Practical Techniques Building on Encryption for Protecting and Managing Data in the Cloud

Sabrina De Capitani di Vimercati, Sara Foresti,
Giovanni Livraga, and Pierangela Samarati

Università degli Studi di Milano – 26013 Crema, Italy
firstname.lastname@unimi.it

Abstract. Companies as well as individual users are adopting cloud solutions at an over-increasing rate for storing data and making them accessible to others. While migrating data to the cloud brings undeniable benefits in terms of data availability, scalability, and reliability, data protection is still one of the biggest concerns faced by data owners. Guaranteeing data protection means ensuring confidentiality and integrity of data and computations over them, and ensuring data availability to legitimate users. In this chapter, we survey some approaches for protecting data in the cloud that apply basic cryptographic techniques, possibly complementing them with additional controls, to the aim of producing efficient and effective solutions that can be used in practice.

1 Introduction

The rapid advancements in Information and Communication Technologies (ICTs) have encouraged the development and use of storage services based on public clouds (e.g., Microsoft Azure and Amazon S3). Users as well as companies have been therefore moving their data to the cloud, thus enjoying several benefits such as data and service availability, scalability, and reliability at a relatively low cost. Although there is no doubt that the use of cloud services brings several benefits, the storage and management of data by external cloud providers introduce new security and privacy risks that can slow down or affect the widespread acceptance of the cloud (e.g., [29,47,49,65]). A major issue concerns the fact that moving data to the cloud, data owners lose control over them and the cloud environment, being not under the direct control of the data owners, may not be fully trusted. This implies the need to protect confidentiality and provide integrity guarantees for data stored or processed in the cloud, as well as for accesses to such data. In the recent years, the research and development communities have dedicated attention to these problems, designing novel techniques to ensure proper data protection in the cloud. Guaranteeing data protection in the cloud requires ensuring their *confidentiality*, *integrity*, and *availability* [36,48,63]. Confidentiality means that data should be accessible and known only to parties authorized for that. Guaranteeing confidentiality requires then to protect: the data externally stored; the identity and/or personal information of the users accessing the data; and the actions that users perform

over the data. Integrity means that data should be protected against unauthorized or improper modifications. Guaranteeing integrity requires ensuring the authenticity of: the subjects interacting in the cloud; the data stored and maintained at cloud providers; the response returned from queries and computations. Availability means that data should be available upon user requests and that cloud providers should satisfy requirements expressed in the Service Level Agreements (SLAs) established between data owners/users and the cloud providers. Guaranteeing availability requires then providing data owners and users with the required services and enabling them to assess the satisfaction of the SLAs.

Cryptography is one of the key techniques that can be adopted to address such confidentiality, integrity, and availability problems and to increase the confidence of cloud service users. Cryptography has evolved from ancient science, mainly dedicated to the design of secret writing codes, to the scientific discipline of modern cryptography that provides techniques for addressing a wide range of security issues. While in the past cryptographic techniques were principally used to protect communications (*data in transit*), today they are also used to protect *data at rest* and *data at use* (e.g., [5,11,44]). Data at rest are recorded on a storage device (e.g., a hard drive) and can remain valuable for very long periods of time. Data at use are processed by applications to respond to queries or to make computations. In this chapter, we discuss some security problems related to the protection of data at rest and data at use in cloud environments. We analyze the relevance of cryptographic techniques to address these problems, also when they are combined with other solutions to improve protection guarantees and/or to limit the computational overhead, thus making such techniques applicable in practice. Figure 1 illustrates the reference scenario: a data owner outsources her data collection to a cloud provider, and different users access these data through their clients. This scenario is characterized by the following key security challenges, which will be covered in the remainder of this chapter.

- *Storage security*: data stored in the cloud should be: protected from unauthorized accesses, even by the storing provider (confidentiality), accessible by authorized users (availability), and correct (integrity).
- *Selective access*: data stored in the cloud should be selectively accessible by users as demanded by the access control policy defined by the data owner.
- *Fine-grained access*: encrypted outsourced data should be used for fine-grained retrieval and query execution.
- *Query confidentiality*: the target of accesses to data should be kept private.
- *Query integrity*: the results of queries and computations should be correct, complete, and fresh.

Note that cryptographic techniques have an important role in protecting data in transit also in cloud environments, where data are often transferred from one cloud provider to another one or within components of the cloud system. In these cases, classical solutions can be applied (e.g., virtual private networks and secure socket layers) and therefore we do not further elaborate on them.

The remainder of this chapter is organized as follows. Section 2 describes solutions for the secure storage of data in the cloud. Section 3 presents some

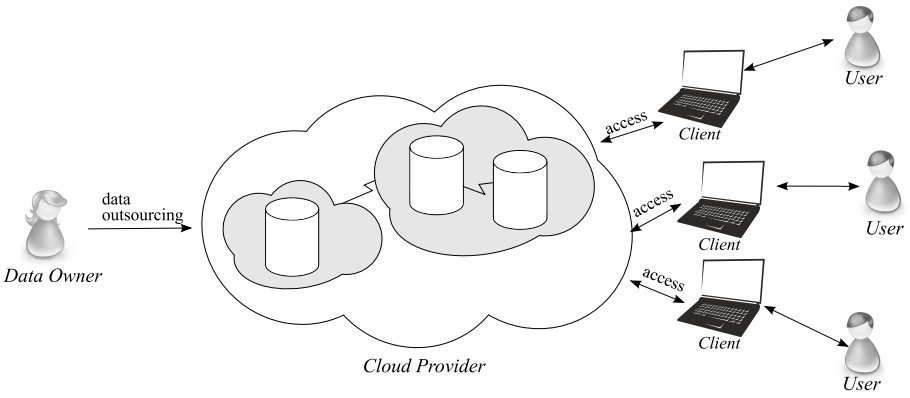


Fig. 1. Reference scenario

approaches enforcing selective access on encrypted data stored in the cloud. Section 4 illustrates approaches that enable the fine-grained access to encrypted outsourced data. Section 5 presents solutions for query privacy, focusing on techniques that protect the accesses to data. Section 6 discusses possible approaches to verify the integrity of query results. Finally, Section 7 gives our conclusions.

2 Protection of data in storage

When data are stored and managed by an external cloud provider, their confidentiality, integrity, and availability become of paramount importance. In this section, we illustrate the role of cryptographic techniques to ensure such properties. For simplicity, in the discussion we assume that outsourced data are organized in a relational database. We note however that all the approaches illustrated can be easily adapted to other data models.

2.1 Data confidentiality

When a data collection is outsourced to a cloud provider, its owner loses control over the data themselves, which should therefore be properly protected. The problem of protecting data when outsourcing them to external providers has been under the attention of the research community since the introduction of the Database-As-a-Service (DAS) paradigm [64]. Different approaches have been proposed to protect data confidentiality, typically relying on data encryption [64] to make data unintelligible to subjects who do not know the encryption keys.

Currently, there are two different approaches for dealing with encryption on the data outsourced to cloud providers: *1)* encryption is managed by the provider itself, which therefore encrypts the data with a key it knows; *2)* data are encrypted before sending them to the cloud provider, which does not know

the encryption key. While the first approach allows for enhanced functionality as data can be easily manipulated and managed by the provider with reduced overhead for data owners, it also implies granting to the provider full access to the data. There are however many scenarios where users might not fully trust cloud providers, which can be chosen based on factors other than security (e.g., economic reasons). Aiming at comprehensively protecting data confidentiality, encryption is typically applied before outsourcing data, so to protect them also against the cloud provider.

Data encryption can employ either symmetric or asymmetric encryption schemes. Many proposals adopt symmetric encryption, since it is cheaper than asymmetric encryption [64]. Regardless of the chosen encryption scheme, it is possible to encrypt data at different granularity levels: cell (each cell is singularly encrypted), tuple (all cells in a tuple of the relation are encrypted together), attribute (all cells in a column of the relation are encrypted together), or relation (the entire relation is encrypted as a single chunk). While the granularity at which encryption operates does not affect the confidentiality of the data, the majority of the existing approaches adopt tuple level encryption as it better supports query evaluation at the cloud provider (see Section 4). In fact, relation-level and attribute-level encryption require to communicate to the client issuing the query the entire relation or the subset of attributes involved in the query without the possibility of filtering at the provider side the encrypted tuples that are not of interest. On the other hand, cell-level encryption would require an excessive workload for data owners and clients in encrypting/decrypting data. Tuple-level encryption represents therefore a good tradeoff between encrypt/decrypt workload for clients and data owners, and query execution efficiency [64].

Adopting tuple level encryption, relation r, defined over relation schema $R(a_1, \ldots, a_n)$, is represented at the cloud provider as an encrypted relation r^k defined over schema $R^k(\underline{\texttt{tid}}, \texttt{enc})$, with `tid` the primary key added to the encrypted relation and `enc` the encrypted tuple. Each tuple t in r is represented as an encrypted tuple t^k in r^k, where $t^k[\texttt{tid}]$ is a random identifier and $t^k[\texttt{enc}]{=}E_k(t)$ is the encrypted tuple content, with E a symmetric encryption function with key k. Figure 2(a) illustrates relation MedicalData, storing medical information about eight patients of a hospital, and Figure 2(b) illustrates the corresponding encrypted relation.

The use of encryption to protect data confidentiality is based on the underlying assumption that all data are equally sensitive and therefore encryption is a price to be paid to protect them. However, this assumption can be an overkill in scenarios where data are not sensitive per se but what is sensitive is their association (e.g., the lists of patients' names and of their diseases in Figure 2(a) might not be sensitive, but the association of each patient's name with her disease should be protected). In these scenarios, encryption can be combined with data fragmentation to protect sensitive associations among attribute values [9,11]. Fragmentation consists in vertically partitioning the set of attributes in relation R in different (vertical) fragments, so that attributes forming a sensitive association are split among different fragments, and sensitive attributes are possibly

MedicalData

	SSN	Name	ZIP	Job	Disease
t_1	123456789	Alice	94110	nurse	asthma
t_2	234567891	Bob	94112	farmer	asthma
t_3	345678912	Carl	94118	teacher	gastritis
t_4	456789123	David	94110	teacher	chest pain
t_5	567891234	Eric	94112	surgeon	gastritis
t_6	678912345	Fred	94117	secretary	asthma
t_7	789123456	Greg	94115	manager	chest pain
t_8	891234567	Hal	94110	secretary	asthma

(a)

MedicalData^k

tid	enc
1	a%g6
2	1p(y
3	Hu8$
4	lR=+
5	kqW
6	nTy&
7	6_R&u
8	fp*r;

(b)

Fig. 2. An example of a relation (a) and corresponding encrypted version (b)

obfuscated (e.g., sensitive attributes are encrypted or not released). Different solutions have been proposed to define a correct fragmentation that minimizes query evaluation costs (e.g., [10,11,18]).

2.2 Data integrity and availability

Data integrity and availability are two critical elements that should be guaranteed when data are stored at an external cloud provider. Data integrity means that neither the cloud provider nor unauthorized parties can improperly tamper with data in storage without being detected. Like for confidentiality, also techniques that provide data integrity can operate at different granularity levels: cell, attribute, tuple, or relation level. Verifying integrity at the relation or at the attribute level, however, would require to access the entire relation (or column, respectively) for each integrity check. On the other hand, integrity verification at the cell level would require a considerable overhead for the client. To find a good tradeoff between integrity guarantees and the additional overhead for the client, the majority of the existing proposals operate at the tuple level. In the following, we illustrate some of the most well-known (encryption-based) techniques for ensuring data integrity and availability.

Digital and aggregate signatures. Data integrity can be ensured through *digital signatures* (e.g., [44]). Each data owner has its own pair ⟨*private_key*,*public_key*⟩ of private and public keys. Each tuple is first signed with the private key of its owner. The signature is then concatenated to the actual tuple, and this concatenated chunk is 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 this burden, multiple digital signatures (related to multiple tuples) can be combined in a single signature by adopting *condensed RSA*, *BGLS*, or

batch *DSA signature aggregation* [55]. Condensed RSA is an extension of the traditional RSA encryption scheme that permits to combine signatures generated by the same signer (i.e., signatures associated with tuples of the same owner). BGLS [6] is an encryption scheme based on bilinear mappings that supports the aggregation of signatures even when they have been generated by different signers (i.e., when the signatures have been generated by different owners). Batch DSA is an extension of traditional DSA signature schema that permits to combine the signature of different tuples, which can be verified together. 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 their composition, thus obtaining a valid signature that may correspond to the aggregate signature of an arbitrary set of tuples. This might represent a threat to the integrity guarantees of the cloud data collection.

POR-PDP. Encryption is also at the basis of Proof Of Retrievability (POR [50]) and Provable Data Possession (PDP [4]) proposals, which aim at ensuring data integrity and availability. These techniques allow a *verifier* (e.g., a requesting client or the data owner) to obtain a proof that the storage cloud provider is correctly maintaining a resource of interest (ensuring its integrity) and can therefore correctly return it (ensuring its availability). The main difference between POR and PDP is the mechanism used to obtain the proof. POR is based on the insertion in the data collection (before outsourcing) of ad-hoc random *sentinels* generated by the data owner, which are made indistinguishable from real data through a layer of encryption. In the verification step, the verifier challenges the provider by requesting some sentinel values. If the data collection has been tampered with by the provider or unauthorized parties, then these values will be incorrect with non-negligible probability, hence signaling that both data integrity and availability have been compromised. This basic technique has been extended along several directions to generate compact proofs to be returned to the data owner (or to an arbitrary verifier) [68]. PDP is based on ad-hoc *homomorphic verifiable tags*. The owner pre-computes a set of tags associated with the data items in her collection, combines the tags and the collection, and stores them at the cloud provider. In the verification step, the client challenges the provider against a randomly selected data item. The provider then generates a proof of possession for the required data item, using both the requested data and the corresponding tags, that the client can easily verify. It is interesting to note that, since tags enjoy the homomorphic property, tags computed for multiple data items can be combined into a single value [4]. We note that POR, whose security is based on the impossibility for the cloud provider to recognize sentinels, can only be employed to guarantee the integrity of encrypted data collections. On the contrary, PDP is more flexible and can be adopted with both encrypted and plaintext datasets.

Auditing. The aforementioned approaches require the client to check itself the integrity of a resource of interest. Aiming at reducing the burden at the client side, in some scenarios it might be desirable to delegate the verification process to a third party, trusted for enforcing integrity checks and to access the data content. The solution in [74] relies on the presence of a trusted auditor in charge of evaluating the integrity of a data collection stored at a cloud provider. Specific techniques (e.g., homomorphic linear authenticators and random masking) can be used if the auditor is not trusted to access the outsourced data collection [74]. Another solution relying on public auditing has been proposed in [86], and aims at increasing the performances of the auditing process.

3 Selective access to data

Data owners outsourcing their data to the cloud may wish to selectively make them visible/accessible to other users. Such a feature requires the support of access control correctly enforcing authorizations defined by the data owners (e.g., [16,24,30,37]). In a cloud scenario, neither the data owners (for performance reasons), nor the cloud providers (for security reasons) can however enforce such authorizations. A promising direction for solving this problem consists in making the outsourced data *self-enforce* the access restrictions [14,20,21]. In this section, we present two families of approaches specifically designed to enforce access control over outsourced data: selective encryption (Section 3.1) and attribute-based encryption (Section 3.2).

3.1 Selective encryption

Selective encryption consists in using different keys to encrypt different tuples, and in selectively distributing those keys to authorized users so that each user can decrypt all and only the tuples she is authorized to access.

Basic technique. The authorization policy, regulating which user in the set U of users of the system can read which tuple of relation r, can be represented as an *access matrix* M with a row for each user $u \in U$, and a column for each tuple $t \in r$, where: $M[u,t]=1$ iff u can access t; $M[u,t]=0$ otherwise. The j^{th} column of an access matrix represents the access control list $acl(t_j)$ of tuple t_j, for each $j = 1, \ldots, |r|$ (i.e., the set of users who can access it). Figure 3 illustrates an example of access matrix regulating access to the tuples in relation MedicalData in Figure 2(a) by users A, B, C, and D. According to the access matrix, $acl(t_1)=AC$.

Enforcing an access control policy with encryption requires to establish keys for encrypting resources and keys to be distributed to users. Equivalence among an *encryption policy* and an access control policy demands that every user should able to decrypt all and only the tuples she is entitled to access according to the access control policy.

There are different ways in which an access control policy can be translated into an equivalent encryption policy. However, this translation should take into

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
A	1	0	1	0	1	0	1	0
B	0	1	0	1	1	1	0	0
C	1	0	0	0	0	0	1	0
D	0	1	1	0	0	0	1	1

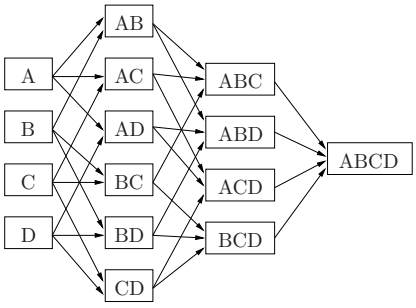
Fig. 3. An example of access matrix for the relation in Figure 2

account two main desiderata [21]: *i)* each user must manage only one key; and *ii)* each tuple must be encrypted with only one key (i.e., no tuple is replicated). These two desiderata are needed to reduce the overhead at user side caused by key management, and the consistency problems typically caused by data replication. To obey these two constraints, selective encryption approaches rely on *key derivation techniques*, which permit to compute an encryption key k_j starting from the knowledge of another key k_i, and of a piece of publicly available information. These techniques are based on the definition of a *key derivation hierarchy* that can be graphically represented as a directed graph with a vertex v_i for each key k_i in the system, and an edge (v_i,v_j) from key k_i to key k_j iff k_j can be directly derived from k_i. Key derivation can be recursively applied, meaning that a generic key k_j can be computed starting from another key k_i if there is a path, of arbitrary length, from vertex v_i to vertex v_j in the key derivation hierarchy. Depending on the kind of the key derivation hierarchy, different key derivation techniques can be applied, as illustrated in the following.

- *Chain of vertices* (e.g., [66]): the key k_j associated with vertex v_j is computed by applying a one-way function to key k_i associated with the predecessor vertex v_i of v_j in the chain. No public information is needed to derive keys.
- *Tree hierarchy* (e.g., [67]): the key k_j associated with vertex v_j is computed by applying a one-way function to key k_i of the direct ancestor of v_j, and a public label l_j associated with k_j. Public labels are necessary to guarantee that different children of the same node in the tree have different keys.
- *DAG hierarchy* (e.g., [2]): vertices in the hierarchy can have more than one direct ancestor, and each edge in the hierarchy is associated with a public *token* [3]. Given two keys k_i and k_j associated with vertices v_i and v_j such that (v_i,v_j) is an edge in the DAG, and the public label l_j of k_j, token $t_{i,j}$ permits to compute k_j from k_i and l_j. Token $t_{i,j}$ is computed as $t_{i,j}=k_j \oplus f(k_i,l_j)$, where $\oplus$ is the bitwise XOR operator, and f is a deterministic cryptographic function. By means of $t_{i,j}$, all users who know, or can derive, key k_i can also derive key k_j.

A key derivation hierarchy can be defined according to any of the above-mentioned models. In the following, we consider the most general case of a DAG, with token-based key derivation [21].

Enforcement of read privileges. A straightforward approach to define a key derivation hierarchy to enforce an access control policy consists in inserting a



(a)

user	key
A	k_A
B	k_B
C	k_C
D	k_D

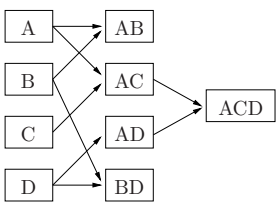
tuple	key
t_1	k_{AC}
t_2	k_{BD}
t_3	k_{AD}
t_4	k_B
t_5	k_{AB}
t_6	k_B
t_7	k_{ACD}
t_8	k_D

(b)

Fig. 4. An example of encryption policy equivalent to the access control policy in Figure 3: key derivation hierarchy (a) and user and tuple keys (b)

vertex in the hierarchy for each subset of users in U, and in exploiting the set containment relationship $\subseteq$ among these subsets to connect vertices. Given a pair of vertices v_i and v_j, there is a path from v_i to v_j iff the set of users represented by v_i is a subset of that represented by v_j. For instance, Figure 4(a) illustrates the key derivation hierarchy induced by the set U={A,B,C,D} of users and the set containment relationship over it. In the figure, vertices are labeled with the set of users they represent. The encryption policy induced by such a hierarchy is equivalent to (and thus, correctly enforces) the authorization policy iff: *i)* each user u_i is provided with the key associated with the vertex representing her; and *ii)* each tuple t_j is encrypted with the key of the vertex representing $acl(t_j)$. These encryption and key distribution strategies guarantee that each tuple can be decrypted by all and only the users in its access control list. Moreover, each user has to manage one key only, and each tuple is encrypted with one key only. With reference to the key derivation hierarchy in Figure 4(a) and the access control policy in Figure 3, Figure 4(b) illustrates the keys assigned to users and those used to encrypt the tuples in relation MedicalData in Figure 2. Note that the encryption policy in Figure 4 is equivalent to the authorization policy in Figure 3 as each user can derive, from her own key, the keys of the vertices representing sets of users including her, and hence can decrypt the tuples she is authorized to read. For instance, user C can derive the keys used to encrypt tuples t_1 and t_7.

While correctly enforcing the given authorization policy, the encryption policy illustrated above defines more keys and tokens than necessary. Managing a large set of tokens reduces the efficiency of the derivation process and, ultimately, increases the response time to users. In fact, tokens are stored in a publicly available catalog, maintained at the provider side: when a user u wants to access a tuple t, she needs to perform a search across the catalog to retrieve a chain of tokens that, starting from her own key, ends in the one used to encrypt t. The total number of tokens is therefore a critical factor for the efficiency of



(a)

user	key
A	k_A
B	k_B
C	k_C
D	k_D

tuple	key
t_1	k_{AC}
t_2	k_{BD}
t_3	k_{AD}
t_4	k_B
t_5	k_{AB}
t_6	k_B
t_7	k_{ACD}
t_8	k_D

(b)

Fig. 5. An example of encryption policy equivalent to the access control policy in Figure 3 with reduced number of tokens: key derivation hierarchy (a) and user and tuple keys (b)

access to remotely stored data. The problem of minimizing the number of tokens in the key derivation hierarchy while still guaranteeing equivalence between the authorization and the encryption policies is NP-hard as it can be reduced to the set cover problem [21]. In [21], the authors present a heuristic approach to reduce the number of tokens that is based on the following two observations:

- the vertices necessary to enforce an authorization policy are those vertices, called *material*, that represent singleton sets of users (whose keys are communicated to users) and the access control lists of the tuples in r (whose keys are used for encryption);
- when two or more vertices have more than two common direct ancestors, the insertion of a vertex representing the set of users in these ancestors reduces the total number of tokens.

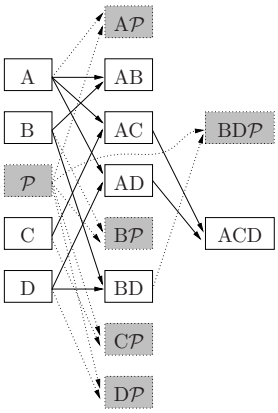
Given an authorization policy, the heuristics first identifies the material vertices and, for each vertex v, finds a set of material vertices that form a *non-redundant set covering* for v (i.e., the smallest set V of vertices such that, for each user u represented by v, there is at least a vertex v_i in V such that u appears in v_i), which become direct ancestors of v. For each set $\{v_1, \ldots, v_m\}$ of vertices that have $n > 2$ common ancestors $v'_1, \ldots, v'_n$, the algorithm inserts an intermediate vertex v representing all the users in $v'_1, \ldots, v'_n$, connects each v'_i, $i = 1, \ldots, n$, with v, and v with each v_j, $j = 1, \ldots, m$. In this way, the encryption policy includes $n + m$, instead of $n \cdot m$ tokens in the catalog [21]. Figure 5 illustrates an encryption policy equivalent to the authorization policy in Figure 3 with a reduced number of tokens. Comparing the key derivation hierarchy in Figure 5(a) with the one in Figure 4(a) it is easy to see the reduction in the number of tokens needed to correctly enforce the access control policy.

Enforcement of write privileges. The approach in [21] assumes outsourced data to be read-only, meaning that only the data owner can update the content of her tuples while other parties can only be granted read privileges over them. This

assumption is not aligned with current trends in technology (e.g., collaborative scenarios), where the data owner might want to selectively grant to other users also write privileges over her resources. The proposal in [17] adopts selective encryption to manage also write authorizations. The basic idea is to associate each tuple with an encrypted *write tag* (i.e., a random value chosen independently from the tuple content), and to allow the update of a tuple t only to users who know the plaintext value of the write tag of t. Access to write tags is regulated through selective encryption: the write tag of tuple t is encrypted with a key derivable only by the users authorized to write t (i.e., the users specified within its write access control list) and the provider. The provider will accept a write request on a tuple only if the requesting user proves to know the corresponding write tag. To this aim, the key derivation hierarchy is extended with the keys used to encrypt write tags and with key $k_{\mathcal{P}}$, specifically assigned to the provider $\mathcal{P}$ to enable write tags verification.

The keys used to encrypt write tags are defined in such a way that: *i)* authorized users can compute them applying a *secure hash function* to a key they already know (or can derive via a sequence of tokens); and *ii)* the provider can directly derive them from key $k_{\mathcal{P}}$ through a token specifically added to the key derivation hierarchy. Note that keys used to encrypt write tags cannot be used to derive other keys in the hierarchy, because the provider is not trusted to access the plaintext content of the tuples in the outsourced relation. For instance, consider the encryption policy in Figure 5 and suppose that the write privilege over tuple t_1 is granted to user A, over t_2 to B and D , over t_3 and t_8 to D, over t_4, t_5, and t_6 to B, and over t_7 to C. Figure 6 illustrates the key derivation hierarchy extended with the key $k_{\mathcal{P}}$ of the provider and the keys necessary to encrypt write tags. In the figure, the additional vertices are in gray, and both additional vertices and edges are dotted. Figure 6(b) reports the keys assigned to users and to the provider, and the keys used to encrypt the tuples in relation MedicalData and their write tags.

Updates to the authorization policy. Since the equivalence between the authorization and encryption policies must be always guaranteed to ensure proper enforcement of authorizations, any change in the access control policy should be enforced by updating the encryption policy. In fact, the keys used to encrypt each tuple t and its write tag depend on the set of users who can read and write it, respectively. To enforce updates to read privileges, it is then necessary to re-encrypt the tuple involved in a policy update with a different key that only the users in its new access control list know or can derive. The overhead for the data owner in executing re-encryption operations is reduced in [21] by introducing the *over-encryption* approach to partially delegate to the provider the management of grant and revoke of read privileges, thus greatly reducing the overhead at the data owner side. Over-encryption adopts two different layers of encryption: the *Base Encryption Layer* (BEL) and the *Surface Encryption Layer* (SEL), each of which is characterized by its own encryption policy (i.e., set of keys, key derivation hierarchy, and key distribution). Each tuple t is protected with two different layers of encryption, and then a user can access t only if she



(a)

user	key
A	k_A
B	k_B
C	k_C
D	k_D
$\mathcal{P}$	$k_{\mathcal{P}}$

tuple	read_key	write_key
t_1	k_{AC}	$k_{A\mathcal{P}}$
t_2	k_{BD}	$k_{BD\mathcal{P}}$
t_3	k_{AD}	$k_{D\mathcal{P}}$
t_4	k_B	$k_{B\mathcal{P}}$
t_5	k_{AB}	$k_{B\mathcal{P}}$
t_6	k_B	$k_{B\mathcal{P}}$
t_7	k_{ACD}	$k_{C\mathcal{P}}$
t_8	k_D	$k_{D\mathcal{P}}$

(b)

Fig. 6. Encryption policy in Figure 5 extended to enforce write privileges: key derivation hierarchy (a) and user and tuple keys (b)

knows both the keys used to encrypt t at BEL and SEL. At initialization time, the encryption policies at BEL and SEL coincide (more precisely, they are both equivalent to the initial authorization policy). In case of policy updates, BEL is updated by only inserting tokens (to allow for new key derivations) in the public catalog. Re-encryption is instead performed at the SEL by the cloud provider.

While effective for updates to the read authorization policy, the over-encryption approach cannot be adopted in case of updates to write privileges. In fact, users are not oblivious, and adding a layer of encryption to a write tag would not prevent a user, whose write privilege on a tuple has been revoked, from exploiting their previous knowledge of the tag of the tuple to perform unauthorized updates. Indeed, to *grant* user u write access to tuple t, the write tag of t can be simply re-encrypted with a key known to the provider and the users authorized to update its content. On the contrary, to *revoke* from user u write access to tuple t, it is necessary to associate with t a fresh write tag, with a new plaintext value independent from the previous one, and to encrypt it with a key known to the provider and the users in the new write access control list of the tuple [17]. Note that, since the provider knows the write tag of each tuple to correctly enforce write privileges, the data owner can delegate to the storing provider both the generation and re-encryption of the write tag of her tuples [17].

3.2 Attribute-based encryption

Another approach to enforce selective access in cloud scenarios is represented by *Attribute-Based Encryption* (ABE [43]).

Basic technique and authorization enforcement. ABE is based on public-key encryption schemes, and enforces access restrictions according to an autho-

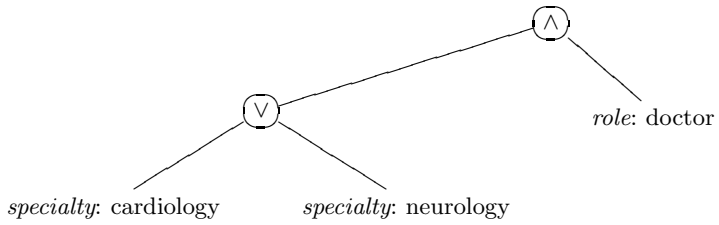


Fig. 7. Access structure associated with tuple t_7 of relation MEDICALDATA in Figure 2 with CP-ABE

rization policy defined on attributes associated with tuples or with users. Based on how attributes and policies are associated with data and users, it is possible to implement ABE as either *Ciphertext-Policy ABE* (CP-ABE [77]) or *Key-Policy ABE* (KP-ABE [43]). In the following, we briefly describe these two approaches.

CP-ABE associates each user u with a set of descriptive attributes and a private key, generated on the basis of these attributes. The attributes associated with u describe her characteristics considered relevant for access control enforcement (e.g., her role and department in a company). Each tuple t in a relation r is instead associated with an *access structure* modeling the authorization policy regulating accesses to t. Graphically, an access structure is a tree whose leaves represent basic conditions over attributes, and whose internal nodes represent logic gates (i.e., conjunctions and disjunctions). For instance, suppose that the access to tuple t_7 in relation MEDICALDATA in Figure 2 should be granted only to doctors specialized in cardiology or neurology. Figure 7 illustrates the access structure associated with tuple t_7, representing the Boolean formula (*role*='doctor') $\wedge$ (*specialty*='cardiology' $\vee$ *specialty*='neurology'). The key generation technique adopted by CP-ABE is specifically designed to guarantee that the key k of user u can decrypt tuple t iff the set of attributes used when generating k satisfies the access control policy represented by the access structure considered when encrypting t.

KP-ABE associates each user u with an access structure and each tuple with a set of attributes describing its characteristics. The key associated with each user is then generated on the basis of her access structure, while the key used to encrypt each tuple depends on its attributes. The key generation technique adopted by KP-ABE is specifically designed to guarantee that each user u can decrypt a tuple t iff the attributes associated with t satisfy the access structure associated with user u.

The support of write privileges is provided by the adoption of Attribute-Based Signature (ABS) techniques. The proposal in [35] combines CP-ABE and ABS techniques to enforce read and write access privileges, respectively. This approach, although effective, has the disadvantage of requiring the presence of a trusted party for correct policy enforcement. A similar approach, based on the combined use of ABE and ABS for supporting both read and write privileges, is

illustrated in [62]. This solution has the advantage over the approach in [35] of being applicable also to distributed scenarios.

Updates to the authorization policy. Although CP-ABE effectively and efficiently enforces access control policies, one of its main drawbacks is related to the management of attribute revocation. When a user loses one of her attributes, she should not be able to access tuples that require the revoked attribute for the access. Attribute revocation is however hard to enforce without causing expensive re-keying and/or re-encryption operations. Solutions to this problem are presented in [72,82,85]. In [82] the authors illustrate an encryption scheme able to manage attribute revocation, ensuring the satisfaction of both backward security (i.e., a user cannot decrypt the tuples requiring the revoked attributes) and forward security (i.e., a new user can access all the tuples outsourced before her join, provided her attributes satisfy the access control policy). In [72] the authors propose a hierarchical attribute-based solution that relies on an extended version of CP-ABE where attributes associated with users are organized in a recursive set structure. Aiming at enforcing updates in the context of KP-ABE, the solution in [85] proposes to couple ABE with proxy re-encryption, in such a way to delegate to the storage provider most of the re-encryption operations necessary to enforce attribute revocation. To reduce the overhead inevitably caused by the adoption of asymmetric encryption, this approach also proposes to adopt KP-ABE to protect the symmetric keys used to encrypt tuple contents. By doing so, only authorized users can retrieve the key physically used to protect the content of the tuples.

4 Fine-grained access to data

Encryption represents an effective means to protect data confidentiality in the cloud. However, cloud providers cannot directly evaluate users' queries on the data they store, as they do not know the encryption keys and therefore cannot access data content. It is also infeasible to require the client to download the encrypted data collection and locally evaluate the queries, as this would nullify the benefits of delegating data storage to cloud providers. Current solutions addressing this issue are based on the definition of *indexes* that enable (partial) query evaluation at the provider side without the need to decrypt data [64], or on specific encryption schemas that support the execution of operations (Figure 8) or SQL queries [60] directly over encrypted data. In the remainder of this section, we describe these two solutions in more details.

4.1 Indexes for query execution

Indexes are metadata whose values depend on the plaintext values of the attributes in the original relation on which they are defined. Indexes are represented in the encrypted relation as additional attributes. Given a relation r, defined over schema $R(a_1, \ldots, a_n)$, the corresponding encrypted and indexed relation r^k is defined over schema $R^k(\underline{\texttt{tid}}, \texttt{enc}, I_{i_1}, \ldots, I_{i_j})$, where $I_{i_l}, l = 1, \ldots, j$,

Encryption	Operations	Security	Cost
Randomized	anything	no leakage	practical
Deterministic	=	leaks duplicates	practical
OPE	$\geq$	leaks order	practical
Pallier	+	no leakage	expensive
El Gamal	$\times$	no leakage	expensive
Fully homomorphic	everything	no leakage	impractical

Fig. 8. Characteristics of some encryption functions

MedicalData

	SSN	Name	ZIP	Job	Disease
t_1	123456789	Alice	94110	nurse	asthma
t_2	234567891	Bob	94112	farmer	asthma
t_3	345678912	Carl	94118	teacher	gastritis
t_4	456789123	David	94110	teacher	chest pain
t_5	567891234	Eric	94112	surgeon	gastritis
t_6	678912345	Fred	94117	secretary	asthma
t_7	789123456	Greg	94115	manager	chest pain
t_8	891234567	Hal	94110	secretary	asthma

(a)

MedicalData^k

tid	enc	I_Z	I_J	I_D
1	a%g6	α	η	κ
2	1p(y	β	ζ	κ
3	Hu8$	γ	θ	λ
4	lR=+	α	θ	λ
5	kqW	β	θ	λ
6	nTy&	δ	η	κ
7	6_R&u	ϵ	ζ	λ
8	fp*r;	α	η	κ

(b)

Fig. 9. Plaintext relation MedicalData (a) and corresponding encrypted and indexed relation (b)

is the index defined over attribute a_{i_l} in R. Note that not all the attributes in R need to have a corresponding index in R^k, but only those that are expected to be involved in queries. For instance, Figure 9(b) represents the encrypted version of relation MedicalData in Figure 2(b), reported also in Figure 9(a) for the reader's convenience, where attributes `ZIP`, `Job`, and `Disease` have been associated with indexes I_Z, I_J, and I_D, respectively. Index values are denoted with Greek letters.

The introduction of indexes allows the cloud provider to (partially) evaluate a query q submitted by the client. The query evaluation process in presence of indexes operates as follows.

- *Step 1.* The user formulates a query q that is sent to the client. Note that, since encryption must be transparent for final users (which could be unaware of the fact that the relation is stored in encrypted form at the cloud provider), q is formulated over the plaintext relation.
- *Step 2.* Upon receiving q, the client generates two queries: q_p, operating on the encrypted relation at the provider using indexes; and q_c, operating on the result of q_p at the client. Query q_p is then communicated to the cloud provider.
- *Step 3.* Upon receiving q_p, the cloud provider executes it on the encrypted relation. The result is then sent to the client.

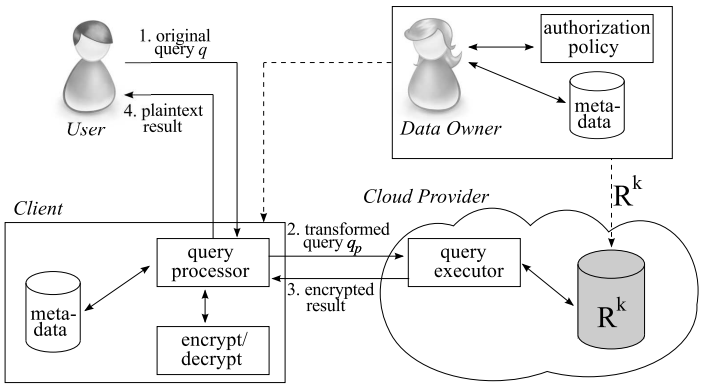


Fig. 10. Query evaluation process

– *Step 4.* The client decrypts the result obtained from the provider, and evaluates q_c on the resulting relation to possibly remove *spurious tuples* (i.e., tuples that satisfy the condition on the index but not the original condition specified by the user) and returns the query result to the user.

Figure 10 illustrates the query evaluation process. Clearly, the translation of query q into queries q_p and q_c depends on the kind of indexes involved in the query. We now illustrate some of the most well-known indexing techniques, classified according to the conditions they support.

Equality conditions (e.g., [15,46]). Equality conditions are conditions of the form $a = v$, with a an attribute and v a value in the domain of a, and are supported by three classes of indexes: *encryption-based* [15], *bucket-based* [46], and *hash-based* [15] indexes.

The encryption-based index for a tuple t over attribute a is computed as $E_k(t[a])$, where E_k is a symmetric encryption function and k the encryption key. An equality condition of the form $a{=}v$ is then translated as $I{=}E_k(v)$. For instance, suppose that index I_Z in Figure 9(b) is an encryption-based index of attribute `ZIP`. Equality condition `ZIP` = '94110' on relation MedicalData is then translated into $I_Z{=}$'α' operating on indexed relation MedicalData^k.

The definition of a bucket-based index over attribute a requires instead to partition the domain of a into non-overlapping subsets of contiguous values, and to associate each partition with a label. Given a tuple t in the outsourced relation r, the value of the index associated with attribute a is the label of the partition containing value $t[a]$. An equality condition of the form $a{=}v$ is therefore translated as $I{=}l$, where l is the label of the partition including v. For instance, suppose that index I_J in Figure 9(b) is a bucket-based index where ζ, η, and θ are the labels of partitions {*farmer*,*manager*}, {*nurse*,*secretary*}, and {*surgeon*,*teacher*}, respectively. Equality condition `Job` = '*farmer*' on relation

MEDICALDATA is then translated as I_J = 'ζ' operating on indexed relation MEDICALDATA^k.

The definition of a hash-based index over attribute a is based on the adoption of a deterministic hash function h that generates collisions. Given a tuple t in r, the value of the index associated with attribute a is computed as $h(t[a])$. An equality condition of the form $a{=}v$ is therefore translated as $I{=}h(v)$. For instance, suppose that index I_D in Figure 9(b) is a hash-based index computed using function h such that $h(asthma){=}\kappa$ and $h(gastritis){=}h(chest\ pain){=}\lambda$. Equality condition `Disease` = '*gastritis*' on relation MEDICALDATA is then translated as I_D = 'λ' operating on indexed relation MEDICALDATA^k.

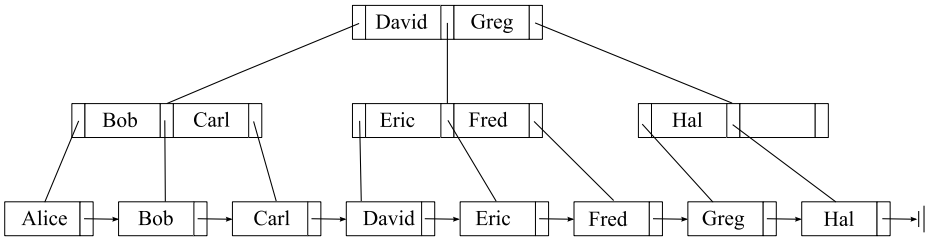
Note that, differently from encryption-based indexes, both bucket-based and hash-based indexes map different plaintext values to the same index value. Therefore, the result computed by the provider from the evaluation of an equality condition can include spurious tuples that the client must filter out to obtain the final query result.

Range conditions (e.g., [1,15,75]). Range conditions are conditions of the form a IN $[v_1, v_2]$, with a an attribute and $[v_1, v_2]$ a range in the domain of a. Bucket-based indexes can support range queries, provided that labels are defined so to preserve the ordering among attribute values. This solution would however leak the order of attribute values to the provider. An alternative solution specifically designed to support equality and range conditions is based on the definition of a B+-tree index over the indexed attribute [15]. The B+-tree index is built over the plaintext values of the attribute, and is represented at the provider as an encrypted relation with two attributes: `id`, containing the node identifier, and `content`, containing the encrypted node content. Pointers to children are represented through node identifiers.

Figure 11 illustrates an example of a B+-tree built over attribute `Name` of relation MEDICALDATA in Figure 2(a). To retrieve the tuples satisfying a range condition, the client iteratively queries the encrypted relation representing the B+-tree at the provider. The client will then perform a sequence of queries to retrieve at each level, starting from the root, the node along the path to the leaf of interest. For instance, with reference to the example in Figure 11, to retrieve patients whose name is between E and G, the client accesses tuples 1, 3, 9, and 10, in the order, in the encrypted relation.

An alternative technique for supporting range conditions relies on *Order Preserving Encryption Schemas* (OPES [1]) or on *Order Preserving Encryption with Splitting and Scaling* schemas (OPESS [75]). OPES is an encryption technique that takes as input a target distribution of index values, and applies an order preserving transformation guaranteeing that the index values follow the target distribution. OPESS guarantees instead that the produced index values follow a flat frequency distribution. This is obtained by mapping the same plaintext value to multiple index values. Since index values preserve ordering, range conditions can be directly evaluated by the provider over indexes.

Aggregate operators (e.g., [38,45]). To compute aggregate functions (such as SUM and AVG), it is necessary to use indexes that support arithmetic operations,



(a)

id_node	node
1	2, David, 3, Greg, 4
2	5, Bob, 6, Carl, 7
3	8, Eric, 9, Fred, 10
4	11, Hal, 11
5	Alice, 6, t_1
6	Bob, 7, t_2
7	Carl, 8, t_3
8	David, 9, t_4
9	Eric, 10, t_5
10	Fred, 11, t_6
11	Greg, 12, t_7
12	Hal, NIL, t_8

(b)

id	content
1	8/*5sym,p
2	mw39wio[
3	gtem945/*c
4	21!p8dq59
5	8dq59wq*d'
6	ue63/)w
7	=wco21!ps
8	oieb5(p8*
9	gte3/)8*
10	rfoi7/(
11	=o54'?c
12	Fer3!-r

(c)

Fig. 11. An example of B+tree index (a), its relational representation (b), and the corresponding encrypted relation (c)

which are defined adopting homomorphic encryption [61], a particular encryption scheme that permits the evaluation of basic arithmetic operations (i.e., $+$, $-$, $\times$). These indexes can therefore be used by the provider to evaluate aggregate functions, as well as equality and range conditions [45]. A *fully* homomorphic encryption scheme (where *fully* means that the homomorphic property remains valid for any operation computed over the encrypted data) has been proposed and studied in [7,38]. This solution allows the computation of an arbitrary function over encrypted data without the need of decryption. Unfortunately, this technique suffers from high computational complexity, which makes it not suitable for real-world scenarios. In [12,13] the authors propose a fully homomorphic scheme enforceable with smaller public keys, hence more manageable and efficient than traditional ones.

4.2 CryptDB

CryptDB [60] supports query execution at the cloud provider directly over encrypted data, without the need of indexes associated with the outsourced relation. To this aim, CryptDB adopts for each attribute different kinds of encryp-

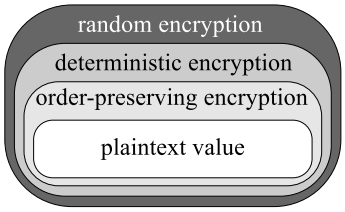


Fig. 12. An example of encryption layers adopted by CryptDB [60]

tion (i.e., random, deterministic, order-preserving, homomorphic, join, order-preserving join, and word search [60]), which are dynamically adjusted depending on the queries that need to be executed. Each cell in the outsourced relation is then wrapped in multiple encryption layers, forming an onion structure, in such a way that the same attribute value is encrypted multiple times to obtain the value stored at the provider. Note that the encryption layers are the same for all the cells in the same column, but they may vary from an attribute to another (depending on the kinds of queries to be supported). Figure 12 illustrates an example of the onion encryption structure wrapped around a plaintext data item. The outermost level features the strongest encryption (i.e., *random encryption*, a probabilistic scheme where two equal values can be mapped to different ciphertexts with non-negligible probability [60]), while the innermost level represents plaintext data. Proceeding through the innermost level, the adopted encryption scheme provides weaker security guarantees but supports more computations over the encrypted data.

CryptDB proposes to dynamically regulate the usage of encryption, possibly removing some of the encryption layers, depending on the operations in the query to be evaluated. The adjustments in the encryption layers is dynamic, that is, it depends on the specific query being evaluated. For instance, if the provider needs to perform a GROUP BY on attribute a, then it should be able to determine which values of a are equal to each other, but without discovering the plaintext values of a. Since random encryption does not support such a functionality, it is removed, leaving data encrypted with a deterministic scheme. As this latter scheme supports grouping operations, it is then not necessary to further peel it out. Note that once a layer of encryption is removed from an attribute, it cannot be restored as data have been exposed to the provider.

Query execution with CryptDB assumes a trusted proxy intercepting all communications between users and the cloud provider. The proxy stores a secret master key k, the database schema, and the current encryption layers of each attribute in the relation. The query evaluation process operates as follows.

– *Step 1.* The user formulates a query q that is sent to the proxy, which rewrites it into an equivalent query $\hat{q}$ operating over the encrypted version of the attributes involved. The proxy then, with its own key k, encrypts

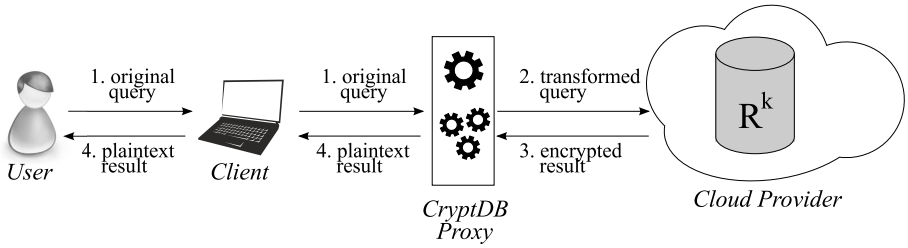


Fig. 13. Query processing in CryptDB

all constant values in q adopting the encryption scheme that best suits the operation to be computed.
- *Step 2.* The proxy checks if the provider should be given keys to remove some of the encryption layers before executing the query $\hat{q}$, and if so, issues an UPDATE query that removes specific layers of encryption for the attributes of interest. The proxy forwards $\hat{q}$ to the cloud provider, which executes it.
- *Step 3.* The provider returns the encrypted result of $\hat{q}$ to the proxy.
- *Step 4.* The proxy decrypts the received result of $\hat{q}$ and sends it to the user.

Figure 13 illustrates the query evaluation process in CryptDB.

5 Protecting query confidentiality

When a user submits a query to a cloud provider, her privacy (and also the privacy of accessed data) can be put at risk due to the knowledge of the query itself [25,31,79]. For instance, knowing that a user submitted a query to an outsourced medical database looking for the symptoms of liver cancer can implicitly reveal that either her or a person close to her suffers from such a disease. Also, it might be possible to analyze the data accesses performed by users to infer the (private) content of the outsourced data collection. For instance, by monitoring patterns of frequently accessed tuples, an observer can draw inferences on their values if she knows how frequently the values in the considered data domain are accessed. To counteract these privacy risks, query confidentiality must be properly protected. Protecting query confidentiality requires ensuring both *access* and *pattern* confidentiality, which consist in protecting the target of an access and the fact that two accesses aim at the same target, respectively.

Traditionally, access and pattern confidentiality have been addressed through Private Information Retrieval (PIR) techniques. These approaches however do not protect the confidentiality of accessed data and are characterized by high computational costs (e.g., [8,56]). Several solutions have been proposed to protect data and access confidentiality (e.g., [39,59,69,73]), but they fall short in protecting pattern confidentiality. In the remainder of this section, we illustrate recent techniques that protect data, access, and pattern confidentiality. The basic idea behind such solutions is to break the otherwise static association between

disk blocks and the information they store, by adopting *dynamically allocated data structures* [52,83].

Oblivious RAM (ORAM). The *Oblivious RAM* (ORAM) [40] data structure is at the basis of several approaches that aim at protecting access and pattern confidentiality in encrypted data collections. With ORAM, the encrypted data are organized as a set of n encrypted blocks, stored in a pyramid-shaped data structure. Each level l of the ORAM structure stores 4^l blocks and is associated with a Bloom filter and a hash table to determine whether a given block is stored at level l and, if this is the case, to identify the block where it is stored [79]. During the search process, the ORAM structure is visited level by level from the top of the pyramid. At each level, one element is extracted (the target of the access or a random element, if the target does not belong to the visited level) and placed in a cache. Note that the visit does not terminate when the target block is found to not reveal any information to the cloud provider. When the cache is full, it is merged with the first level of the ORAM and all elements are then shuffled (i.e., allocated to a different physical block on the provider's disk) to destroy any correspondence between old and new data items. Analogously, when the first (i-th in general) level is full, it is merged with the second ($i+1$-th, in general) one and their elements are shuffled.

While ORAM effectively guarantees access and pattern confidentiality, the re-organization of the lower levels of the pyramid is highly expensive. Access requests submitted during the reordering of lower levels of the database might therefore suffer from a high response time. To mitigate such cost, the proposal in [34] limits the shuffling operation to the blocks that store accessed tuples. Most ORAM solutions rely on the presence of a secure coprocessor operating at the provider side. This assumption however may not be viable in many real world scenarios. Alternative solutions for reducing access times are based on the idea of minimizing the number of interactions between the client and the provider [41,78], or support concurrent accesses [42,80].

Path-ORAM is a recent enhancement of the traditional ORAM structure, which reduces the overhead due to the re-organization of lower layers in the ORAM structure [70,71]. Path-ORAM proposes to organize data in a tree, whose nodes are buckets storing a fixed number of blocks that can contain either dummy or real tuples. Each block is mapped to a random leaf, and stored either at the client side (in a local cache called *stash*), or in one of the buckets along the path to the leaf with which it is associated. Read operations download from the provider and store in the stash all the buckets along the path from the root to the leaf to which the tuple of interest is mapped. The mapping of the target tuple is then changed randomly, choosing a new leaf in the tree. The accessed path is then written back, possibly inserting into the written block some of the tuples in the local stash. A tuple can be inserted into a block if such a block is along the path to the leaf to which it is mapped and it is not full. In the insertion of a tuple into a block, Path-ORAM privileges blocks close to the leaf to which the tuple is mapped.

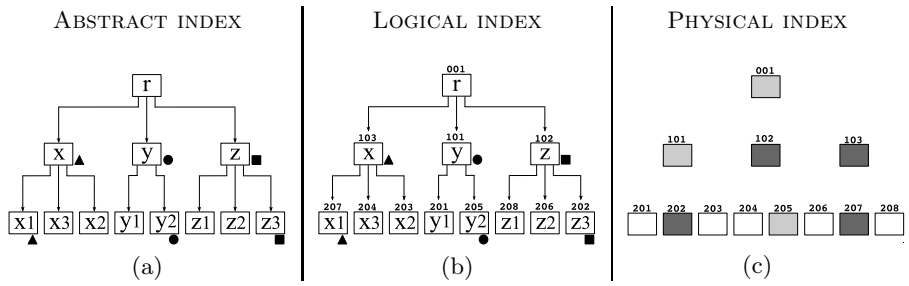


Fig. 14. An example of abstract (a), logical (b), and physical (c) shuffle index
Legend: ■ target, ● node in cache, ▲ cover; blocks read and written: dark gray filling, blocks written: light gray filling

Shuffle index. An efficient technique recently proposed to protect both access and pattern confidentiality is based on the definition of a *shuffle index* [25]. The shuffle index is a privacy-preserving indexing technique used for organizing data in storage and for efficiently executing users' queries. It can be seen at three abstraction levels: abstract, logical, and physical. At the *abstract level*, the shuffle index is an *unchained B+-tree* with fan-out F, built over a candidate key K of the indexed relation. Each internal node in the tree has $q \geq \lceil F/2 \rceil$ children (except for the root node, where $1 \leq q \leq F$), and stores $q-1$ ordered key values $val_1 \leq \ldots \leq val_{q-1}$. The i-th child of a node represents the root of the subtree storing all the values between val_i and val_{i+1}. The leaves store the actual tuples together with their key values. Unlike traditional $B+$-tree structures, leaves are not connected in a chain (to hide the relative value order). Figure 14(a) illustrates an example of unchained $B+$-tree with fan-out 3.

At the *logical level*, each abstract node n is represented by a pair $\langle id, n \rangle$ where id is the *logical identifier* associated with the node and n is its content. Pointers to children of internal nodes of the abstract data structure are represented through node identifiers. Figure 14(b) illustrates an example of logical representation of the abstract index in Figure 14(a). Note that the order of logical identifiers does not necessarily reflect the value-order relationship between the node contents. For readability, in the figure logical identifier are reported on the top of each node, and their first digit corresponds to the level of the node in the tree. Finally, at the *physical level*, each logical node $\langle id, n \rangle$ is concatenated with a random salt, to destroy plaintext distinguishability, and then encrypted in CBC mode, using a symmetric encryption algorithm. The logical identifier of the node easily translates into the physical address where the block representing the encrypted node is stored at the provider. Figure 14(c) illustrates an example of physical representation of the logical index in Figure 14(b), which corresponds to the view of the cloud provider.

Protection of access and pattern confidentiality is provided by the combined adoption of the following three protection techniques.

- *Cover searches.* Cover searches are *fake* searches, not recognizable as such by the provider, executed in conjunction with the actual search for the target value. For each level of the shuffle index (but the root level) the client downloads $num\_cover + 1$ blocks: one for the node along the path to the target, and *num_cover* for the nodes along the paths to the covers. Hence, from the provider point of view, each of the $num\_cover + 1$ accessed leaf blocks has the same probability of storing the target. Cover searches must guarantee both *indistinguishability* with respect to target searches (i.e., the provider should not be able to determine whether an accessed block is a cover or the target) and *block diversity* (i.e., paths to covers and to the target must be disjoint, except for the root).
- *Cached searches.* Cached searches make repeated accesses to a node content *indistinguishable* from non-repeated accesses. The cache is a layered structure, with a layer for each level in the shuffle index. It is maintained plaintext at the client side and stores the nodes along the paths to the targets of the *num_cache* most recent accesses to the shuffle index. Each layer of the cache is managed according to the Least Recently Used (LRU) policy: in this way, the parent of each cached node (and hence the path connecting it to the root of the tree) is also in cache. Whenever the target of an access is in cache, an additional cover is used during the access, to guarantee that $num\_cover + 1$ blocks are downloaded for each level of the tree (but the root level). The adoption of a local cache prevents short-time intersection attacks, which could be exploited by the provider to identify repeated accesses when subsequent searches download non-disjoint sets of blocks.
- *Shuffling.* Shuffling breaks the relationship between a block and the content of the node it stores. In this way, accesses to the same physical block may not correspond to accesses to the same node content. Shuffling consists in *moving* the content of accessed (either as target or as covers) and cached nodes to different blocks. Shuffling then assigns a different block to each accessed node, choosing among the downloaded blocks. To prevent the provider from inferring information about shuffling, every time a node is moved to a different block, it is re-encrypted using a different random salt. The parent of a shuffled node is updated to preserve the consistency of the structure.

The search process, operating at the client side, visits the $B+$-tree of the shuffle index level-by-level, from the root to the leaves. Each access combines the three protection techniques illustrated above, and the search process is guaranteed to protect both access and pattern confidentiality [25]. As an example of access to the shuffle index in Figure 14, consider a search for $z3$, and suppose that the adopted cover is $x1$, and that the cache contains the path to value $y2$. Since the client has the root r in cache, it first downloads from the provider the blocks at level 1 along the paths to $x1$ (block 103 storing value x) and to $z3$ (block 102 storing value z). It then decrypts and shuffles the accessed and cached nodes at level 1 allocating, for example, x to block 102, y to 101, and z to 103. As a consequence of the shuffling, the client updates the root node, encrypts its content and writes it back at the provider. It then updates the cache

at level 1 inserting node z. The client then downloads and decrypts the blocks at level 2 along the path to $z3$ and $x1$ (202 and 207, respectively). It decrypts these blocks retrieving the target of the search, and shuffles their content along with node $y2$ (205) in the cache. The client updates the content of nodes x, y, and z according to the shuffling, re-encrypts them and writes them back to the provider. Analogously, it encrypts and writes at the provider blocks 202, 205, and 207. Also, it inserts node $z3$ in the cache. Figure 14(c) illustrates the cloud provider's view over the access in terms of blocks read and/or written. It is easy to note that the provider can detect neither which among the accessed leaves is the target of the access, nor how the block contents have been shuffled [25].

The original shuffle index proposal has been extended to support concurrent accesses to the data, accesses to attributes different from the key (e.g., [27]), and to operate in a distributed system (e.g., [26,28]).

6 Protecting query integrity

Another important issue that needs to be considered when storing and processing data in the cloud is the ability of users to verify the correct behavior of cloud providers. This implies providing users with techniques that allow them to check the *correctness*, *completeness*, and *freshness* of query results. Correctness means that the result has been performed on the original data and the computation performed correctly. Completeness means that no tuple is missing from the query result. Freshness means that the query result has been computed on the most up-to-date version of the data. Two classes of techniques have been proposed to provide such guarantees: *deterministic* techniques (Section 6.1) and *probabilistic* techniques (Section 6.2).

6.1 Deterministic approaches

Deterministic approaches are typically based on the adoption of *authenticated data structures* such as signature chaining, Merkle hash trees, and skip lists (e.g., [32,51,54,57,58,84]). These solutions build an authenticated data structure on the outsourced dataset and return, for each query q, a *verification object* VO extracted from the structure that can be used for verification. If VO is consistent with the data structure, this guarantees that the query result is correct and complete. Since they are defined over the whole data collection, authenticated data structures also provide integrity of data in storage, as unauthorized modifications can be immediately detected when checking the integrity of query results.

Signature chaining. Signature chaining has been originally proposed to verify the integrity of the result of *range* queries [57] operating on an attribute a of the outsourced relation r, defined over domain D, and characterized by a total order relationship. These techniques adopt a one-way hash function h, and require to order the tuples in r according to the values of attribute a. The signature associated with each tuple t_i is computed by signing the string resulting from

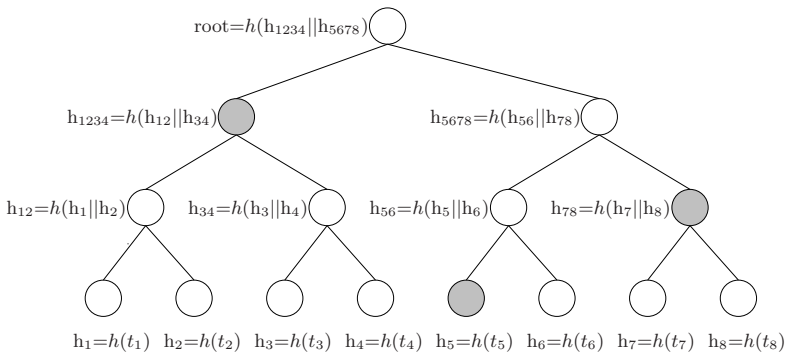


Fig. 15. A Merkle hash tree over attribute `Name` of relation MEDICALDATA in Figure 2

the concatenation of $h(t_{i-1})$ with $h(t_i)$, with t_{i-1} the tuple preceding t_i in the order. Given a range query q operating on a, incompleteness of the result can be immediately detected by checking the signature associated with the tuples in the query result. For instance, suppose that tuple t_i has been omitted in the computation of the query result. While checking the signature of the tuples, the client would discover that the computed signature of t_{i+1} (i.e., $h(t_{i-1})||h(t_{i+1})$) is different from the one stored with t_{i+1}, which is $h(t_i)||h(t_{i+1})$.

Since signature chaining guarantees completeness of query results only with respect to the attribute on which the signature chain has been defined, a signature chain should be defined for each attribute that may be involved in a range query. The main limitation of this approach is related to the size of the signature associated with each tuple, which increases linearly with the number of signature chains.

Merkle hash trees. Integrity of query computations can be provided also by using a Merkle hash tree built over the outsourced relation [54]. 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. The root of the Merkle hash tree is signed by the data owner and communicated to authorized users. Figure 15 illustrates an example of a Merkle hash tree defined over attribute `Name` of relation MEDICALDATA in Figure 2(a). Given a range query to be evaluated over attribute a, the result returned to the requesting client includes also a verification object 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 query result, the client computes the value of the root using the VO and the tuples in the query result. It then checks if the computed value is the same as the root initially received from the data owner [32]. The computation of VO depends on the type of query to be evaluated. For instance, in case of a selection

query that returns 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 and the Merkle hash tree in Figure 15, consider a query returning the patient with name *Fred*. The query returns tuple t_6 and its VO contains the gray nodes in the figure.

The original technique illustrated in [32] has been extended to improve the efficiency of the verification processes (e.g., [51,58]), and to support integrity verification of join results [84].

Skip lists. Another authenticated structure that can be used to verify the integrity of queries searching for a key value in a set of elements is represented by *skip lists* [33]. A skip list for a set $\mathcal{S}$ 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 $\mathcal{S}$ 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 16(a) illustrates a skip list with three levels for $\mathcal{S}$={5,6,8,9,10}.

The search operation for a key value v in a skip list starts from sentinel $-\infty$ in the top list (i.e., S_k) and 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 16(a), Figure 16(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 queries searching for a value v in $\mathcal{S}$. To this aim, the skip list defined for $\mathcal{S}$ 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 16(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 16(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 sent to all the authorized users.

If a query searching for element v returns a positive answer (i.e., $v \in \mathcal{S}$), 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 client receives a verification object that includes

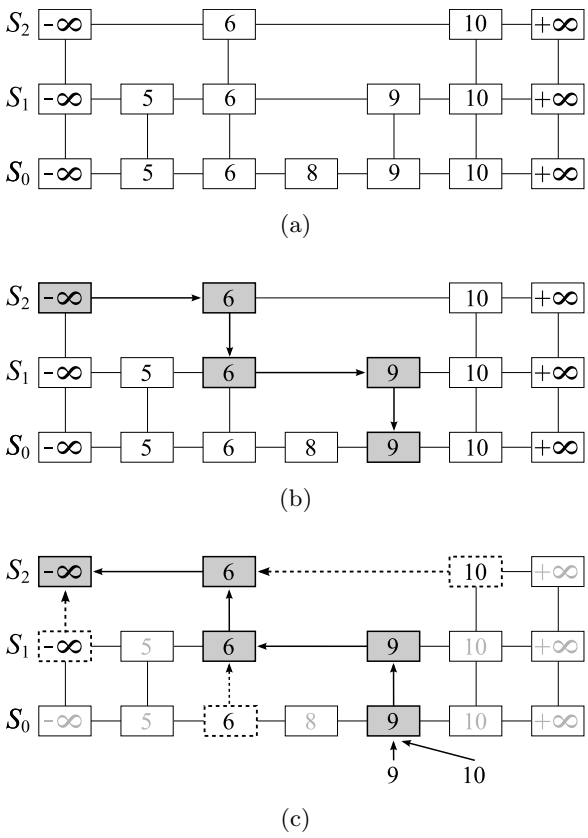


Fig. 16. A skip list for set $\mathcal{S}$={5,6,8,9,10} with three levels (a), search process for key value 9 (b), and verification object for a query searching for value 9 (c)

the label of the nodes on the right and below the nodes forming the path to v, which are necessary and sufficient to the client for computing the label of the received nodes. For instance, consider the skip list in Figure 16(a) and suppose to search key value 9. Figure 16(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 client 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.

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

6.2 Probabilistic approaches

All the techniques described in Section 6.1 can assess the integrity of query results only for 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 tuples [53,81]. 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 [81], even a limited number of fake tuples ensures high probabilistic guarantee of completeness.

Controlled replication [76]. An alternative probabilistic approach to verify the completeness of selection queries consists in replicating all 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., $E_{k_2}(t)$). 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.

Combining fake tuples and controlled replication [19,22,23]. These proposals permit to assess the integrity of join queries in a scenario where two trusted storage providers $\mathcal{S}_l$ and $\mathcal{S}_r$ store the base relations L and R to be joined, and a non fully trusted computational provider $\mathcal{C}_p$ is in charge of evaluation the join. To verify the correctness and completeness of the join result, the client collaborates with the storage providers, asking them to insert *markers* and *twins* in their relations before being sent to the computational provider. Markers are fake tuples, not recognizable as such by the computational provider, dynamically inserted into the operand relations by the storage providers. To ensure the presence of markers in the result of the join operation, the same set of markers is

inserted into both L and R. The client then coordinates the number of markers and their values of the join attribute. Twins are copies of the original tuples that satisfy a twinning condition C_{twin} defined by the client and communicated to both the storage providers. The twinning condition regulates the percentage of twins to be inserted in the operand relations (and hence also in the join result). To be applicable to both L and R, the twinning condition operates on the join attribute (which is the only attribute common to the two operand relations). Note that the values of the join attribute for markers and twins are chosen outside the domain of the original join attribute values, to prevent the insertion of spurious tuples in the result computed by the computational provider. To protect the confidentiality of the data and to prevent the computational provider from identifying markers and twins, the operand relations are *encrypted* (with a key chosen by the client) before sending them to the computational provider. Also, the frequency distribution of values of the join attribute of the tuples participating in a one-to-many join is flattened by adopting *salts* and/or *buckets*. Salts consist in combining different occurrences of the same join attribute value in the relationon the side "many" with a different salt, to guarantee that they map to different encrypted values. At the same time salted replicas are created at side "one" of the join so to create the corresponding matching. Bucketization consists instead in making the number of occurrences of each value of the join attribute at the side many of the join equal by also inserting dummy tuples when necessary. The client checks whether the tuples in the join result satisfy the join condition, the presence of the expected markers, and verifies whether the tuples satisfying the twinning condition are duplicated in the result.

Twins and markers offer complementary controls [22]: twins are twice as effective as markers, but loose their effectiveness when the computational provider omits a large fraction of tuples; markers allow detecting extreme behavior (all tuples omitted) and provide effective when the computational provider omits a large fraction of tuples. Figure 17 illustrates the computation of a one-to-one join between L(`Id`,`Name`) and R(`Id`,`Premium`) with the adoption of twins, markers, and encryption on-the-fly [22]. The two relations are first extended with twins (light gray) and markers (dark gray). The resulting extended relations (L^* and R^*) are then encrypted and sent to the computational provider (in the figure, encrypted values are denoted by uppercase Greek letters). The encrypted relations L_k^* and R_k^* have two attributes: I_k, the encrypted join attribute; $L^*.Tuple_k$ and $R^*.Tuple_k$, the encryption of all attributes (including the join attribute). The computational provider computes the natural join between the received encrypted relations and sends the result (J_k^*) to the client. The client projects over attributes $L^*.Tuple_k$ and $R^*.Tuple_k$, decrypts the result of projection (obtaining relation J^*), verifies its completeness and correctness, and if no omission is detected, removes twins and markers to obtain the join result (J).

The solution in [22] has been extended in [19] to support arbitrary kinds of joins (i.e., one-to-one, one-to-many, and many-to-many) and join sequences in a distributed and parallel platform (MapReduce). Also, the work in [23] presents some optimizations for limiting the overhead to be paid for integrity guarantees.

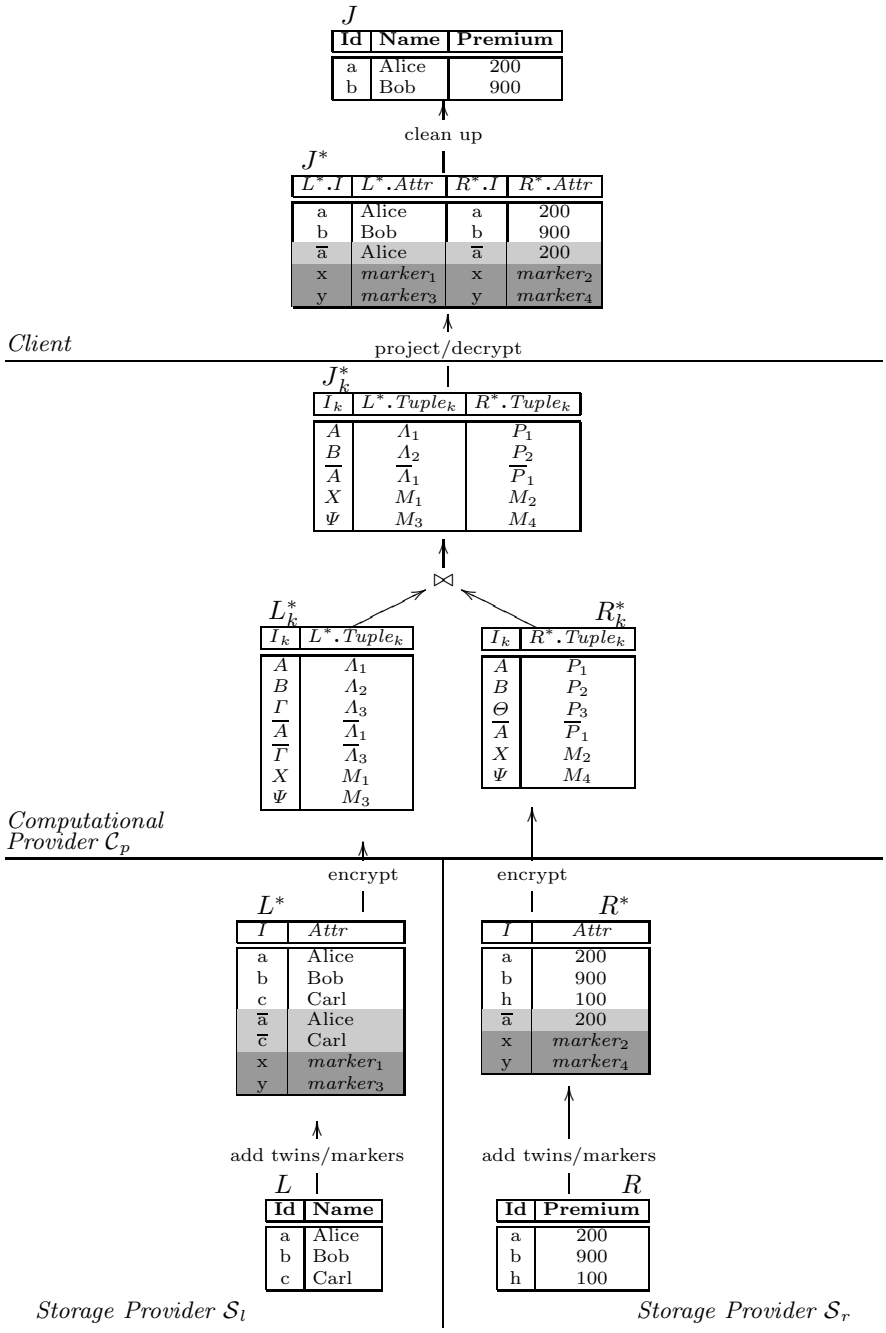


Fig. 17. An example of query evaluation process with twins (light gray) on 'a' and 'c' and two markers (dark grey)

7 Conclusions

In this chapter, we have illustrated some encryption-based approaches for protecting and managing data in the cloud. In particular, we have discussed the application of encryption for protecting confidentiality, integrity, and availability of externally stored data, and for enforcing access control restrictions over them. We have also illustrated techniques for evaluating queries over encrypted data. Finally, we have discussed approaches for protecting the confidentiality of accesses and for guaranteeing the integrity, in terms of correctness and completeness, of query results.

Acknowledgements. This work was supported in part by: the EC within the 7FP under grant agreement 312797 (ABC4EU) and within the H2020 program under grant agreement 644579 (ESCUDO-CLOUD), and the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH).

References

1. Agrawal, R., Kierman, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proc. of SIGMOD 2004. Paris, France (June 2004)
2. Akl, S., Taylor, P.: Cryptographic solution to a problem of access control in a hierarchy. ACM Transactions on Computer Systems 1(3), 239–248 (August 1983)
3. Atallah, M., Blanton, M., Fazio, N., Frikken, K.: Dynamic and efficient key management for access hierarchies. ACM TISSEC 12(3), 18:1–18:43 (January 2009)
4. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Proc. of CCS 2007 (October–November 2007)
5. Barni, M., Bianchi, T., Catalano, D., Raimondo, M.D., Labati, R.D., Failla, P., Fiore, D., Lazzeretti, R., Piuri, V., Scotti, F., Piva, A.: A privacy-compliant fingerprint recognition system based on homomorphic encryption and fingercode templates. In: Proc. of BTAS 2010. Washington, D.C., USA (September 2010)
6. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Proc. of EUROCRYPT 2003. Warsaw, Poland (May 2003)
7. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. SIAM Journal on Computing 43(2), 831–871 (April 2014)
8. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Proc. of EUROCRYPT 1999. Prague, Czech Republic (May 1999)
9. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Fragmentation and encryption to enforce privacy in data storage. In: Proc. of ESORICS 2007. Dresden, Germany (September 2007)
10. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Keep a few: Outsourcing data while maintaining confidentiality. In: Proc. of ESORICS 2009. Saint Malo, France (September 2009)
11. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Combining fragmentation and encryption to protect privacy in data storage. ACM TISSEC 13(3), 22:1–22:33 (July 2010)

12. Coron, J.S., Mandal, A., Naccache, D., Tibouchi, M.: Fully homomorphic encryption over the integers with shorter public keys. In: Proc. of CRYPTO 2011. Santa Barbara, CA, USA (August 2011)
13. Coron, J.S., Naccache, D., Tibouchi, M.: Public key compression and modulus switching for fully homomorphic encryption over the integers. In: Proc. of EUROCRYPT 2012. Cambridge, UK (April 2012)
14. Damiani, E., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Selective data encryption in outsourced dynamic environments. In: Proc. of VODCA 2006. Bertinoro, Italy (September 2006)
15. Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. In: Proc. of ACM CCS 2003. Washington, DC, USA (October 2003)
16. Damiani, E., De Capitani di Vimercati, S., Samarati, P.: New paradigms for access control in open environments. In: Proc. of ISSPI 2005. Athens, Greece (December 2005)
17. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G., Paraboschi, S., Samarati, P.: Enforcing dynamic write privileges in data outsourcing. Computers & Security (November 2013)
18. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G., Paraboschi, S., Samarati, P.: Fragmentation in presence of data dependencies. IEEE TDSC 11(6), 510–523 (November–December 2014)
19. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G., Paraboschi, S., Samarati, P.: Integrity for distributed queries. In: Proc. of CNS 2014. San Francisco, CA, USA (October 2014)
20. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Pelosi, G., Samarati, P.: Preserving confidentiality of security policies in data outsourcing. In: Proc. of WPES 2008. Alexandria, Virginia, USA (October 2008)
21. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM TODS 35(2), 12:1–12:46 (April 2010)
22. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Integrity for join queries in the cloud. IEEE TCC 1(2), 187–200 (July–December 2013)
23. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: 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)
24. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Samarati, P.: Access control policies and languages in open environments. In: Yu, T., Jajodia, S. (eds.) Secure Data Management in Decentralized Systems. Springer-Verlag (2007)
25. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. of ICDCS 2011. Minneapolis, MN, USA (June 2011)
26. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Distributed shuffling for preserving access confidentiality. In: Proc. of ESORICS 2013. Egham, UK (September 2013)
27. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Supporting concurrency and multiple indexes in private access to outsourced data. JCS 21(3), 425–461 (2013)

28. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Protecting access confidentiality with data distribution and swapping. In: Proc. of BDCloud 2014. Sydney, Australia (December 2014)
29. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Managing and accessing data in the cloud: Privacy risks and approaches. In: Proc. of CRiSIS 2012. Cork, Ireland (October 2012)
30. De Capitani di Vimercati, S., Samarati, P., Jajodia, S.: Policies, models, and languages for access control. In: Proc. of DNIS 2005. Aizu-Wakamatsu, Japan (March 2005)
31. Delerue Arriaga, A., Tang, Q., Ryan, P.: Trapdoor privacy in asymmetric searchable encryption schemes. In: Proc. of AFRICACRYPT 2014. Marrakesh, Morocco (May 2014)
32. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.: Authentic third-party data publication. In: Proc. of DBSec 2000. Schoorl, The Netherlands (August 2000)
33. Di Battista, G., Palazzi, B.: Authenticated relational tables and authenticated skip lists. In: Proc. of DBSec 2007. Redondo Beach, CA, USA (July 2007)
34. Ding, X., Yang, Y., Deng, R.: Database access pattern protection without full-shuffles. IEEE Transactions on Information Forensics and Security 6(1), 189–201 (March 2011)
35. Fangming, Z., Takashi, N., Kouichi, S.: Realizing fine-grained and flexible access control to outsourced data with attribute-based cryptosystems. In: Proc. of ISPEC 2011. Guangzhou, China (May-June 2011)
36. Foresti, S.: Preserving Privacy in Data Outsourcing. Springer (2011)
37. Gamassi, M., Piuri, V., Sana, D., Scotti, F.: Robust fingerprint detection for access control. In: Proc. of RoboCare Workshop 2005. Rome, Italy (May 2005)
38. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proc. of STOC 2009. Bethesda, MA, USA (May 2009)
39. Goh, E.J.: Secure indexes. Tech. Rep. 2003/216, Cryptology ePrint Archive (2003), http://eprint.iacr.org/
40. Goldreich, O., Ostrovsky, R.: Software protection and simulation on Oblivious RAMs. Journal of the ACM 43(3), 431–473 (May 1996)
41. Goodrich, M., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Practical oblivious storage. In: Proc. of CODASPY 2012. San Antonio, TX, USA (February 2012)
42. Goodrich, M., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless Oblivious RAM simulation. In: Proc. of SODA 2012. Kyoto, Japan (January 2012)
43. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: Proc. of ACM CCS 2006. Alexandria, VA, USA (October–November 2006)
44. Hacigümüs, H., Iyer, B., Mehrotra, S.: Ensuring integrity of encrypted databases in database as a service model. In: Proc. of DBSec 2003. Estes Park, CO, USA (August 2003)
45. Hacigümüs, H., Iyer, B., Mehrotra, S.: Efficient execution of aggregation queries over encrypted relational databases. In: Proc. of DASFAA 2004. Jeju Island, Korea (March 2004)
46. Hacigümüs, H., Iyer, B., Mehrotra, S., Li, C.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. of SIGMOD 2002. Madison, WI, USA (June 2002)
47. Jhawar, R., Piuri, V., Samarati, P.: Supporting security requirements for resource management in cloud computing. In: Proc. of CSE 2012. Paphos, Cyprus (December 2012)

48. Jhawar, R., Piuri, V., Santambrogio, M.: A comprehensive conceptual system-level approach to fault tolerance in cloud computing. In: Proc. of SysCon 2012. Vancouver, BC, Canada (March 2012)
49. Jhawar, R., Piuri, V., Santambrogio, M.: Fault tolerance management in cloud computing: A system-level perspective. IEEE Systems Journal 7(2), 288–297 (June 2013)
50. Juels, A., Kaliski Jr, B.S.: PORs: Proofs of retrievability for large files. In: Proc. of ACM CCS 2007. Alexandria, VA, USA (October–Novermber 2007)
51. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: Proc. of SIGMOD 2006. Chicago, IL, USA (June 2006)
52. Lin, P., Candan, K.: Hiding traversal of tree structured data from untrusted data stores. In: Proc. of WOSIS 2004. Porto, Portugal (April 2004)
53. Liu, R., Wang, H.: Integrity verification of outsourced XML databases. In: Proc. of CSE 2009. Vancouver, Canada (August 2009)
54. Merkle, R.: A certified digital signature. In: Proc. of CRYPTO 1989. Santa Barbara, CA, USA (August 1989)
55. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. ACM TOS 2(2), 107–138 (May 2006)
56. Ostrovsky, R., Skeith, III, W.E.: A survey of single-database private information retrieval: Techniques and applications. In: Proc. of PKC 2007. Beijing, China (April 2007)
57. Pang, H., Jain, A., Ramamritham, K., Tan, K.: Verifying completeness of relational query results in data publishing. In: Proc. of SIGMOD 2005. Baltimore, MA, USA (June 2005)
58. Pang, H., Tan, K.: Authenticating query results in edge computing. In: Proc. of ICDE 2004. Boston, MA, USA (April 2004)
59. Pang, H., Zhang, J., Mouratidis, K.: Enhancing access privacy of range retrievals over $B+$-trees. IEEE TKDE 25(7), 1533–1547 (July 2013)
60. Popa, R., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: Protecting confidentiality with encrypted query processing. In: Proc. of SOSP 2011. Cascais, Portugal (October 2011)
61. Rivest, R., Adleman, L., Dertouzos, M.: On data banks and privacy homomorphisms. In: DeMillo, R., Lipton, R., Jones, A. (eds.) Foundation of Secure Computations. Academic Press (1978)
62. Ruj, S., Stojmenovic, M., Nayak, A.: Privacy preserving access control with authentication for securing data in clouds. In: Proc. of CCGrid 2012. Ottawa, Canada (May 2012)
63. Samarati, P.: Data security and privacy in the cloud. In: Proc. of ISPEC 2014. Fuzhou, China (May 2014)
64. Samarati, P., De Capitani di Vimercati, S.: Data protection in outsourcing scenarios: Issues and directions. In: Proc. of ASIACCS 2010. Beijing, China (April 2010)
65. Samarati, P., De Capitani di Vimercati, S.: Cloud security: Issues and concerns. In: Murugesan, S., Bojanova, I. (eds.) Encyclopedia on Cloud Computing. Wiley (2015)
66. Sandhu, R.: On some cryptographic solutions for access control in a tree hierarchy. In: Proc. of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow. Dallas, TX, USA (October 1987)
67. Sandhu, R.: Cryptographic implementation of a tree hierarchy for access control. Information Processing Letters 27(2), 95–98 (February 1988)

68. Shacham, H., Waters, B.: Compact proofs of retrievability. In: Proc. of ASIACRYPT 2008. Melbourne, Australia (December 2008)
69. Song, D., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proc. of IEEE S&P 2000. Berkeley, CA, USA (May 2000)
70. Stefanov, E., van M. Dijk, Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: An extremely simple Oblivious RAM protocol. In: Proc of ACM CCS 2013. Berlin, Germany (November 2013)
71. Stefanov, E., Shi, E.: ObliviStore: High performance oblivious cloud storage. In: Proc. of IEEE S&P 2013. Berkeley, CA, USA (May 2013)
72. Wan, Z., Liu, J., Deng, R.H.: HASBE: A hierarchical attribute-based solution for flexible and scalable access control in cloud computing. IEEE Transactions on Information Forensics and Security 7(2), 743–754 (April 2012)
73. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. IEEE Transactions on Parallel and Distributed Systems 23(8), 1467–1479 (August 2012)
74. Wang, C., Chow, S.S., Wang, Q., Ren, K., Lou, W.: Privacy-preserving public auditing for secure cloud storage. IEEE Transactions on Computers 62(2), 362–375 (February 2013)
75. Wang, H., Lakshmanan, L.: Efficient secure query evaluation over encrypted XML databases. In: Proc. of VLDB 2006. Seoul, Korea (September 2006)
76. Wang, H., Yin, J., Perng, C., Yu, P.: Dual encryption for query integrity assurance. In: Proc. of CIKM 2008. Napa Valley, CA, USA (October 2008)
77. Waters, B.: Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In: Proc. of PKC 2011. Taormina, Italy (March 2011)
78. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: Proc. of ACM CCS 2012. Raleigh, NC, USA (October 2012)
79. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: Proc. of ACM CCS 2008. Alexandria, VA, USA (October 2008)
80. Williams, P., Sion, R., Tomescu, A.: PrivateFS: A parallel oblivious file system. In: Proc. of ACM CCS 2012. Raleigh, NC, USA (October 2012)
81. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity auditing of outsourced data. In: Proc. of VLDB 2007. Vienna, Austria (September 2007)
82. Yang, K., Jia, X., Ren, K.: Attribute-based fine-grained access control with efficient revocation in cloud storage systems. In: Proc. of ASIACCS 2013. Hangzhou, China (May 2013)
83. Yang, K., Zhang, J., Zhang, W., Qiao, D.: A light-weight solution to preservation of access pattern privacy in un-trusted clouds. In: Proc. of ESORICS 2011. Leuven, Belgium (September 2011)
84. Yang, Y., Papadias, D., Papadopoulos, S., Kalnis, P.: Authenticated join processing in outsourced databases. In: Proc. of SIGMOD 2009. Providence, RI, USA (June-July 2009)
85. Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: Proc. of INFOCOM 2010. San Diego, CA, USA (March 2010)
86. Zhu, Y., Ahn, G.J., Hu, H., Yau, S., An, H., Hu, C.J.: Dynamic audit services for outsourced storages in clouds. IEEE Transactions on Services Computing 6(2), 227–238 (April 2013)