secret information. In the verification phase, the user generates a random challenge query and the data storage center replies by generating a concise proof based on the uses's data and the corresponding authenticating information. The user afterwards decides to accept of refuse the proof, only by using his/her secret information. Informally, security requires that with very high probability a malicious data storage center cannot trick a verifier letting them believe that they can retrieve a given file.

The first efficient and provably secure (in the sense of Juels and Kaliski [JKJ07]) POR scheme was given by Shacham and Waters [SW08]. Their scheme has short client's query and server's response. It is based on the BLS signatures and it is secure in the random oracle model. In

addition, their scheme allows *public verifiability* (first proposed by Ateniese *et al.* [ABC+11]). Namely, anyone can verify the data storage center's proofs, not just the data owner. While this scheme offers fast and flexible verification, the communication costs both grow linearly roughly with the size of data. Later, Dodis *et al.* [DVW09] improve Shacham and Waters' scheme reducing the challenge message size, but the response size is still linear in the size of data. To overcome this limitation, Xu *et al.*[XC11] constructed a POR scheme with private verification (i.e., only the data owner can verify the remotely stored data) and constant communication cost. Later Yuan and Yu [YY13] constructed a POR scheme with public verification and constant communication cost.

3.4 Integrity assurance towards multiple verifiers

The techniques for integrity verification reported in the preceding sections rely on public keys and, more generally, authenticators (like the verification objects) being available to a client that attempts to verify a query response. It is often the case that multiple clients retrieve data and may also update it. In this case one has to additionally ensure that all clients are synchronized in terms of the *version* of the data against which they verify integrity.

In a model where many clients may perform updates, further work has considered that all clients may *write to* and *read from* the remote storage service. Assuming the clients do not communicate with each other and in the absence of synchronization, a fundamental impossibility result prevents complete consistency among all clients. In particular, since the service may mount a replay attack and answer with responses from outdated state to a target client, the victim cannot discover that more recent operations have been performed by other clients.

This leads to the possibility of *replay attacks*, where the service may answer with correctly authenticated but obsolete values, and to *split-brain attacks* against consistency, where the service may answer differently depending on the client posing the query.

Starting with SUNDR [MS02, LKMS04], several contributions in this context have improved the achievable consistency guarantees, explored the fundamental tradeoffs regarding the interaction between clients and the service, and reduced the implementation cost [CSS07, WSS09, CKS11].

Recent protocols guarantee atomic operations to all clients when the service is correct and so-called *fork-linearizable semantics* when the service is faulty. Fork-linearizability makes it much easier for the clients to detect violations of integrity and consistency by the service; specifically, it means that all clients which observe each other's operations are consistent, in the sense that their own operations, plus those operations whose effects they see, have occurred atomically in one sequence. Otherwise, a faulty service could answer with arbitrary values from past operations and return diverging results to different clients.

The mechanisms employed by these protocols, intuitively, embed a digest of the past service state and the interactions of a client with the service into every request. Other clients, which read data modified by a first client, will verify that the digest of the past interactions is consistent with their own "view" and past interactions with the service. Such digests have been implemented with vector clocks and hash chains in different protocols.

4. Techniques for access confidentiality

The use of the encryption techniques presented in the previous chapters offers a protection of confidentiality that is adequate for many scenarios. Still, the fact that the storage provider controls the physical storage of the data can be the basis for the leakage of some information about the behavior of the client and of the content stored in the system. In order to mitigate these threats, techniques are being developed by the research community [SS13, SvS$^+$13, WSC08, WSS09] that aim at voiding or significantly reducing the information that the storage provider can derive from the control of the physical representation of the data.

In this chapter we will consider the threats that derive from the control of the physical representation of the data and will provide an overview of the techniques that are being designed to solve this problem. Specific attention will be dedicated to the description of the shuffle index, which is expected to be the center of a specific implementation effort within ESCUDO-CLOUD.

4.1 Motivation

When data is stored externally by providers, there is a natural need to protect the confidentiality of the data as well as to control access to it. The confidentiality requirements can indeed be structured in: the data being outsourced (*content confidentiality*); the fact that accesses aim at a specific piece of information (*access confidentiality*); the fact that two accesses aim at the same target (*pattern confidentiality*).

Several solutions have been proposed for guaranteeing data availability and for protecting the confidentiality of the remotely stored data. Typically, as discussed previously in the document, solutions focusing on data confidentiality consider an honest-but-curious server. These systems rely on the application of an encryption layer to protect the outsourced data. These solutions provide a variety of techniques for elaborating queries on encrypted data. In general, they target content confidentiality but do not address access and pattern confidentiality.

Access and pattern confidentiality have been addressed within several lines of research. One is represented by *Private Information Retrieval* (PIR) proposals (e.g., [OS07, SC07]), which provide protocols for querying a data collection that prevent the storage server from deriving information about the tuples that are being accessed. PIR approaches typically work in scenarios where the external database being accessed is in plaintext (i.e., content confidentiality is not an issue). Current PIR solutions have high computational complexity and are considered not yet applicable to real systems. Recent solutions, based on a careful adaptation of the Oblivious RAM data structure (e.g., [SS13, SvS$^+$13, WST12]), protect access and pattern confidentiality at a reduced cost with respect to PIR. These solutions however imply a still relevant computational overhead, compared to the adoption of non-privacy preserving access structures.

In this chapter we provide a concise description of a novel approach to efficiently retrieve data from an encrypted representation, addressing the aspects of the privacy problem discussed above.

The reference scenario is the one common to the ESCUDO-CLOUD project, where a *data owner* stores and retrieves data from an external honest-but-curious server. The access requests should not reveal to the server which kind of access is being executed (i.e., read or update) and should guarantee content, access, and pattern confidentiality.

The approach is based on a data structure called *shuffle index*. The shuffle index can be considered an extension of the well-known $B+$-tree, the structure most commonly used to efficiently store and execute direct access over large collections of data. The critical advantage $B+$-trees offer compared to hash structures is the support of a total order relationship among the values of the index key. The shuffle index assumes data to be organized in a $B+$-tree with no chain of pointers connecting the leaves. All the nodes are encrypted, to protect at a basic level data confidentiality. The principle used to realize stronger confidentiality relies on the adoption of several techniques: cover (fake) requests, cache nodes, and shuffling of nodes after each access. The risk is that each access may give to the server some information; the same access request also introduces a source of uncertainty that reduces the amount of information the server can build over the behavior of the user. The shuffle index supports both equality and range queries. Range queries indeed represent a crucial property of the shuffle index, with important application impact and receiving limited support in other techniques for access privacy. To guarantee that confidentiality is preserved when the outsourced data collection is updated, a "probabilistic split" approach is used. In this way, an observer cannot infer whether an access to the shuffle index is updating or not the data collection.

The evaluation of performance for the shuffle index is not trivial. Since it requires the execution of a number of additional access requests, it inevitably imposes an overhead in terms of data transfers compared to a solution that does not provide access privacy. On the other hand, all solutions of this type impose such an overhead and in many applications the real bottleneck to system performance is represented by the latency of the network rather than its capacity. Compared to solutions that offer similar ability in supporting the efficient execution of range queries over arbitrarily large data collections, the overhead is limited to 20%, which can be considered acceptable. It is also to note that the shuffle index relies on a soft and compact state stored at the client side. This makes it resilient against failures at the client side and applicable even in clients with limited computational resources and activated on-the-fly. These properties are not exhibited by several of the solutions that have similar goals and often do not permit to be accessible by different clients. Instead, after each access the shuffle index stored at the server is in a consistent state.

4.2 Violations of access privacy

To understand the motivation for the adoption of techniques protecting access privacy, we describe how sensitive information about users' activities or data content can be improperly leaked even in the presence of encryption of the stored data.

The accesses to data can leak to observers, including the server, sensitive information about users' activities or enable them to infer data content (otherwise protected by encryption). The specific assumption is that, to offer adequate efficiency in the management of large data collections, the outsourced data are accessed using a *Precise Query Protocol* (PQP). The support of the protocol requires the organization of the data into separate nodes, allowing the client to retrieve only the node that contains the data item(s) the client is interested in recovering.

For concreteness, we consider two representative scenarios. In the first scenario (also described in [WSC08, SvS$^+$13]) data describe stocks stored on behalf of a financial organization and accessed by the organization's customers. Access to a given item corresponds to the search, in the

data structure, of a specific stock. In the second scenario, data are encrypted authenticators (e.g., passwords or biometric traits) which control physical access to locations by employees. Access to a given data item corresponds to the search, in the data structure, of the authentication proof being sought (e.g., the encrypted biometric traits of the employee entering the gate). While in both scenarios basic encryption is certainly adopted, this is not sufficient, as observations on the accesses to such data can, if no access privacy technique is used, breach confidentiality of the data themselves or leak sensitive information on users' activities.

For simplicity, we assume that each atomic unit of access (i.e., a node) corresponds to a single data item. This simplification does not limit the flexibility of the approach; for trees supporting range queries, while every data block will store multiple items, such items will all reside within a given range. Also, the fact that multiple data items can be associated with a single node adds noise in the reconstruction.

The external server, storing the (encrypted) data and managing access to it, knows the encrypted physical blocks where the data is stored and observes every access to those blocks. We now show how the server can, based on such information, breach confidentiality of the encrypted data or of the actions on them. Four main cases can be identified, depending on the type of access and the assumption on the knowledge held by the server.

No knowledge Even if the server has no knowledge about the data, the static nature of the (encrypted) data structure leaks information. By observing accesses, the server can establish whether two accesses targeted the same data item. For the financial domain, the server would be able to infer whether different transactions operated on the same stock. In the authentication scenario, the server would be able to recognize all accesses by the same employee.

Single-access knowledge If the server has knowledge about one specific access instance (e.g., it knows a given access is aimed at a given stock or that a given employee is entering a given building), then this specific piece of information can violate the confidentiality of many accesses. Knowledge of a specific access discloses in fact to the server where a given data item is stored: all accesses operating on the same block would then aim at the same item. For the financial application, the server would then be able to know when different transactions will operate on the same stock. In the authentication scenario, the server would be able to keep track of all the accesses by the same employee.

Data or access frequency knowledge If the server has (some) knowledge on the frequency of accesses to the stored items, then this can lead to the violation of confidentiality. Such knowledge does not necessarily require availability of confidential information (e.g., distribution of stock operations). Since the server observes every access, it can build a frequency histogram of the data blocks accessed. Comparing these observations with the frequency knowledge above, the server can establish the content of some physical blocks (typically of outliers, most frequently or least frequently accessed).

No knowledge, with range queries If the query protocol supports the efficient execution of range queries, the server can observe the fact that accesses hit a specific collection of blocks. If the blocks are accessed in a rigid order consistent with the key order, very quickly the server can infer how the content is organized; if the collection returned for each query does not leak, the order, it is sufficient for the server to combine several of such observations. Then, a limited amount of additional knowledge about the content of the encrypted data can allow the server

to reconstruct with great precision the plaintext content. In [IKK14] the authors show that such an attack is able to identify up to 90% of the plaintext content after a limited number of range queries have been executed, even under the assumption that the leaves of the tree are not visited in order and are simply accessed as a set. In both scenarios such inference would allow the server to establish the plaintext order of the stored values (stocks or authenticator tokens, assuming the ordering key of the first ot be the stock compact identifier and of the latter to be the employee's last name). A limited set of range queries quickly discloses the ordering of blocks and can be used to reconstruct the content.

All the scenarios of inference above are made possible by the static nature of the data structure. The application of encryption does not protect against the exposure of the fact that accesses targeted an identical item and, when the item becomes known, repeated accesses are recognizable. Techniques for access privacy, like shuffle index, break such a static correspondence by continuously reorganizing the data structure at every access. Subsequent accesses to the same data item cannot be related anymore since they will target different physical blocks.

4.3 Shuffle index

The design of the shuffle index starts from the observation that the B+-tree structure used by a regular index sees a natural separation between the abstract structure and its logical realization. What is new in the domain of encrypted data is the presence of an additional level, corresponding to the concrete physical implementation. At the abstract level, we have the leaves of the tree that contain the tuples that have to be managed and in the nodes belonging to the levels above the leaves we have the representation of all the key values and identifiers of descendent nodes. At the abstract level we assume all the leaves of the tree to be ordered based on the key of the index and the nodes above the leaves can also be assumed to be located right above their descendents. This is the classical representation of trees and the connections between the nodes never intersect, with a clear organization. The logical level is the one that typically characterizes the implementation of B+-trees. Each node is associated with a logical identifier, which represents in classical realizations of the tree the identifier of the node. Node identifiers are not necessarily contiguous and there is no direct correspondence between the logical identifier and the position of the node in the abstract representation of the tree. The logical identifiers can be expected to be assigned to the nodes depending on the evolution of the tree itself. As values are inserted into the tree, nodes are added to the structure, in a position that depends on the specific value that is being inserted, which generates saturation in the node and the need to split the content of the previous node among the node itself and an additional node.

The crucial novelty that characterizes the shuffle index is the introduction of an additional layer, the *physical* representation of the tree. The idea is that the nodes are stored in the server using an encrypted format. Only the data owner, who knows the encryption key, is able to see the content of each node and interpret the logical model. What the server sees is the physical representation of the tree. From the way in which nodes are accessed, the server is able to recognize which is the level of the tree associated with every node, but the content is hidden and the server views an undistinguished collection of blocks organized in separate levels. In case the structure is static (like the encrypted B+-tree proposed in [DDJ^{+}03]), the server could be expected to monitor the sequence of accesses to the nodes to establish the connection between nodes, in the end establishing the correspondence between the physical and logical structure of the B+-tree. Knowledge

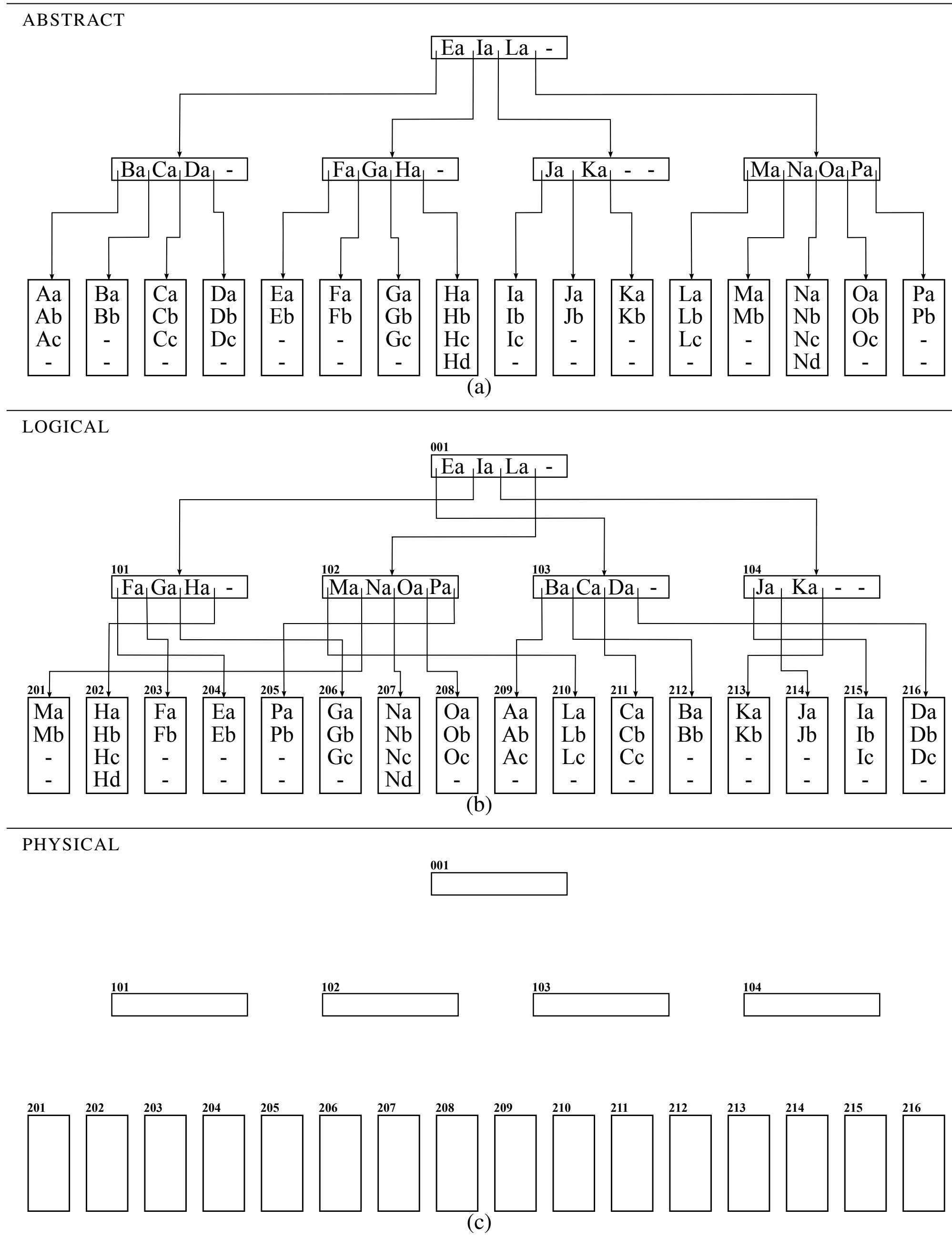


Figure 4.1: An example of abstract (a) and logical (b) representation of an unchained $B+$-tree, and of the corresponding view of the server (c)

of this relationship allows the server to realize attacks aiming at the violation of the confidentiality of the data. The design of the shuffle index can be described as aiming at keeping hidden the correspondence between the logical and physical structure. This is realized by reading at every access more than one node and after the access executing a random mixing of the nodes, which is hidden from the server as a result of some nonce that is added during every encryption.

To describe the way in which the shuffle index works, we use an example of access to the structure. For a full analysis of the algorithm, we refer to the research paper [DFP+15] produced within ESCUDO-CLOUD. In Figure 4.2 we have the same initial logical configuration presented

in Figure 4.1. We assume that the client wants to access the key value *Fb*. The shuffle index uses *cover* nodes and *cache* nodes to hide the access profile.

- Cover nodes correspond to additional accesses that are made to the tree nodes, looking for (pseudo-)randomly chosen nodes. The motivation for the use of cover nodes is to make it more difficult for the server to recognize which is the real target of an access request, which will be hidden among a number of fake covers.

- Cache nodes are instead copies of the recently accessed nodes, which are kept at the client-side and which do not have to be read from the tree. Since cache nodes are available to the client, they can be used in the shuffling step that is executed immediately after the completion of the read operations.

Consider the index in Figure 4.1 (reported for convenience at the top of Figure 4.2) and assume we use one access as cover and use two cache nodes. The target of the access is assumed to be value 'Fb', and the cache initially stores these nodes: root {001} at level 0, nodes {101, 103} at level 1, and nodes {206, 209} at level 2. Initially, two values, for example, 'Ma' and 'Ic', are randomly chosen as covers for 'Fb'.

For the first level, the one immediately below the root, the identifier of the node along the path to 'Fb' is 101, which is already in cache. Therefore, the two nodes in the paths to the cover searches (i.e., 102 and 104, respectively) are read from the server. The nodes in the first level of the cache and those accessed (i.e., downloaded from the server) are shuffled according to the following randomly chosen permutation: 101's content moves to block 102; 102's to 101; 103's to 104; and 104's to 103. The cache at the first level is updated by refreshing the timestamp of node 102. Finally, the pointers in n_0 (i.e., the root node) are updated according to the permutation, and node 001 is encrypted and stored at the server.

For the second level of the tree, the identifier of the node along the path to 'Fb' is 203, which does not belong to cache, and hence the second cover is dropped. Nodes 201 and 203 are read from the server. The nodes in the second level of the cache and those just read are shuffled according to the following permutation: 201's content moves to block 209; 203's to 201; 206's to 203; and 209's to 206. Node 201 is inserted into the second level of the cache and node 206 (which we suppose is the least recently used) is pushed out. The pointers in the nodes accessed during the previous iteration (i.e., 101, 102, 103, 104) are updated according to the permutation, encrypted, and sent to the server.

Finally, accessed leaf nodes (i.e., 201, 203, 206, 209) are encrypted and sent to the server. Node 201 is returned to the client. The state of the shuffle index at the end of the sequence of operations is shown at the bottom of Figure 4.2.

4.4 Security analysis

The protection offered by the shuffle index derives from the separation between the logical level of the tree, fully accessible only when the encryption key is known, and the physical level, which is under the control of the server and can also be observed in its accesses by any adversary who may be able to monitor the access requests produced by the client.

The continuous shuffling, which occurs at every access, is able to degrade any information the server may possess on the correspondence between nodes and blocks, reaching, after a sufficient number of accesses, a complete loss of information. This result shows an interesting feature of the

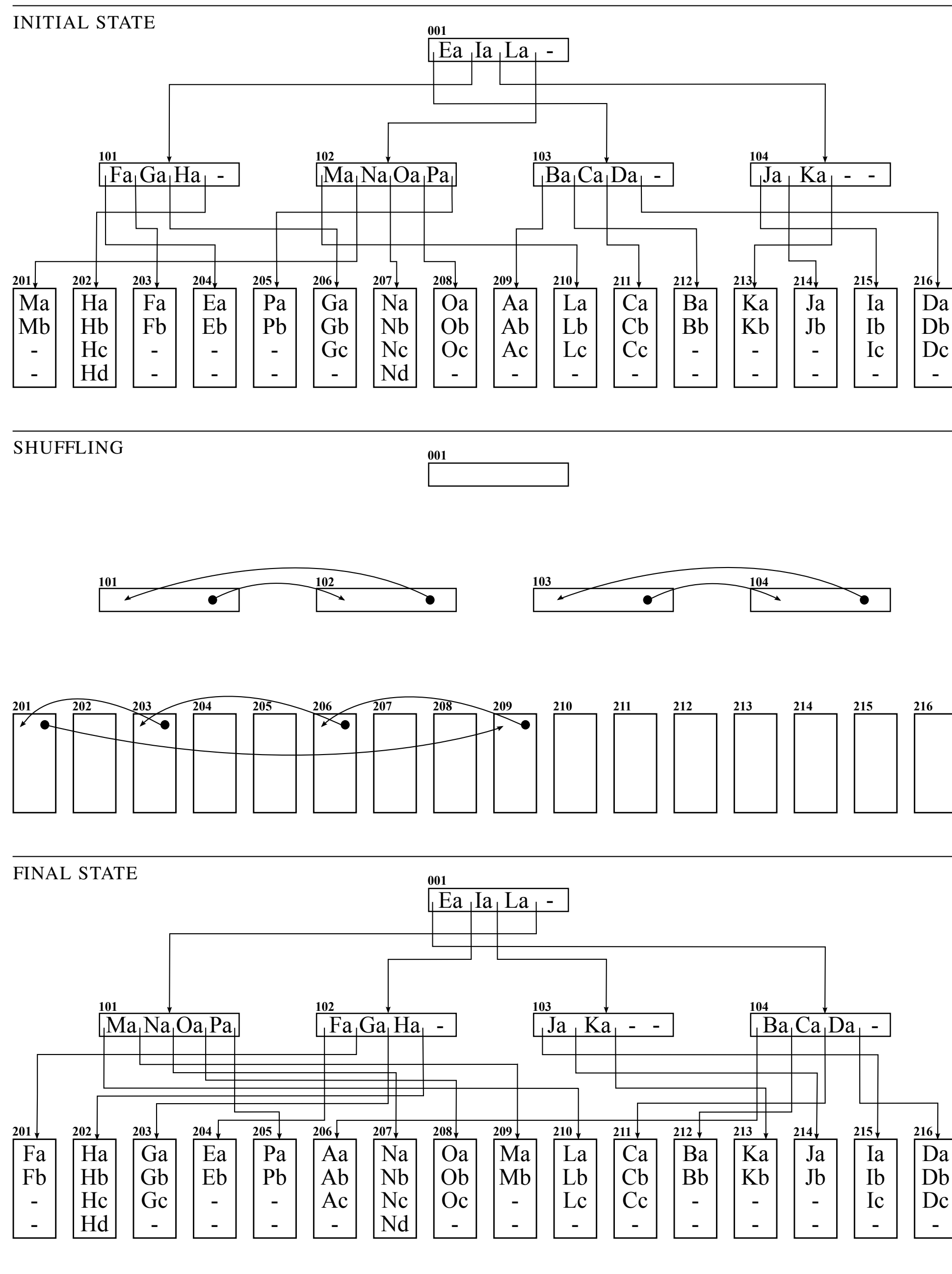


Figure 4.2: Evolution of the shuffle index

shuffle index behavior and the absence of long term accumulation of information. The knowledge that the server has on the correspondence between nodes and blocks storing their encrypted content can be modeled as the probability p that the server can establish an association between a node at the logical level and a block at the physical level.

The application of cover searches and shuffling techniques at each access progressively destroys any information the server may have acquired on the correspondence between the logical and physical level, thus providing access confidentiality. This can be proved formally and we refer to the discussion in [DFP$^+$15].

Access confidentiality is characterized as the protection against the server's ability to associate a specific access request with a specific node/data. As discussed before, static encrypted indexing

structures do not exhibit access confidentiality, because the server may exploit information on the frequency of accesses (e.g., the server may know that people last names are used as key and "Smith" is the most frequently accessed value) and may thus identify the content associated with a specific block.

The shuffle index offers a natural protection against this attack. Even disregarding the use of cache nodes and considering only the contribution offered by covers, every time an access is performed any information on the specific access has to be divided among all the nodes involved in the access request (the real target plus the cover nodes). After the nodes are shuffled, the information on the correspondence between nodes and blocks is further destroyed. In general, we observe here a reenforcing mechanism: access confidentiality is typically at risk when there are values that are characterized by high access frequency, but the higher the access frequency, the greater the destruction of information realized by shuffling. A verification of this aspect is provided by experiments that show how extremely different target distributions produce almost identical block access profiles.

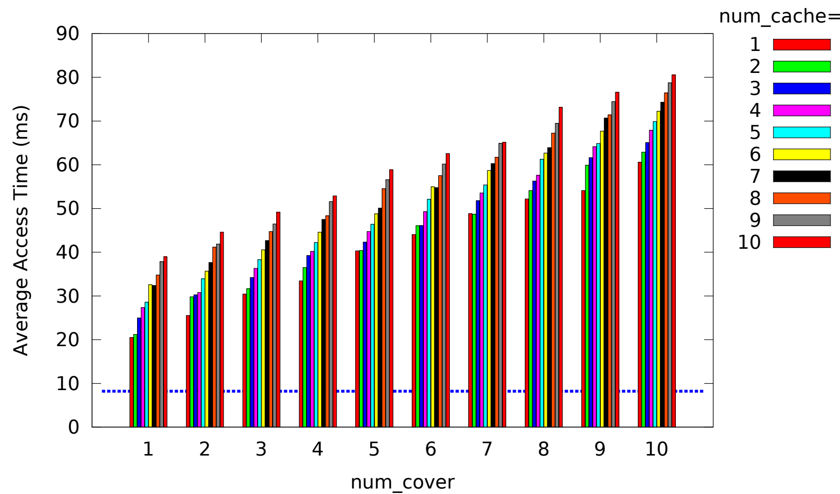
Pattern confidentiality is characterized as the protection against the server ability to recognize that two separate accesses refer to the same node. The degradation of information that derives from shuffling guarantees that accesses separated by a significant number of steps will not be recognizable. With respect to short distances among the accesses, the shuffle index fully protects pattern confidentiality when the distance between the observations is within the size of the cache. As a consequence, the evaluation of range queries guarantees their indistinguishability from a sequence of arbitrary accesses to the shuffle index. Indeed, the distance between the accesses in a range query is within the size of the cache, for any non-empty cache, and presents repeated accesses to internal nodes.

The analysis of the security of the insertion or deletion of values form the index shows that these operations can be made indistinguishable from read accesses, offering an additional level of access confidentiality.

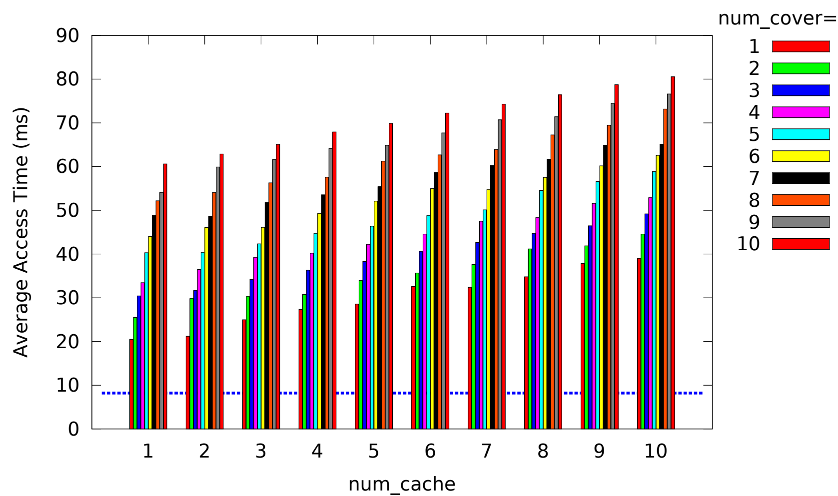
4.5 Overhead analysis

The protection offered by the shuffle index comes at a cost. The additional accesses to nodes and the need to continuously rewrite, after each access, the content of the accessed nodes has a significant impact on the amount of data that has to be transfered and increases the time required to complete an access request. It is noted that the evaluation of the cost deriving from the use of the shuffle index has to be done in comparison with the cost exhibited by index structures that offer the same flexibility and basic protection guarantees, i.e., are able to support the efficient retrieval by key value of tuples, are able to support the efficient execution of range queries on large data collections, and adopt an encrypted representation of the information on the server. Given these requirements, the natural comparison is with the cost exhibited by an encrypted static B+-tree, as other solutions (e.g., keyed-hash structures, Bloom filters, etc.) do not satisfy all these requirements.

An implementation of the shuffle index has been used to evaluate its performance. Experiments have considered two main scenarios. First a scenario where the client and the server operate in a wide area network, by properly simulating adequate network configurations, and then a scenario where they operate in a local area network. The hardware used in the experiments included a server machine with an Intel Core i7-920 CPU at 2.6 GHz, L3–8 MiB, 12 GiB RAM DDR3 1066, 120 GiB SSD disk SATA III with read throughput 240 MiB/s, write throughput 220 MiB/s,



(a)



(b)

Figure 4.3: Access time of the shuffle index in a switched Fast Ethernet LAN as a function of the number of covers (a) and of the size of the cache (b). The dashed line (baseline) represents the service time observed using a plain encrypted index with the same height.

running an Ubuntu operating system with the ext4 file system. The client machine was running an Intel Core i5-2520M CPU at 2.5 GHz, L3–3 MiB, 8 GiB RAM DDR3 1066, running an Arch Linux operating system with the ext4 file system.

To evaluate the performance of the shuffle index we took into consideration the cost of: CPU, disk, and network.

CPU: The computational load required for the management of the shuffle index is quite limited. The algorithm uses only symmetric encryption and a MAC function; the execution times we measured on a 8 KiB block for both cryptographic functions are under 100 μs, a negligible fraction of the time required by network and disk accesses, which then drive the performance of the shuffle index.

Disk: We analyzed the performance of the shuffle index when the client and the server operate in a local area network (we used a 100 Mbps switched Fast Ethernet network with average Round Trip Time - RTT equal to 0.67 ms). In this configuration, disk performance becomes the limiting factor. Figures 4.3(a) and 4.3(b) report observed times in milliseconds. The values are grouped by the same value of cover nodes and by the same value of cache nodes, both varying from 1 to 10. The access time grows linearly with the number of cover searches, since every additional cover requires to traverse an additional path in the shuffle index; the depth of the tree remains constant. Although an increase in cache nodes causes a growth in the number of blocks written for each level of the shuffle index, the number of cached nodes has a smaller impact on the access time. This is justified by the fact that the disk operations caused by the increase in cache size greatly benefit from buffering and cache mechanisms at the operating system and disk controller level, as cached blocks will be repeatedly written and the cost of those operations will be lower than that associated with cover and target blocks. As it is typical for database index structures, the bottleneck in the performance of the shuffle index in a Local Area Network (LAN) scenario is the number of read and write operations on the local disk.

The dashed lines in Figures 4.3(a) and 4.3(b) represent the baseline obtained measuring the access time of a plain encrypted index with the same static structure of the shuffle index (this is essentially the tree structure that was proposed in [DDJ$^+$03]). The adoption of a plain encrypted index still requires the client to visit the nodes in the tree level-by-level, but it does not use covers, caching, and shuffling to provide access and pattern confidentiality. The performance overhead introduced by the adoption of our protection techniques ranges from $\times 2.5$ to $\times 10$ of the baseline, depending on the number covers and on the size of the cache. This increase in the access time is mainly due to the disk cost caused by a higher number of (random) read/write operations.

Network: To better analyze how the network impacts on the time necessary to access the shuffle index, we first study the number of messages and the number of nodes (which corresponds to the number of bytes) exchanged between the client and the server during each access. Given a shuffle index of height h, the client and the server exchange $2h+1$ messages for the evaluation of each access. In fact, for each level in the shuffle index but the root level, the client sends a request for a set of blocks, and the server replies with their content. The client also sends to the server, together with his/her request for blocks at level l, also the blocks to be rewritten at level $l-1$. The visit of each level then implies the exchange of two messages. The root level, on the contrary, implies no message exchange as the root node is in cache. An additional message is finally required to write the leaves of the shuffle index.

The number of blocks downloaded from the server for each level of the shuffle index (except for the root level) is always equal to $1{+}Cov$ (where Cov represents the number of cover nodes). The number of blocks sent to the server is $1{+}Cov{+}Cache$ for each level (where $Cache$ represents the number of cache nodes), but the root level that requires to write one block only. Figure 4.4 illustrates the average number of bytes exchanged during an access, considering a shuffle index with Fanout=512, Height=2, blocks of 8KiB, and varying the number of cover searches and of nodes in cache for each level of the shuffle index between 0 and 10.

We analyzed the performance of the shuffle index when the client and the server operate in a Wide Area Network (WAN). This scenario, where a client uses a remote untrusted party for the private access to data, is the most interesting and natural for the shuffle index. We adopted a network configuration suitable for interactive traffic between the client and the server, in contrast to configurations where network connections are sized to better support database replication or

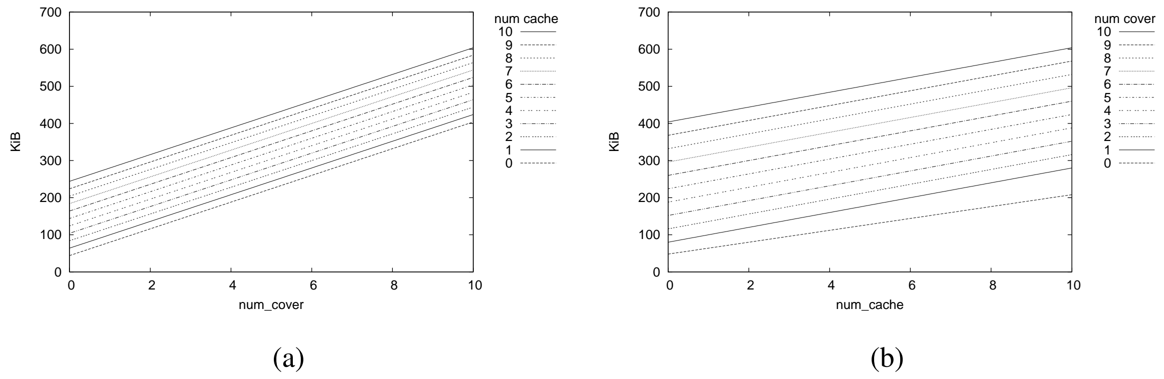


Figure 4.4: Number of KiB exchanged as a function of the number of covers (a), and of the size of the cache (b)

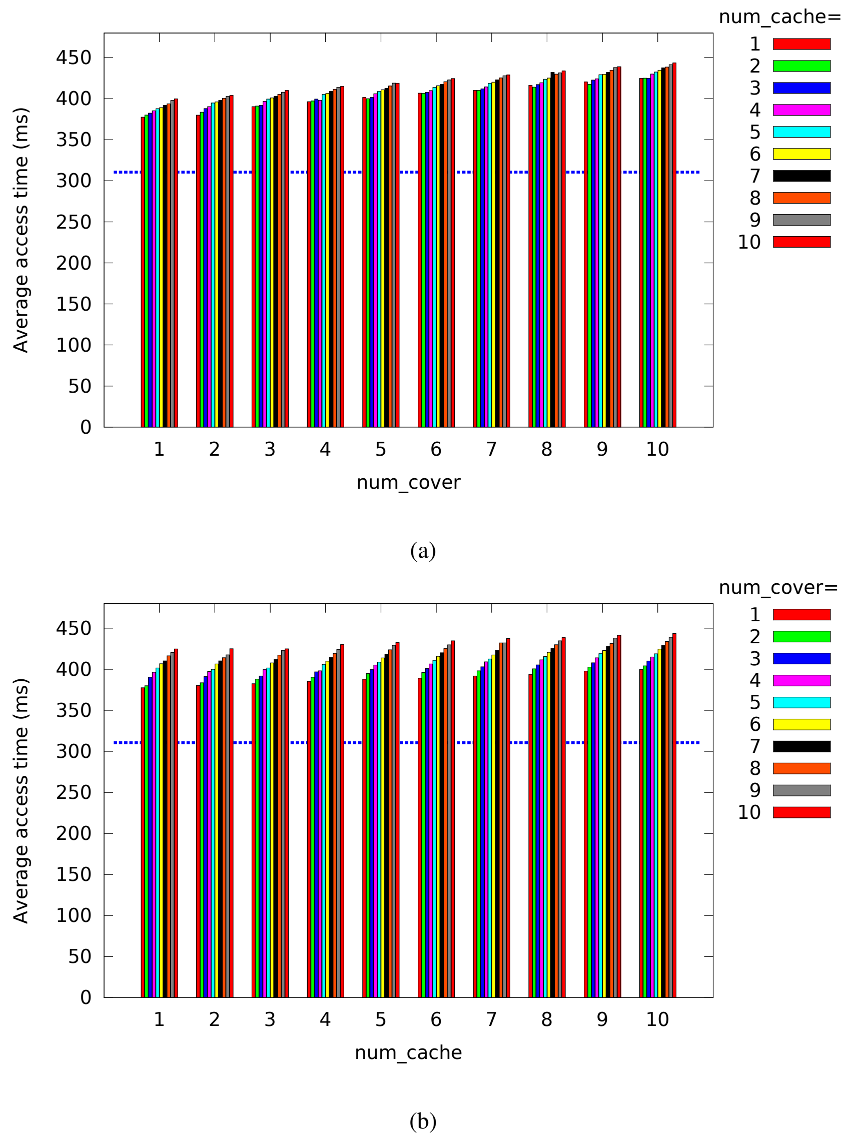


Figure 4.5: Access time of the shuffle index in a WAN configuration as a function of the number of covers (a), and of the size of the cache (b). The dashed line (baseline) represents the service time observed using a plain encrypted index with the same height.

database back up operations. We chose a representative WAN configuration with LAN-like bandwidth and round trip time typical of US east-coast to Europe connections (modeled as a normal distribution with mean of 100 ms and standard deviation of 2.5 ms). In this configuration, network performance becomes the limiting factor. Figures 4.5(a) and 4.5(b) report observed times in milliseconds. The values are grouped by the same value of cover nodes and by the same value of cache nodes, both varying from 1 to 10. As it is visible from the figure, for every tested pair of cover and cache nodes, the average access time was below 450 ms. The average access time to the shuffle index mostly depends on the number of send and receive operations on the network channel between the client and the server. Therefore, the performance overhead caused by an additional cover search is greater than the overhead caused by an additional cached search. Indeed, cached nodes do not need to be downloaded from the server at each access as they are locally stored at the client side.
As with the LAN scenario, we compared the access time of our shuffle index with the access time of a plain encrypted index with the same static structure of the shuffle index, which is represented by the dashed lines in Figures 4.5(a) and 4.5(b). We note that the performance of the shuffle index scales much better in a WAN configuration than in a LAN scenario. Indeed, the performance overhead caused by the adoption of our protection techniques ranges from $\times 1.2$ up to $\times 1.4$, depending on the number covers and on the size of the cache. More precisely, from the results obtained we can conclude that each increase in the number of covers and cache searches adds 1.2% and 0.6%, respectively, of the plain encrypted access time to the overall performance. We note that configurations with Cov=1 and *Cache* between 1 and 2 already provide a strong degree of access and pattern confidentiality and cause a limited performance overhead. The choice of Cov=1 permits at each access to shuffle the position of the target node with a randomly chosen node; larger numbers of covers would offer a faster degradation of the information the server may obtain from the access, but each cover also requires an additional node involved at every step of the protocol. With respect to the cache, the choice of *Cache* between 1 and 2 guarantees the protection of accesses that are repeated after a short period, and this is particularly important for range queries, which would otherwise be recognizable by the repetition of the accesses to the nodes above the leaf; the increase in *Cache* produces a limited performance impact and the chosen value appears a good compromise between security and the extreme attention to performance that characterizes most scenarios. Configurations with Cov=1 and *Cache* between 1 and 2 have a performance overhead factor of $\approx$20% with respect to a plain encrypted index (the measured values were 310.58 ms for the plain encrypted index and less than 380 ms for the shuffle index).

The performance overhead caused by our approach to support updates to the data collection in a broadband WAN configuration is $\approx 1.42(Cache{+}Cov)$ ms, depending on the number of nodes that have been split. The overhead caused by the support of updates to the data collection is almost the same in the LAN and in the WAN scenarios, because the split of a node causes a limited increase in the transmission time of the set of blocks exchanged between the client and the remote server, which is not influenced by the network delay.

We note that, even in a broadband WAN configuration where the network latency is the dominant factor, our solution enjoys a limited communication and computational cost. From the observations above, we believe our approach to be particularly appealing in many application scenarios, since it provides adequate access and pattern confidentiality at an affordable overhead.

5. Conclusions

The protection of data stored by a cloud provider can be based on contractual obligations. Greater confidence in the respect of the confidentiality and integrity of the data can derive from the use of technical solutions. As shown in this document, many solutions rely on the use of cryptographic techniques, but other approaches can also be used. Classical encryption functions have to be adapted in order to meet the protection requirements that characterize this domain.

The work in WP2 will continue with the further refinement of the techniques analyzed before. An activity is also ongoing in ESCUDO-CLOUD to adapt the security solutions to frameworks supporting the realization of cloud applications. This implementation effort will also contribute to the project Use Cases, with a direct impact on the industrial scenarios that are considered in WP1.

Bibliography

[ABC+07] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS 2007*, 2007.

[ABC+11] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song. Remote data checking using provable data possession. *ACM Trans. Inf. Syst. Secur.*, 14(1):12:1–12:34, June 2011.

[ABG+05a] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In *Proc. of CIDR 2005*, Asilomar, CA, USA, January 2005.

[ABG+05b] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In *Proc. of CIDR 2005*, Asilomar, CA, USA, January 2005.

[AGT01] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In George I. Davida and Yair Frankel, editors, *Proc. 4th Information Security Conference (ISC)*, volume 2200 of *Lecture Notes in Computer Science*. Springer, 2001.

[BBL12] M. Benedikt, P. Bourhis, and C. Ley. Querying schemas with access restrictions. *Proc. of VLDB Endowment*, 5(7):634–645, March 2012.

[BGLS03] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proc. of EUROCRYPT 2003*, Warsaw, Poland, May 2003.

[BKR12] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. Cryptology ePrint Archive, Report 2012/631, 2012. `http://eprint.iacr.org/`.

[BKR13] M. Bellare, S. Keelveedhi, and T. Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, 2013.

[BPW11] J. Biskup, M. Preuß, and L. Wiese. On the inference-proofness of database fragmentation satisfying confidentiality constraints. In *Proc. of ISC 2011*, Xi'an, China, October 2011.

[CDF+07] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Fragmentation and encryption to enforce privacy in data storage. In *Proc. of ESORICS 2007*, Dresden, Germany, September 2007.

[CDF+09a] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Fragmentation design for efficient query execution over sensitive distributed databases. In *Proc. of ICDCS 2009*, Montreal, Canada, June 2009.

[CDF+09b] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *Proc. of ESORICS 2009*, Saint Malo, France, September 2009.

[CDF+10] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM TISSEC*, 13(3):22:1–22:33, July 2010.

[CDF+11] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Selective data outsourcing for enforcing privacy. *JCS*, 19(3):531–566, 2011.

[CDF+12] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati. An obdd approach to enforce confidentiality and visibility constraints in data publishing. *JCS*, 20(5):463–508, 2012.

[CKS11] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. *SIAM Journal on Computing*, 40(2):493–533, April 2011. Preliminary version appears in *Proc. DSN 2009*.

[Cor15] IBM Corporation. Storage virtualization, 2015. `http://www-03.ibm.com/systems/storage/virtualization/index.html`.

[CSS07] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2007.

[DAB+02] J. R. Douceur, A. Adya, W. J. Bolosky, D. R. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical Report MSR-TR-2002-30, Microsoft Research, 2002.

[DDJ+03] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, Washington, DC, USA, October 2003.

[DF09] M. Dutch and L. Freeman. Understanding data de-duplication ratios. SNIA forum, 2009. `http://www.snia.org/forums/dmf/news/articles/SNIA_DeDupe_Ratio_Feb09.pdf`.

[DFJ+13] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Integrity for join queries in the cloud. *IEEE TCC*, 1(2):187–200, July–December 2013.

[DFJ+14a] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati. Integrity for distributed queries. In *Proc. of CNS 2014*, San Francisco, CA, USA, October 2014.

[DFJ+14b] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Optimizing integrity checks for join queries in the cloud. In *Proc. of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2014)*, Vienna, Austria, July 2014.

[DFP+15] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Shuffle index: Efficient and private access to outsourced data. *ACM Transactions on Storage (TOS)*, 11(4):1–55, October 2015. Article 19.

[DGMS00] P.T. Devanbu, M. Gertz, C.U. Martel, and S.G. Stubblebine. Authentic third-party data publication. In *Proc. of DBSec 2000*, Schoorl, The Netherlands, August 2000.

[DP07] G. Di Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. In *Proc. of DBSec 2007*, Redondo Beach, CA, USA, July 2007.

[DVW09] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, TCC '09, pages 109–127, Berlin, Heidelberg, 2009. Springer-Verlag.

[FT02] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[Gro09] Trusted Computing Group. Solutions guide to data-at-rest, 2009. `http://www.trustedcomputinggroup.org/resources/solutions_guide_to_dataatrest`.

[HHPSP11] 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 '11, 2011.

[HIM03] H. Hacigümüs, B. Iyer, and S. Mehrotra. Ensuring integrity of encrypted databases in database as a service model. In *Proc. of DBSec 2003*, Estes Park, CO, USA, August 2003.

[HMN+12] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik. Estimation of deduplication ratios in large data sets. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, 2012.

[HPSP10] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *Security Privacy, IEEE*, 2010.

[IKK14] M.S. Islam, M. Kuzu, and M. Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proc. of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY 2014)*, San Antonio, TX, USA, March 2014.

[JKJ07] A. Juels and B. S Kaliski Jr. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS 2007*, Alexandria, VA, USA, October–Novermber 2007.

[LHKR06] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. of SIGMOD 2006*, Chicago, IL, USA, June 2006.

[LKMS04] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

[LW09] R. Liu and H. Wang. Integrity verification of outsourced XML databases. In *Proc. of CSE 2009*, Vancouver, Canada, August 2009.

[MB09] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[Mer89] R.C. Merkle. A certified digital signature. In *Proc. of CRYPTO 1989*, Santa Barbara, CA, USA, August 1989.

[MND$^+$04] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:21–41, 2004.

[MNT06] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM TOS*, 2(2):107–138, May 2006.

[MS02] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[OS07] R. Ostrovsky and W. E. Skeith, III. A survey of single-database private information retrieval: Techniques and applications. In *Proc. of the 10th International Conference on Practice and Theory in Public-Key Cryptography (PKC 2007)*, Beijing, China, April 2007.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 2/E*. Prentice-Hall, Inc., 1999.

[PJRT05] H. Pang, A. Jain, K. Ramamritham, and K.L. Tan. Verifying completeness of relational query results in data publishing. In *Proc. of SIGMOD 2005*, Baltimore, MA, USA, June 2005.

[PS12] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2012)*, 2012.

[PT04] H. Pang and K.L. Tan. Authenticating query results in edge computing. In *Proc. of ICDE 2004*, Boston, MA, USA, April 2004.

[SC07] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proc. of the 14th Annual Network & Distributed System Security Conference (NDSS 2007)*, San Diego, CA, USA, February - March 2007.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2/E, 1996.

[SGLM08] M. W. Storer, K. Greenan, D. D.E. Long, and E. L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, 2008.

[SS13] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *Proc. of the 34th IEEE Symposium on Security and Privacy (S&P 2013)*, San Francisco, CA, USA, May 2013.

[SSAK14] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In *Financial Crypto*, 2014.

[SvS+13] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple Oblivious RAM protocol. In *Proc. of the 20th ACM Conference on Computer and Communications Security (CCS 2013)*, Berlin, Germany, November 2013.

[SW08] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag.

[Tam03] R. Tamassia. Authenticated data structures. In G. Di Battista and U. Zwick, editors, *Proc. 11th European Symposium on Algorithms (ESA)*, volume 2832 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2003.

[WSC08] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proc of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, Alexandria, VA, USA, October 2008.

[WSS09] P. Williams, R. Sion, and D. Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2009.

[WST12] P. Williams, R. Sion, and A. Tomescu. PrivateFS: A parallel oblivious file system. In *Proc. of the ACM Conference on Computer and Communications Security (CCS 2012)*, Raleigh, NC, USA, October 2012.

[WYPY08] H. Wang, J. Yin, C. Perng, and P.S. Yu. Dual encryption for query integrity assurance. In *Proc. of CIKM 2008*, Napa Valley, CA, USA, October 2008.

[XC11] J. Xu and E.-C. Chang. Towards efficient provable data possession. *IACR Cryptology ePrint Archive*, 2011:574, 2011.

[XCZ13] J. Xu, E.-C. 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*, 2013.

[Xu10] L. Xu. Securing the enterprise with intel AES-NI. Intel Corporation Whitepaper, 2010. `http://www.intel.com/content/dam/doc/white-paper/enterprise-security-aes-ni-white-paper.pdf`.

[XWYM07] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proc. of VLDB 2007*, Vienna, Austria, September 2007.

[YPPK09] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proc. of SIGMOD 2009*, Providence, RI, USA, June-July 2009.

[YY13] J. Yuan and S. Yu. Proofs of retrievability with public verifiability and constant communication cost in cloud. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, Cloud Computing '13, pages 19–26, New York, NY, USA, 2013. ACM.