Selective and Private Access to Outsourced Data Centers

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

1 Introduction

The increasing amount of information being generated, collected, shared, and disseminated nowadays is making the in-house management of data centers by private and public companies more and more difficult and economically expensive. The wide availability of cloud providers offering high-quality services for data storage and management is then a driving motivation for companies that more often move their data centers to the cloud. Although this trend has clear economic advantages, it also introduces novel security issues. In fact, when moving a data center to the cloud, the data are no more under the direct control of their owner who needs to rely on an external system for providing the same guarantees as in their in-house management (e.g., data availability, protection against external attacks, selective access to the data, fault tolerance management [32–34, 39]). However, being external third parties, cloud providers are often assumed to be *honest-but-curious*, and hence trusted to correctly manage the data they store but not trusted to access their content. This situation raises several concerns, especially with respect to the proper protection of the confidentiality of the data. An effective solution consists in encrypting the data before outsourcing them so that non-authorized parties (including the cloud provider), not knowing the encryption key, cannot access the data content in plaintext [9, 31]. Data encryption before outsourcing presents however some disadvantages.

S. De Capitani di Vimercati (✉) · S. Foresti · G. Livraga · P. Samarati
Dipartimento di Informatica, Università degli Studi di Milano,
Via Bramante 65, 26013 Crema, Italy
e-mail: sabrina.decapitani@unimi.it

S. Foresti
e-mail: sara.foresti@unimi.it

G. Livraga
e-mail: giovanni.livraga@unimi.it

P. Samarati
e-mail: pierangela.samarati@unimi.it

S. U. Khan, A. Y. Zomaya (eds.), *Handbook on Data Centers,*
DOI 10.1007/978-1-4939-2092-1_33

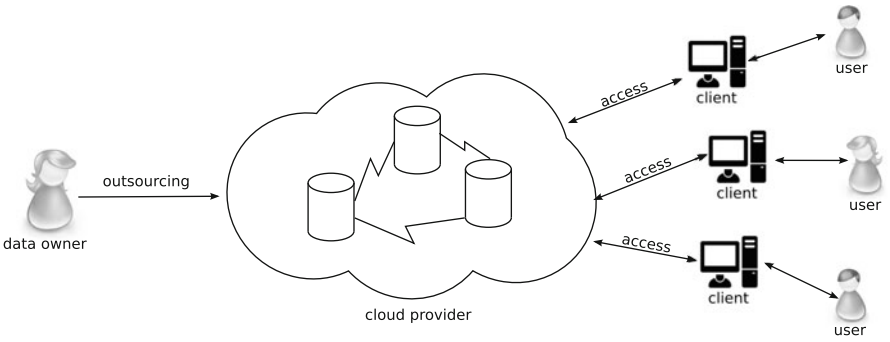


Fig. 1 Reference scenario

First, while effectively hiding plaintext data to the eyes of the provider, encrypting all data with a single key would require either all users to have complete visibility of the resources in the data collection, or the data owner to mediate access requests to the data to enforce selective access. Second, encryption complicates query evaluation since the cloud provider cannot directly evaluate users queries over encrypted data. Third, in cases where also the queries posed by users need to be protected, encryption might not provide sufficient protection guarantees.

To overcome such issues, different techniques have been proposed that aim at supporting selective and private access to outsourced data. These techniques are based on the use of *selective encryption*, meaning that different pieces of data are encrypted with different keys according to who can access them. *Indexes* are instead used by cloud providers to select the data to be returned in response to a query, possibly even without revealing the target of the query itself. While, singularly taken, these techniques represent effective solutions, the combined adoption of selective encryption and indexes may cause violations of confidentiality that still need to be carefully addressed. In this chapter, we present an overview of the techniques proposed for enabling data to self-enforce the access control policy defined by their owner, and for supporting query evaluation on encrypted data. Fig. 1 illustrates the reference scenario where a data owner outsources her data to a cloud provider and users access such data.

The remainder of this chapter is organized as follows. Sect. 2 shows how encrypted data can enforce access control restrictions, without requiring the intervention of the data owner or the collaboration of the storing server. Sect. 3 presents an overview of the techniques proposed for supporting query evaluation over encrypted data. Sect. 4 describes novel solutions for accessing outsourced data collections without revealing the target of the query to the storing server. Sect. 5 illustrates the privacy issues arising when combining solutions for access control enforcement with indexing techniques and introduces preliminary solutions to this problem. Finally, Sect. 6 presents our closing remarks.

2 Access Control Enforcement

The information stored in data centers can be of any type: relational databases, XML documents, multimedia files, and so on. For simplicity, but without loss of generality, in this chapter we assume the data stored in the cloud to be organized in a relational database, with the note that all approaches illustrated in the following can be easily adapted to operate on any logical data modeling. We then consider a relation r defined over schema $R(a_1, \ldots, a_n)$, where attribute a_i is defined over domain D_i, $i = 1, \ldots, n$. At the storing server, relation r is represented through an encrypted relation r^k, defined over schema $R^k(\underline{\texttt{tid}}, \texttt{enc})$, with `tid` a numerical 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 randomly chosen by the data owner and $t^k[\texttt{enc}]=E_k(t)$, with E a symmetric encryption function with key k.

Different techniques have been proposed to enforce access control with the intervention of neither the storing server, for confidentiality reasons, nor the data owner, for efficiency reasons (e.g., [10, 12, 29]). These solutions are based on the idea that data *self-enforce* selective access restrictions through encryption, as illustrated in the following of this section.

2.1 Selective Encryption

A promising solution for enforcing access control to outsourced data is based on selective encryption, which adopts different encryption keys for different tuples, and selectively distributes keys to authorized users. Each user can decrypt and therefore access a subset of tuples, depending on the keys she knows. The authorization policy regulating which user can read which tuple is defined by the data owner before outsourcing relation r (e.g., [10, 12]). The authorization policy can be represented as a binary *access matrix* M with a row for each user u, and a column for each tuple t, where: $\mathrm{M}[u,t] = 1$ iff u can access t; $\mathrm{M}[u,t] = 0$ otherwise. To illustrate, consider relation PATIENTS in Fig. 2. Figure 3 illustrates an example of access matrix regulating access to the tuples in relation PATIENTS by users A, B, C, D, and E. The j^{th} column of the matrix represents the access control list $acl(t_j)$ of tuple t_j, for each $j = 1, \ldots, |r|$. As an example, with reference to the matrix in Fig. 3, $acl(t_1) = ABC$. The encryption policy, which defines and regulates the set of keys used to encrypt tuples and the distribution of keys to the users, must be *equivalent* to the authorization policy, meaning that each user should be able to decrypt all and only the tuples she is authorized to access.

Solutions translating an authorization policy into an equivalent encryption policy (e.g., [12]) have two main design desiderata: *(i)* guarantee that each user has to manage only one key; and *(ii)* encrypt each tuple with only one key (i.e., no tuple is replicated). To fulfill these two requirements, selective encryption approaches rely on *key derivation techniques* that permit to compute an encryption key k_j starting from the knowledge of another key k_i and (possibly) a piece of publicly available

PATIENTS

	SSN	Name	ZIP	MarStatus	Illness
t_1	123456789	Ann	22010	single	gastritis
t_2	234567891	Barbara	24027	divorced	neuralgia
t_3	345678912	Carl	22010	married	gastritis
t_4	456789123	Daniel	20100	married	gastritis
t_5	567891234	Emma	21048	single	neuralgia
t_6	678912345	Fred	23013	married	hypertension
t_7	789123456	Gary	22010	widow	gastritis
t_8	891234567	Harry	24027	widow	hypertension

Fig. 2 An example of relation

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

Fig. 3 An example of access matrix

information. To determine which key can be derived from which other key, key derivation techniques require the preliminary definition of a *key derivation hierarchy*. A key derivation hierarchy 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. Note that key derivation can be applied in chain, meaning that key k_j can be computed starting from key k_i if there is a path (of arbitrary length) from v_i to v_j in the key derivation hierarchy.

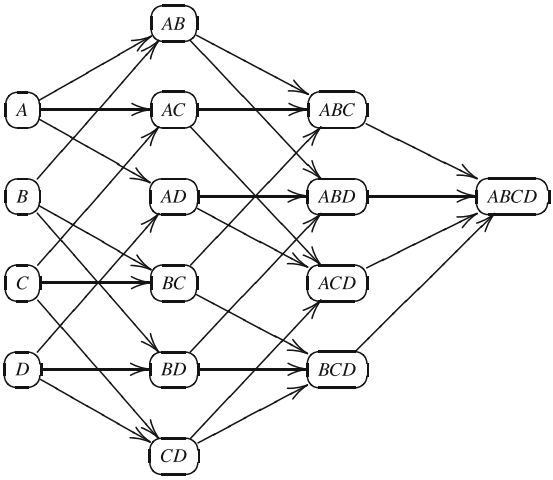
A key derivation hierarchy can have different shapes, as described in the following.

- *Chain of vertices* (e.g., [40]): the key k_j associated with vertex v_j is computed by applying a one-way function to the key k_i of its predecessor in the chain. No public information is needed.
- *Tree hierarchy* (e.g., [41]): the key k_j associated with vertex v_j is computed by applying a one-way function to the key k_i of its direct ancestor, 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, 3, 7, 19]): keys in the hierarchy can have more than one direct ancestor, and each edge in the hierarchy is associated with a publicly available *token* [3]. Given two keys k_i and k_j, 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 knowing (or able to derive) key k_i can also derive key k_j.

Each of the proposed key derivation hierarchies has advantages and disadvantages. However, the token-based key derivation best fits the outsourcing scenario by minimizing the need of re-encryption and/or key re-distribution in case of updates to the authorization policy [12] (for more details, see Sect. 2.2).

The set containment relationship $\subseteq$ over the set U of users can nicely be used to define a DAG key derivation hierarchy suited for access control enforcement and able to satisfy the desiderata of limiting the key management overhead [12]. Such a hierarchy has a vertex for each of the elements of the power-set of the set U of users, and 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. The correct enforcement of the authorization policy defined by the data owner is guaranteed iff: *(i)* each user u_i is communicated the key associated with the vertex representing her; and *(ii)* each tuple t_j is encrypted with the key of the vertex representing $acl(t_j)$. With this strategy, each tuple can be decrypted and accessed by all and only the users in its access control list, meaning that the encryption policy is equivalent to the authorization policy defined by the data owner. Furthermore, each user has to manage one key only, and each tuple is encrypted with one key only. For instance, Fig. 4a illustrates the key derivation hierarchy induced by the set U$= \{A, B, C, D\}$ of users and the subset containment relationship over it (in the figure, vertices are labeled with the set of users they represent). Fig. 4b and Fig. 4c illustrate the keys assigned to users in the system and the keys used to encrypt the tuples in relation PATIENTS in Fig. 2, respectively. The encryption policy in the figure enforces the access control policy in Fig. 3 restricted to the set U$= \{A, B, C, D\}$ of users as each user can derive, from her own key, the keys of the vertices to which she belongs and hence decrypt the tuples she is authorized to read. For instance, user C can derive the keys used to encrypt tuples t_1, t_2, t_3, t_5, and t_6, and then access their content.

Even though this approach correctly enforces an authorization policy and enjoys ease of implementation, it defines more keys and more tokens than necessary. Since tokens are stored in a publicly available catalog at the server side, when a user u wants to access a tuple t she needs to interact with the server to visit the path in the key derivation hierarchy from the vertex representing u to the vertex representing $acl(t)$. Therefore, keeping the number of tokens low increases the efficiency of the derivation process, and then also of the response time to users. The problem of minimizing the number of tokens, while guaranteeing equivalence between the authorization and the encryption policies, is NP-hard (it can be reduced to the set cover problem) [12]. It is however interesting to note that: *(i)* the vertices needed for correctly enforcing an authorization policy are only those representing singleton sets of users (corresponding to users' keys) and the access control lists of the tuples (corresponding to keys used to encrypt tuples) in r; *(ii)* when two or more vertices have more than two common direct ancestors, the insertion of a vertex representing the set of users corresponding to these ancestors reduces the total number of tokens. Elaborating on these two intuitions to reduce the number of tokens, the following heuristic approach efficiently provides good results [12].



user	key
A	k_A
B	k_B
C	k_C
D	k_D

b

tuple	key
t_1	k_{ABC}
t_2	k_{ABC}
t_3	k_{BC}
t_4	k_{ABD}
t_5	k_{ABCD}
t_6	k_{ACD}
t_7	k_A
t_8	k_D

c

Fig. 4 An example of encryption policy equivalent to the access control policy in Fig. 3, considering the subset $\{A, B, C, D\}$ of users

1. *Initialization*. The algorithm first identifies the vertices necessary to implement the authorization policy, that is, the vertices representing: *(i)* singleton sets of users, whose keys are communicated to users and that allow them to derive the keys of the tuples they are entitled to access; and *(ii)* the access control lists of the tuples, whose keys are used for encryption. These vertices represent the set of *material* vertices of the system.
2. *Covering*. For each material vertex v corresponding to a non-singleton set of users, the algorithm finds a set of material vertices that form a *non-redundant set covering* for v, which become direct ancestors of v. A set V of vertices is a set covering for v if for each u in v, there is at least a vertex v_i in V such that u appears in v_i. It is non-redundant if the removal of any vertex from V produces a set that does not cover v.
3. *Factorization*. 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 $v'_1, \ldots, v'_n$ and 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.

Figure 5 illustrates, step by step, the definition of the key derivation hierarchy through the algorithm in [12], for the authorization policy in Fig. 3. The initialization phase generates the set of (material) vertices in Fig. 5a. The covering phase generates the preliminary key derivation hierarchy in Fig. 5b, where each vertex is connected to a set of parents including all and only the users in the vertex itself. The factorization phase generates the key derivation hierarchy in Fig. 5c, which has an additional non-material vertex (i.e., *ADE*, denoted with a dotted line in the figure) representing the users that belong to both *ABDE* and *ACDE*. This factorization saves one token.

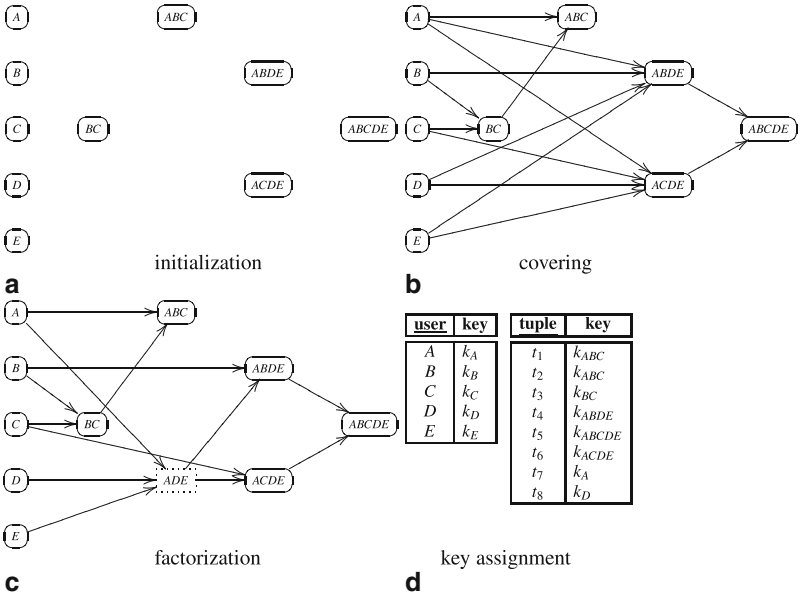


Fig. 5 Definition of an encryption policy equivalent to the access control policy in Fig. 3

Figure 5d illustrates the keys assigned to users in the system and the keys used to encrypt the tuples in relation PATIENTS in Fig. 2.

2.2 *Updates to the Access Control Policy*

In case of changes to the authorization policy, the encryption policy must be updated accordingly, to guarantee their equivalence. Since the key used to encrypt each tuple t in r depends on the set of users who can access it, it might be necessary to re-encrypt the tuples involved in the policy update with a different key that only the users in their new access control lists know or can derive. A trivial approach to enforce a grant/revoke operation on tuple t requires the data owner to: *(i)* download t^k from the server; *(ii)* decrypt it; *(iii)* update the key derivation hierarchy if it does not include a vertex representing the new set of users in $acl(t)$; *(iv)* encrypt t with the key k' associated with the vertex representing $acl(t)$; *v)* upload the new encrypted version of t on the server; and *(vi)* possibly update the public catalog containing the tokens. For instance, consider the encryption policy in Figs. 5c–d and assume that user D is granted access to tuple t_1. The data owner should download t_1^k; decrypt it using key k_{ABC}; insert a vertex representing $acl(t_1) = ABCD$ in the key derivation hierarchy; encrypt t_1 with k_{ABCD}; and upload the encrypted tuple on the server, together with the tokens necessary to users A, B, C, and D to derive k_{ABCD}. This approach, while effective and correctly enforcing authorization updates, leaves to the data owner the

burden of managing the update. Also, re-encryption operations are computationally expensive. To limit the data owner overhead, in [12] the authors propose to use two layers of encryption (each characterized by its own encryption policy) to partially delegate to the server the management of grant and revoke operations.

- The *Base Encryption Layer* (BEL) is applied by the data owner before outsourcing the dataset. A BEL key derivation hierarchy is built according to the authorization policy existing at initialization time. In case of policy updates, BEL is only updated by possibly inserting tokens in the public catalog (i.e., edges in the BEL key derivation hierarchy). Note that each vertex v in the BEL key derivation hierarchy has two keys: a derivation key k (used for key derivation only), and an access key k^a (used to encrypt tuples, but that cannot be exploited for key derivation purposes).
- The *Surface Encryption Layer* (SEL) is applied by the server over the tuples that have already been encrypted by the data owner at BEL. It dynamically enforces the authorization policy updates by possibly re-encrypting tuples and changing the SEL key derivation hierarchy to correctly reflect the updates. Differently from BEL, vertices in the SEL key derivation hierarchy are associated with a single key k^s.

Intuitively, with the over-encryption approach, a user can access a tuple t only if she knows the keys used to encrypt t at BEL and SEL. At initialization time, the encryption policies at BEL and SEL coincide, but they immediately change and become different at each policy update. Grant and revoke operations are enforced as follows.

- *Grant*. When user u is granted access to tuple t, she needs to know the key used to encrypt t at both BEL and SEL. Hence, the data owner adds a token in the BEL key derivation hierarchy from the vertex representing u to the vertex whose key is used to encrypt t (i.e., the vertex representing $acl(t)$ at initialization time). The owner then asks the server to update the key derivation hierarchy at SEL and to possibly re-encrypt tuples. Tuple t in fact needs to be encrypted at SEL with the key of the vertex representing $acl(t) \cup \{u\}$ (which is possibly inserted into the hierarchy). Besides t, also other tuples may need to be re-encrypted at SEL to guarantee the correct enforcement of the policy update. In fact, tuples that are encrypted with the same key as t at BEL and that user u is not allowed to read must be encrypted at SEL with a key that u does not know (and cannot derive). The data owner must then make sure that each tuple t_i sharing the BEL encryption key with t are encrypted at SEL with the key of the vertex representing $acl(t_i)$. For instance, consider the access matrix in Fig. 3 and the encryption policies at BEL and SEL enforcing it in Fig. 6, and assume that user D is granted access to tuple t_1. Figure 7 illustrates the encryption policies at BEL and SEL after the enforcement of the grant operation. To enforce this change in the access control policy, the data owner must first add a token that permits user D to derive the access key of vertex ABC (k^a_{ABC}) used to encrypt t_1 at BEL (dotted edge in the figure). Also, she will ask the server to update the SEL key derivation hierarchy

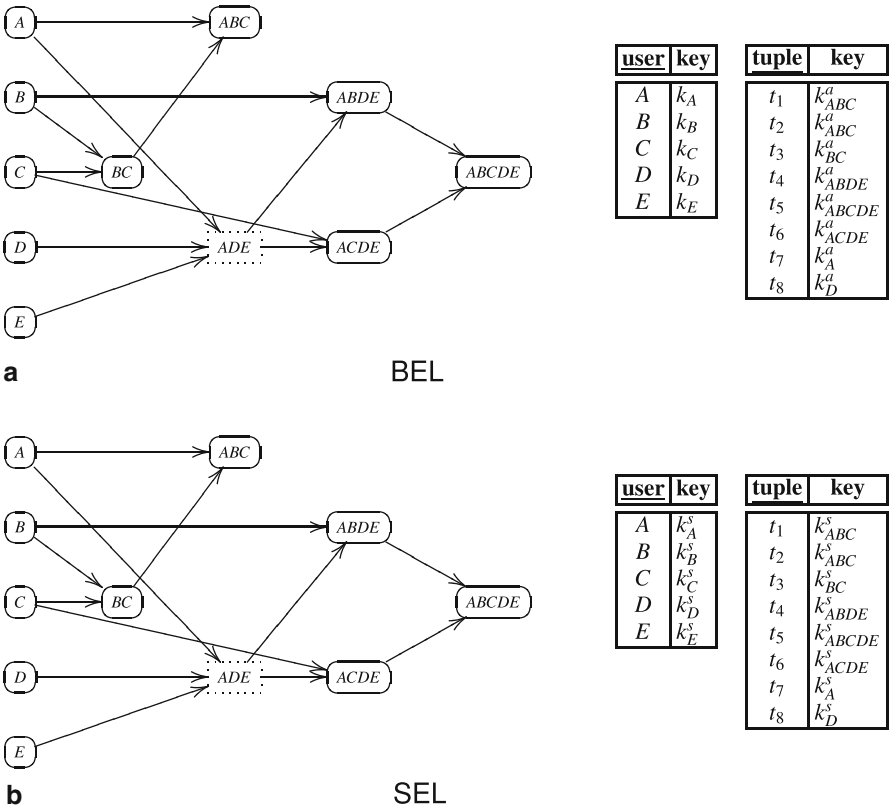
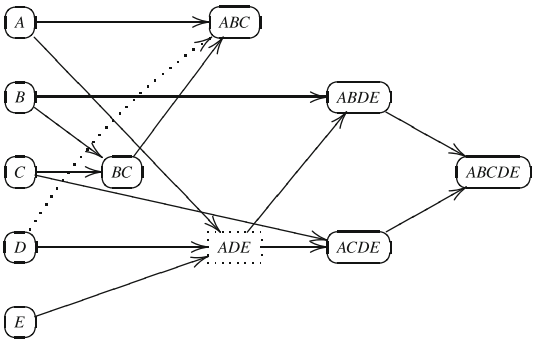


Fig. 6 Encryption policies at BEL and SEL, equivalent to the access control policy in Fig. 3

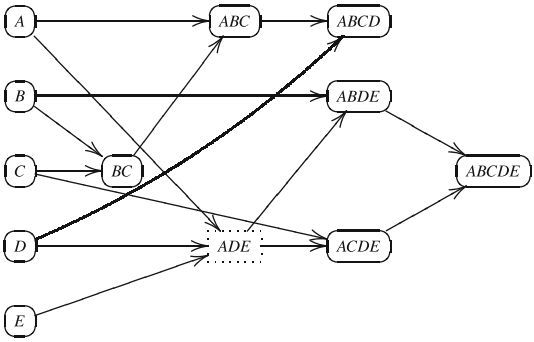
to add a vertex representing *ABCD*. Tuple t_1 is then over-encrypted at SEL with the key of this new vertex.

- *Revoke.* When user u loses the privilege of accessing tuple t, the data owner simply asks the server to re-encrypt (at SEL) the tuple with the key associated with the set $acl(t) \setminus \{u\}$ of users. If the vertex representing this set of users is not represented in the SEL key derivation hierarchy, the server first updates the hierarchy inserting the new vertex, and then re-encrypts the tuple. For instance, consider the encryption policies at BEL and SEL in Fig. 7 and assume that the data owner revokes *B* the privilege to access t_4. The data owner requires the server to change SEL (BEL is not affected by revoke operations) to guarantee that tuple t_4 is encrypted with a key that user *B* cannot derive. To this aim, t_4 is re-encrypted with key k^s_{ADE}. Figure 8 illustrates the encryption policies at BEL and SEL after the enforcement of the revoke operation. Note that vertex *ABDE* is removed from the hierarchy since it is neither necessary for policy enforcement nor useful for reducing the number of tokens.



user	key
A	k_A
B	k_B
C	k_C
D	k_D
E	k_E

tuple	key
t_1	k^a_{ABC}
t_2	k^a_{ABC}
t_3	k^a_{BC}
t_4	k^a_{ABDE}
t_5	k^a_{ABCDE}
t_6	k^a_{ACDE}
t_7	k^a_A
t_8	k^a_D

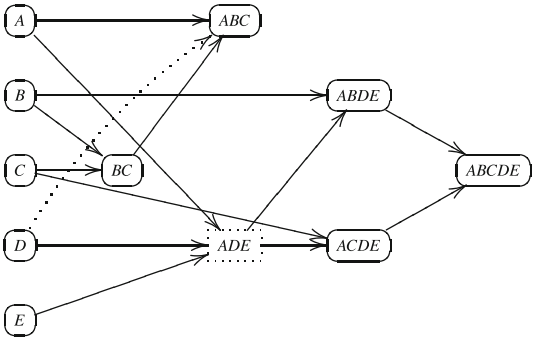


user	key
A	k^s_A
B	k^s_B
C	k^s_C
D	k^s_D
E	k^s_E

tuple	key
t_1	k^s_{ABCD}
t_2	k^s_{ABC}
t_3	k^s_{BC}
t_4	k^s_{ABDE}
t_5	k^s_{ABCDE}
t_6	k^s_{ACDE}
t_7	k^s_A
t_8	k^s_D

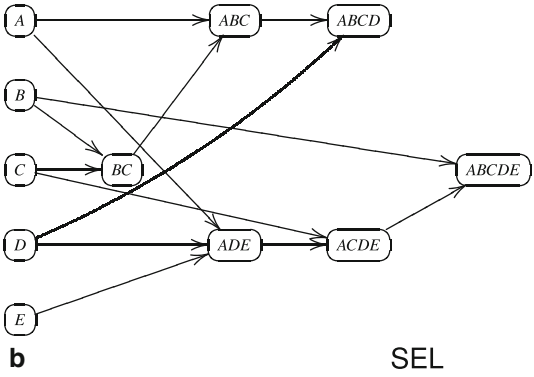
Fig. 7 Encryption policies at BEL and SEL in Fig. 6 after granting D access to t_1

Since the management of (re-)encryption operations at SEL is delegated to the server, there is the risk of collusions with users. In fact, by combining their knowledge, a user and the server can possibly decrypt tuples that neither the server nor the user can access. For instance, with reference to the encryption policy in Fig. 8, the server and user D can access to tuple t_2 by combining their knowledge. In fact, this tuple is encrypted with access key k^a_{ABC} at BEL, known to user D as it is used to encrypt t_1, and with key k^s_{ABC} at SEL, known to the server. Collusion represents a risk to the correct enforcement of the authorization policy, but this risk is limited. In fact, collusion between a user u and the server permits them to decrypt a tuple t that they are not authorized to access only if u is granted the privilege to read a tuple t' (different from t) that is encrypted with the same key as t at BEL. Indeed, u knows the key with which t is encrypted at BEL (as it is necessary to access t') while the server knows the key with which it is encrypted at SEL (as it manages all the encryption keys at SEL). Collusion risk can then be mitigated at the price of using a higher number of keys at BEL, that is, by using the same encryption key at BEL only for tuples whose *acl*s are likely to evolve in the same way [12].



user	key
A	k_A
B	k_B
C	k_C
D	k_D
E	k_E

tuple	key
t_1	k^a_{ABC}
t_2	k^a_{ABC}
t_3	k^a_{BC}
t_4	k^a_{ABDE}
t_5	k^a_{ABCDE}
t_6	k^a_{ACDE}
t_7	k^a_A
t_8	k^a_D



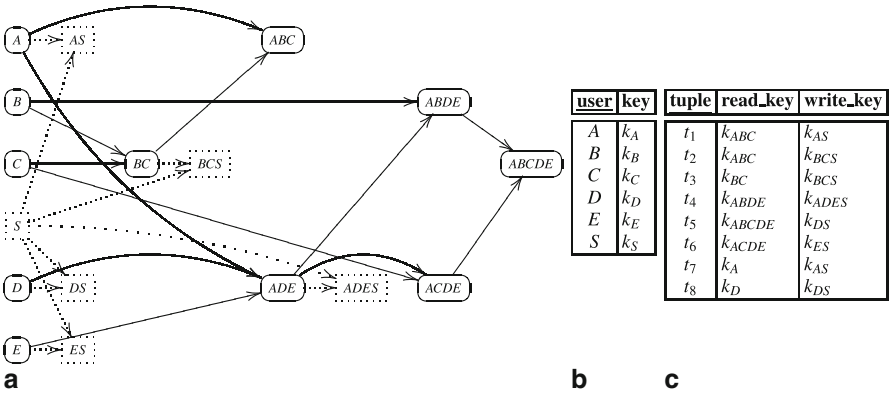
user	key
A	k^s_A
B	k^s_B
C	k^s_C
D	k^s_D
E	k^s_E

tuple	key
t_1	k^s_{ABCD}
t_2	k^s_{ABC}
t_3	k^s_{BC}
t_4	k^s_{ADE}
t_5	k^s_{ABCDE}
t_6	k^s_{ACDE}
t_7	k^s_A
t_8	k^s_D

Fig. 8 Encryption policies at BEL and SEL in Fig. 7 after revoking B access to t_4

2.3 Write Privileges

The solution described in the previous section, while effectively enforcing read privileges and updates to them, assumes the outsourced relation to be read-only (i.e., only the owner can modify tuples). To allow the data owner to selectively authorize other users to update the outsourced data, this approach has been complemented with a specific technique to manage write privileges. The approach in [11] associates each tuple with a *write tag* (i.e., a random value independent from the tuple content) defined by the data owner. Access to write tags is regulated through selective encryption: the write tag of tuple t is encrypted with a key known only to the users authorized to write t (i.e., the users specified within its write access list, denoted $acl_w t$) and by the server. In this way, only the server and authorized writers have access to the plaintext write tag of each tuple. The server will then accept a write request on a tuple when the requesting user proves knowledge of the corresponding write tag.



user	key
A	k_A
B	k_B
C	k_C
D	k_D
E	k_E
S	k_S

tuple	read_key	write_key
t_1	k_{ABC}	k_{AS}
t_2	k_{ABC}	k_{BCS}
t_3	k_{BC}	k_{BCS}
t_4	k_{ABDE}	k_{ADES}
t_5	k_{ABCDE}	k_{DS}
t_6	k_{ACDE}	k_{ES}
t_7	k_A	k_{AS}
t_8	k_D	k_{DS}

Fig. 9 Encryption policy in Figs. 5c–d extended to the enforcement of write privileges

Since the key used for encrypting the write tag of a tuple has to be shared between the server and the tuple writers, it is necessary to extend the key derivation hierarchy with the storing server. However, the server cannot access the outsourced tuples in plaintext, and hence it cannot be treated as an additional authorized user (i.e., with the ability of deriving keys in the hierarchy). The keys used to encrypt write tags are then 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 server can directly derive them from a key k_S assigned to it, through a token specifically added to the key derivation hierarchy. Note that keys used for encrypting write tags cannot be used to derive other keys in the hierarchy. For instance, consider the encryption policy in Figs. 5(c–d) and assume that $acl_w(t_1) = acl_w(t_7) = A$, $acl_w(t_2) = acl_w(t_3) = BC$, $acl_w(t_4) = ADE$, $acl_w(t_5) = acl_w(t_8) = D$, and $acl_w(t_6) = E$. Figure 9a illustrates the key derivation hierarchy, extended with the key k_S assigned to the server S and the keys necessary to encrypt write tags (the additional vertices and edges are dotted in the figure). Figures 9b–c summarize the keys assigned to users and to the server, and the keys used to encrypt the tuples in relation PATIENTS and their write tags, respectively.

The over-encryption approach (Sect. 2.2), while effective for enforcing updates to a read authorization policy, cannot unfortunately be adopted to enforce grant and revoke of write authorizations. A possible way to enforce dynamic write privileges [11] operates as follows.

- *Grant.* When user u is granted the privilege to modify tuple t, the write tag of t is encrypted with a key known to the server and the users in $acl_w(t) \cup \{u\}$. If the key derivation hierarchy does not include it, such a key is created and properly added to the hierarchy. For instance, with reference to the encryption policy in Figure 9, assume that user B is granted the write privilege over t_4. The write tag of the tuple needs to be encrypted with key, k_{ABDES}, which is inserted into the key derivation hierarchy, while key k_{ADES} can be removed.

- *Revoke.* When user u is revoked the write privilege over tuple t, a fresh write tag must be defined for t, having a value independent from the former tag (e.g., it can be chosen adopting a secure random function). This is necessary to ensure that u, who is not oblivious, cannot exploit her knowledge of the former write tag of tuple t to perform unauthorized write operations. After the tag has been generated, it is encrypted with a key known to the server and to the users in $acl_w(t) \setminus \{u\}$. For instance, with reference to the encryption policy in Fig. 9, assume that user C is revoked the write privilege over t_3. The write tag of the tuple needs to be changed and encrypted with key k_{BS}, which should be inserted into the key derivation hierarchy.

Note that, since the server is authorized to know the write tag of each and every tuple to correctly enforce write privileges, the data owner can delegate to the storing server both the generation and encryption (with the correct key) of the write tag of the tuples [11].

2.4 Attribute-Based Encryption

An alternative solution to selective encryption for the enforcement of access restrictions is represented by *Attribute-Based Encryption* (ABE [29]). ABE is a particular type of public-key encryption that regulates access to tuples on the basis of policies defined on descriptive attributes, associated with tuples and/or users. ABE can be implemented as either *Ciphertext-Policy ABE* (CP-ABE [47]), or *Key-Policy ABE* (KP-ABE [29]), depending on how attributes and authorization policies are associated with tuples and users. Both the strategies have been recently widely investigated, and several solutions have been proposed, as briefly illustrated in the following.

CP-ABE CP-ABE associates with each user u a set of descriptive attributes, and a private key that is generated on the basis of these attributes. Each tuple t in a relation r is associated with an *access structure* modeling the access control policy regulating accesses to t. Graphically, an access structure is a tree whose leaves represent attributes and whose internal nodes represent logic gates (e.g., conjunctions and disjunctions). Figure 10 illustrates an example of access structure associated with tuple t_2 in relation PATIENTS in Fig. 2. This access structure corresponds to the Boolean formula (job = 'doctor' $\vee$ job = 'nurse') $\wedge$ $ward$ = 'neurology', meaning that only doctors or nurses working in the neurology ward can access the medical data of *Barbara* (i.e., tuple t_2). CP-ABE key generation technique guarantees that the key k of user u can decrypt tuple t only if the set of attributes used when generating k satisfies the access policy represented by the access structure considered when encrypting t. Although CP-ABE effectively and efficiently enforces access control policies, one of its main drawbacks is related to the management of attribute revocation. Intuitively, when a user loses one of her attributes, she should not be able to access tuples that require the revoked attribute for the access —which however is hard to enforce while guaranteeing efficiency. A solution to this problem is presented in

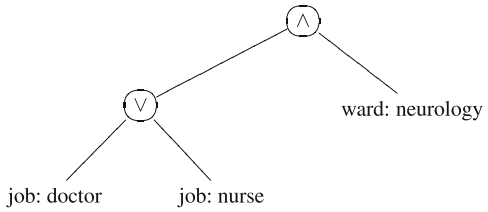


Fig. 10 An example of access structure associated with tuple t_2 of relation PATIENTS in Fig. 2

[51], where 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 attribute revoked to the user) 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 [44], the authors instead define a hierarchical attribute-based solution that relies on an extended version of CP-ABE in which attributes associated with users are organized in a recursive set structure, and propose a flexible and scalable approach to support revocations.

KP-ABE KP-ABE associates an access structure with each user and a set of descriptive attributes with each tuple. 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. Thanks to the properties of KP-ABE key generation techniques, each user u can decrypt only tuples t such that the attributes of tuple t satisfy the access structure associated with user u. Since ABE is based on public-key encryption, to reduce the overhead caused by asymmetric encryption, the tuple content can be encrypted with a symmetric key, which is in turn protected through KP-ABE [53]. Only authorized users can remove the KP-ABE encryption layer to retrieve the symmetric key use to protect the content of the tuples. This solution also efficiently supports policy updates and couples ABE with proxy re-encryption to delegate to the storing server most of the re-encryption operations necessary to enforce policy updates.

The support of write privileges is provided by the adoption of Attribute-Based Signature (ABS) techniques. The proposal in [21] 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 [38]. This solution has the advantage over the approach in [21] of being suited also to distributed scenarios.

3 Efficient Access to Encrypted Data

Since data stored in the cloud are encrypted for confidentiality reasons, the storing server cannot directly evaluate users' queries since it is not trusted to access the data content. This makes access to outsourced data time consuming and computationally

PATIENTS

	SSN	Name	ZIP	MarStatus	Illness
t_1	123456789	Ann	22010	single	gastritis
t_2	234567891	Barbara	24027	widow	neuralgia
t_3	345678912	Carl	22010	married	gastritis
t_4	456789123	Daniel	20100	married	gastritis
t_5	567891234	Emma	21048	single	neuralgia
t_6	678912345	Fred	23013	married	hypertension
t_7	789123456	Gary	22010	widow	gastritis
t_8	891234567	Harry	24027	divorced	hypertension

a

PATIENTS^k

tid	enc	I_z	I_m	I_i
1	aD4%l	α	ζ	η
2	7Eoi)	β	κ	ξ
3	Gx?b3	α	θ	μ
4	dn4$z	γ	θ	η
5	2Cyl=	δ	ζ	ξ
6	Joi2s	ε	θ	ρ
7	(ceWm	α	κ	μ
8	w2!qk	β	κ	ρ

b

Fig. 11 An example of plaintext relation (**a**) and the corresponding encrypted and indexed relation (**b**)

expensive (the client would need to download the data and locally evaluate her query). To limit such an overhead, either *keyword search* or *index*-based approaches can be adopted, which enable query evaluation at the server side without the need to decrypt data [39]. Keyword search techniques (e.g., [6, 8, 25, 42, 45]) permit to search for documents including a keyword of interest in an encrypted data collection. Indexes are metadata that depend on the plaintext values of the attributes in the original relation, and are stored 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 has schema $R^k(\underline{\texttt{tid}}, \texttt{enc}, I_{i_1}, \ldots, I_{i_j})$, where $I_{i_l}, l = 1, \ldots, j$, 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, Fig. 11b represents an example of an encrypted version of relation PATIENTS in Fig. 2 (also reported in Fig. 11a for the reader' s convenience), where attributes `ZIP`, `MarStatus`, and `Illness` are associated with indexes I_z, I_m, and I_i, respectively. In this and in the following examples, for readability, we will denote index values with Greek letters.

To provide efficient access to the outsourced data collection, different indexing techniques have been proposed, aimed at supporting the server-side evaluation of a variety of conditions and clauses in SQL queries. The most important indexing approaches can be classified in three main categories, depending on how the index function ι maps the original attribute values to the corresponding index values, as illustrated in the following.

- *Direct Index*. Each plaintext value is represented by a different index value and vice versa. An example of direct index (e.g., [9]) is adopted by encryption-based indexes, which map plaintext value *val* to index value $E_k(val)$, where E is a symmetric encryption function with key k. Index I_z in Fig. 11b is an example of a direct index defined over attribute `ZIP` in Fig. 11a.
- *Bucket-based Index*. Each plaintext value is represented by one index value, but different plaintext values are mapped to the same index value, generating collisions. There are different approaches for defining which plaintext values are

represented by the same index value. The two most common techniques are partition-based and hash-based indexes. Partition-based indexes (e.g., [31]) partition the domain D of attribute a into subsets of contiguous values and associate a label with each of them. The index value representing a value in a partition is the label of the partition. Hash-based indexes (e.g., [5]) instead rely on a secure hash function h generating collisions. Given plaintext value *val*, its corresponding index value is computed as $h(val)$. Index I_m in Fig. 11b is an example of hash-based index over attribute `MarStatus` in Fig. 11a, where values divorced and widow generate a collision and are both represented by index value κ.

- *Flattened Index*. Each plaintext value is represented by different index values, each characterized by the same number of occurrences (flattening). Each index value, however, represents one plaintext value only. A flattened index can be obtained by properly combining encryption with a flattening post-processing that guarantees that the frequency of index values be the same (e.g., [46]). Index I_i in Fig. 11b is an example of a flattened index over attribute `Illness` in Fig. 11a, where plaintext value gastritis is represented by index values η and μ.

Intuitively, the fact that the outsourced relation is encrypted and enriched with indexes must be transparent to the final users. The basic indexing techniques illustrated above nicely support the server-side evaluation of simple SQL queries including equality conditions in the WHERE clause. Consider a query q submitted by a user of the form "SELECT *Att* FROM *R* WHERE *Cond*", where $Att \subseteq R$ and *Cond* is a set of equality conditions of the form $a = val$, with $a \in R$ and *val* a constant value in the domain D of a. To determine the query that should be submitted to the storing server, each condition $a = val$ in *Cond* is first translated into an equivalent condition of the form: $I = \iota(val)$, if I is a direct or a bucket-based index; and I IN $\iota(val)$, if I is a flattened index and hence $\iota(val)$ may return a set of values. The query q_s submitted to the server is then "SELECT `enc` FROM R^k WHERE $Cond^k$", where $Cond^k$ is obtained as illustrated above. The result returned by the server must then be decrypted by the client, to retrieve the plaintext content of the tuples. The client may also need to perform a projection over the attributes in *Att*, if they represent a proper subset of R, and to filter spurious tuples, that is, tuples that satisfy $Cond^k$ but that do not belong to the query result (i.e., they do not satisfy *Cond*). Note that the presence of spurious tuples may depend on collisions possibly caused by bucket-based indexes, where multiple plaintext values are mapped to the same index value. The client then evaluates a query q_c of the form "SELECT *Att* FROM $D(Res^k)$ WHERE *Cond*", where Res^k is the relation returned by the server as the result of the evaluation of query q_s. The result of query q_c is returned to the requesting user. Consider, as an example, a query q = SELECT `SSN, Name` FROM PATIENTS WHERE `MarStatus` = "widow" AND `Illness` = "gastritis" operating on relation PATIENTS in Fig. 11a. The query q_s to be sent to the server is SELECT `enc` FROM PATIENTS^k WHERE $I_m = \kappa$ AND I_i IN $\{\eta,\mu\}$, which returns tuple t_7. The client will then decrypt the result returned by the server and evaluate query SELECT `SSN, Name` FROM $D(Res^k)$ WHERE `MarStatus` = "widow" AND `Illness` = "gastritis" to check whether tuple t_7 satisfies both the conditions and to project the attributes of interest for the requesting user.

Besides the techniques illustrated and classified above, many other approaches have been proposed for efficiently delegating to the server the evaluation of complex conditions and/or SQL clauses. As an example, order preserving encryption has been proposed as an effective solution for supporting range conditions, as well as grouping and ordering clauses (e.g., [1, 46]). Aggregate functions can instead be computed if the index over the attribute of interest has been defined through homomorphic encryption techniques, which support the evaluation of arithmetic operators on encrypted data (e.g., [24, 30]). Different techniques, which do not fit into the classification above, have also been proposed to the aim of enjoying the advantages of traditional database indexing techniques also in the outsourcing scenario (e.g., in [9] the authors propose to use encrypted $B+$-trees for the efficient evaluation of range queries).

4 Protecting Access Privacy

Besides protecting the confidentiality of the outsourced data collection, it is also paramount to protect the privacy of the accesses to the data themselves. In fact, queries can be exploited for inference, making both users' and data privacy at risk. As an example, consider a scenario where the outsourced data contain medical information. Revealing that a user submitted a query looking for the symptoms of lung cancer implicitly reveals that (with high probability) either her or a person close to her suffers from such a disease. Also, users accesses may be exploited to infer the private content of the outsourced data collection. Indeed, by monitoring patterns of frequently accessed tuples, an observer can draw inferences on their specific values thanks to additional knowledge she may have on how frequently each piece of data in a given domain is accessed. In this case, it is necessary to protect both *access confidentiality* (i.e., each query singularly taken) and *pattern confidentiality* (i.e., the fact that two queries aim at the same target value). A first attempt to protect access confidentiality is represented by keyword search approaches (e.g., [6, 8, 25, 42, 45]), which do not reveal to the server any information about the outsourced data and the target keyword. A similar approach consists in defining a set of tokens that can be adopted by users to evaluate queries on outsourced data without disclosing the conditions in their queries to the storage server [18, 36]. Protection of accesses to a $B+$-tree index structure can instead be obtained by grouping the nodes in the tree into buckets [37]. The use of homomorphic encryption techniques then permits to access the node of interest in each bucket, while preventing the server from precisely identifying the node target of each access. These approaches represent a first step towards the definition of privacy-preserving indexing approaches, but they fall short in protecting the confidentiality of repeated accesses and, more in general, of patterns thereof. In the remainder of this section, we will illustrate some of the most important approaches recently proposed to address both access and pattern confidentiality in a scenario where data need to remain confidential (i.e., outsourced data are encrypted).

4.1 Oblivious RAM

One of the first approaches [49] proposed to protect access and pattern confidentiality in a scenario where also the confidentiality of the data must be protected is based on the *Oblivious RAM* (ORAM) data structure [26]. The outsourced database is organized as a set of n encrypted blocks, which are stored in a pyramid-shaped data structure. Each level l of the ORAM structure stores 4^l blocks and is characterized by a Bloom filter and a hash table that permit to quickly determine whether an index value is stored in the level and, if this is the case, to identify the block where it is stored. Access and pattern confidentiality are provided by guaranteeing that: *(i)* the search process does not reveal the level in the structure where the target block is stored, and *(ii)* a block in the hash table is never accessed more than once with the same search key.

The search algorithm visits the ORAM structure level by level, starting from the top of the pyramid. For each level l, the search algorithm uses the Bloom filter to determine whether the target of the search is stored in the level. If this is the case, the item of interest is extracted from the level (by accessing the block identified by the hash table), decrypted, re-encrypted with a different nonce, and inserted into a cache. Otherwise, the algorithm extracts a random element from the level and inserts it into the cache. We note that, even when the target element is retrieved, the search process visits all the lower levels in the ORAM structure extracting at each level a random (fake) element. This guarantees that, by observing accesses to the structure, the server is not able to identify the level where the target of the search was stored. The search process terminates when the last level in the ORAM structure is visited.

When the cache is full, it is merged with the first level of the ORAM structure and the items in the resulting new level are re-shuffled, to destroy any correspondence between old and new items in the level. As a consequence, the Bloom filter associated with the level is re-defined, to correctly refer to the new level content. The same process applies to each level in the structure: when level l is full, it is merged with level $l+1$, the blocks are re-shuffled, and the Bloom filter is redefined. The cost of accessing the ORAM clearly depends on the possible need to reorganize a level in the indexing structure while visiting it, and on the specific level that needs to be redefined. The amortized cost per query, which takes into consideration the impact of periodic reorganizations of the structure, is $O(\log n \log\log n)$, under the assumptions of $O(\sqrt{n})$ temporary client storage and of $O(n)$ server storage overhead. However, the cost of reorganizing the bottom level of the pyramid is $O(n)$, where n is the number of index values in the dataset. Response time of any access request submitted during the reordering of lower levels of the database is therefore high and not acceptable in many real-world scenarios.

To mitigate the cost of query evaluation when low levels in the ORAM structure need to be reorganized, the proposal in [20] puts forward the idea of limiting the shuffling operation to the blocks that store accessed tuples. This approach is based on the presence of a secure coprocessor on the server that locally manages a cache of size k, which is empty at initialization time. Each tuple in the dataset is associated

with a label, initially set to value 'white'. Once a tuple is accessed, its label becomes 'black'. For each access to the dataset, the search algorithm fetches a black tuple and a white tuple. If the target tuple is already in cache, the algorithm retrieves two randomly chosen fake tuples (a black one and a white one), otherwise it accesses the target tuple and a randomly chosen fake tuple. When the cache is full, the secure coprocessor shuffles black tuples (performing a partial shuffling) only and re-encrypts them before re-writing the blocks on the server. Partial shuffling provides access and pattern confidentiality, since white tuples have not been accessed and hence it is not necessary to move their content to hide the traces that an access could have left. The amortized cost per query of this solution is $O(\sqrt{n \log n/k})$, which is lower than the proposal in [49]. It however relies on a secure coprocessor for guaranteeing access and pattern confidentiality.

Alternative techniques that can be adopted to limit the response time of the ORAM structure are based on the idea of minimizing the number of interactions between the client and the server [27, 48]. Indeed, the communication costs have a high impact on response times and reducing the number of interactions provides benefits to users. Other approaches instead rely on enhancing the support of concurrent accesses [28, 50]. These solutions basically define copies of the levels of the ORAM structure. Searches operate on a read-only copy of the level of interest, while the master copy of the same level is dynamically updated and used for reordering purposes only. In this way, exclusive locks blocking access to a level of the structure during its reorganization process do not delay users' accesses.

Path-ORAM has recently been proposed as an alternative approach to provide access and pattern confidentiality without paying the high price of re-shuffling, which characterizes traditional ORAM structures [43]. Path-ORAM is a tree-shaped data structure whose nodes are buckets storing a fixed number of blocks (which can either be dummy or contain actual data). Each block is mapped to a randomly chosen leaf in the tree and it is stored either at the client side (in a local cache, which is called stash) or in one of the buckets along the path to the leaf to which it is associated. Read operations download from the server and store in the stash all the buckets in the path from the root to the leaf to which the block of interest is mapped. The mapping of the target block is randomly updated (i.e., the block is mapped to a new, randomly chosen, leaf). The path read from the server is then written back, inserting into the buckets the blocks in the local stash (provided the bucket is along the path to the leaf to which the block is mapped).

4.2 Dynamically Allocated Data Structures

Dynamic data allocation solutions aim at destroying the otherwise static relationship between disk blocks and the information they store. These approaches are based on the definition of a dynamically allocated index structure (e.g., a $B+$-tree, a hash table, a flat index) that guarantees private and efficient access to the data.

If the data are organized in a tree-shaped index structure, access confidentiality is provided by guaranteeing that the storing server does not know (nor can infer) which is the node in the tree target of the access, as it would otherwise reveal the value target of the search. The first step to protect the confidentiality of the dataset content consists in encrypting the nodes in the tree before outsourcing, and in storing each encrypted node in a different disk block. However, repeated accesses to the same physical block inevitably represent repeated accesses to the same node content and hence queries aiming to the same value (or to values within a small interval). If the storing server knows the relative frequency of accesses to the plaintext values, it can reconstruct the correspondence between node contents and encrypted blocks, by simply matching access frequencies. A preliminary approach aimed at protecting access confidentiality through a privacy-preserving tree relies on the combined adoption of the following three protection techniques [35]:

- *access redundancy*: each access request visits, besides the node target of the access, m additional blocks (at least one of which should be empty) for each level in the tree to hide the target of the access in a set of $m + 1$ equally-probable candidate nodes;
- *node swapping*: the node target of the access is swapped with one of the empty blocks downloaded from the server for the same level, meaning that the target node is stored in an empty block and viceversa;
- *node re-encryption*: all the nodes downloaded from the server are re-encrypted, to hide the swap.

Although effective for protecting content and access confidentiality, this proposal falls short in providing pattern confidentiality, since frequently accessed blocks can easily be identified by the server and then exploited for inference purposes.

An alternative approach, which does not operate on a tree-shaped index structure, is based on the adoption of a lightweight scheme that provides access and pattern confidentiality by combining the following three protection techniques [52]:

- *dummy data items*: each access request visits, besides the block target of the access, two additional blocks;
- *swapping*: the target of the access is swapped with one of the dummy data items downloaded from the server;
- *repeated patterns*: dummy data items are selected in such a way that, out of the three blocks downloaded from the server, two (and only two) are among the ones accessed during the previous search.

The goal of the combined adoption of these three protection techniques is to make each access to the outsourced data collection indistinguishable from the server's point of view. In fact, each access has two blocks in common with the previous one, while the third one is fresh. Swapping protects repeated accesses and is combined with re-encryption of the content of all the accessed blocks, to prevent the server from reconstructing which swap has been performed (thus possibly recognizing repeated accesses).

4.3 *Shuffle Index*

A recent technique addressing the need of providing efficient query execution, while protecting access and pattern confidentiality, is based on the definition of a *shuffle index* [14].

Data Structure A 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 different abstraction levels (i.e., abstract, logical, and physical), as illustrated in the following. At the abstract level, the shuffle index is an *unchained B+-tree* with fan-out F, built over a candidate key K of relation r. Each internal node of the tree represents the root of a sub-tree with $q \geq \lceil F/2 \rceil$ children (except for the root node, where $1 \leq q \leq F$), and stores $q-1$ ordered values $val_1 \leq \ldots \leq val_{q-1}$ of attribute K. The leaves store the tuples of the outsourced relation, together with their key value, but (in contrast to traditional $B+$-tree structures) are not connected in a chain, so not to allow the server storing the data to discover the relative order among the values in the leaves. Figure 12a illustrates an example of unchained $B+$-tree.

At the logical level, each node n of the unchained $B+$-tree 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, at the logical level, through the identifier of child nodes. Figure 12b illustrates an example of logical representation of the unchained $B+$-tree in Fig. 12a. Note that the order of logical identifiers does not necessarily reflect the value-order relationship between the node contents. For readability, in the figure nodes are ordered according to their logical identifier (reported on the top of each node), whose first digit corresponds to the level of the node in the tree.

At the physical level, each 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 server side (for simplicity, we assume that the physical address of a block coincides with the logical identifier of the corresponding node). Figure 12c illustrates the physical representation of the logical index in Fig. 12b. Note that the physical representation of the shuffle index coincides with the view of the storage server over the outsourced data collection. In fact, although the server does not have knowledge of the encryption key, it can establish the level in the tree associated with each block by observing a long enough history of accesses to the $B+$-tree structure, because accesses visit the tree level by level.

Protection Techniques To protect content, access, and pattern confidentiality, encryption is complemented with three protection techniques: *cover searches*, *cached searches*, and *shuffling*. These protection techniques apply to every access to the shuffle index, which proceeds by visiting the $B+$-tree level by level from the root to the leaves.

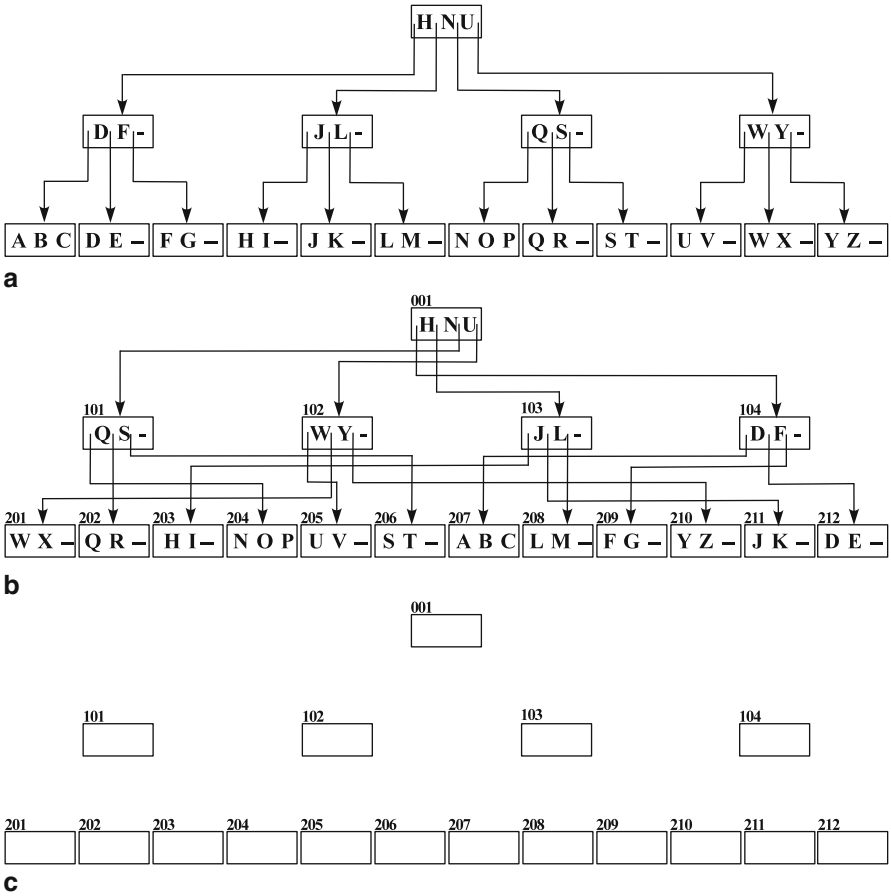


Fig. 12 An example of abstract (**a**), logical (**b**), and physical (**c**) representation of a shuffle index

- *Cover searches.* Cover searches aim at hiding the target of an access within a set of other potential targets, in such a way that the server cannot recognize the value of interest for the user. Cover searches are *fake* searches, which are not recognizable as such by the storage server, that are executed in conjunction with the search for the target value (i.e., the value of interest for the requester). 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, at the server's eyes, each of the $num\_cover + 1$ leaf blocks accessed during a visit of the shuffle index has the same probability of storing the target. To provide sufficient protection to the target of the access, cover searches must guarantee: *(i)* indistinguishability with respect to target searches, meaning that the server should not be able to determine whether an accessed block

is a cover or the target; and *(ii)* block diversity, meaning that paths to covers and to the target must be disjoint (except for the root node).

- *Cached searches.* Cached searches aim at protecting repeated accesses to a node content, by making them indistinguishable from non-repeated accesses to the eyes of the server. The cache is a layered structure with a layer for each level in the shuffle index. It is maintained 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. Being stored at a trusted party, the cache is maintained in plaintext. Each layer of the cache is managed according to the Least Recently Used (LRU) policy, which guarantees the property that the parent of each cached node (and hence also the path connecting it to the root of the tree) is also in cache. Whenever the target of an access is in cache, it is replaced by an additional cover for the access, to guarantee that $num\_cover + 1$ blocks are downloaded for each level of the tree (but the root level). This makes repeated accesses look like accesses to nodes that have not been previously accessed. The adoption of a local cache prevents short-time intersection attacks, which could be exploited by the server to identify repeated accesses when subsequent searches download non-disjoint sets of blocks. In fact, accesses within a time frame of *num_cache* accesses do not have nodes in common.
- *Shuffling.* Shuffling aims at breaking the relationship between node content and block where it is stored, to avoid that accesses to the same physical block correspond to accesses to the same node content. By changing the node-block allocation, the server cannot draw conclusions on the content of the accessed nodes by observing accessed blocks. In fact, repeated accesses to the same block do not necessarily correspond to repeated 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 (i.e., shuffling assigns a different block address to each accessed node, choosing among the downloaded blocks). To prevent the server from reconstructing node shuffling, every time a node content is moved to a different block, it is re-encrypted using a different random salt. Its parent is also updated to guarantee that the parent-child relationship between them is preserved.

Search Process Each search operation for a value then combines these three protection techniques, as described in the following.

Given the value *target_value*, target of the access to the outsourced relation, the search algorithm (operating at the client side) first randomly chooses a set of $num\_cover + 1$ cover values in the actual domain of the key attribute K on which the shuffle index has been defined. Since these values should act as cover searches for *target_value*, this choice must guarantee both indistinguishability and block diversity, as described above. Note that the algorithm chooses one additional cover (i.e., $num\_cover + 1$ instead of *num_cover*) as it is needed if the target of the access is in the local cache. The search algorithm then visits the shuffle index level by level, starting from the root. For each level l of the shuffle index, the search algorithm first checks whether the node in the path to *target_value* is in the local cache and, if a cache miss occurs, it discards one of the cover searches initially chosen proceeding the search

with *num_cover* covers. It then determines the address of the blocks storing the nodes along the paths to *target_value* and to the *num_cover* cover searches (*num_cover* + 1 covers if a cache hit occurred). These blocks are then downloaded from the server, decrypted to retrieve the content of the nodes they store, and randomly shuffled together with the nodes at level l in the local cache. To preserve the correctness of the shuffle index data structure, the parents of shuffled nodes are updated, guaranteeing that pointers refer to the blocks where the children of each node are stored. The search algorithm also updates the local cache structure, according to the LRU policy. If the node along the path to *target_value* is in cache, the algorithm simply refreshes its timestamp; otherwise, the node along the path to the target is inserted as the most recently accessed node and the least recently accessed node is removed from the cache. Before moving to the next level, the nodes shuffled during the previous iteration (i.e., accessed and cached nodes at level $l - 1$) are encrypted with a fresh random salt and sent to the server for storage. Upon receiving the encrypted blocks, the server replaces the old block stored at each physical address with the new one received from the client. The process terminates when the visit of the shuffle index reaches the leaf level. The leaf along the path to *target_value* is then returned to the requesting user, since it contains the tuple with value *target_value* for attribute K, if such a tuple exists in r. For instance, consider a search for value 'W' on shuffle index in Fig. 12 that adopts one cover. Also, assume that the cache has size 2 and that it stores: the root node at level 0; nodes 103[J,L,–] and 102[W,Y,–] at level 1; and leaves 211[J,K,–] and 210[Y,Z,–] at level 2. The client first chooses two covers for the target 'W', say 'E' and 'Q', and visits the root node (block 001), which is stored in the local cache. It then identifies the block at level 1 along the paths to the target (i.e., 102) and to the two covers (i.e., 104 and 101, respectively). Since block 102 is in cache, the client downloads from the server blocks 104 and 101, decrypts their content, and shuffles the accessed and cached blocks (i.e., 101, 102, 103, and 104) as illustrated in Fig. 13b. It then updates the pointers to children in the root node, encrypts its content and sends it back to the server for storage. Moving to the next level, the client first identifies the leaf blocks along the path to the target (i.e., 201) and to the two covers (i.e., 212 and 202, respectively). Since block 201 is not in cache, one of the two covers is discarded, say 202, and the client downloads from the server and decrypts blocks 201 and 212. The client then shuffles blocks 201, 210, 211, and 212, updates the pointer to them in their parents, encrypts nodes 101, 102, 103, and 104, and sends the resulting blocks to the server for storage. Then, it updates the cache at level 2 inserting leaf node 212[W,X,–] and removing leaf node 211[Y,Z,–]. Finally, the client encrypts the shuffled leaves and sends the resulting blocks to the server. Figure 13c illustrates the logical shuffle index resulting after the access.

The search algorithm operates in logarithmic time in the size of the outsourced database (i.e., its computational complexity is $O((1+ num\_cover + num\_cache) \log_F (n))$, with n the number of tuples in r), since for each search the algorithm visits *num_cover* + 1 different paths of the shuffle index.

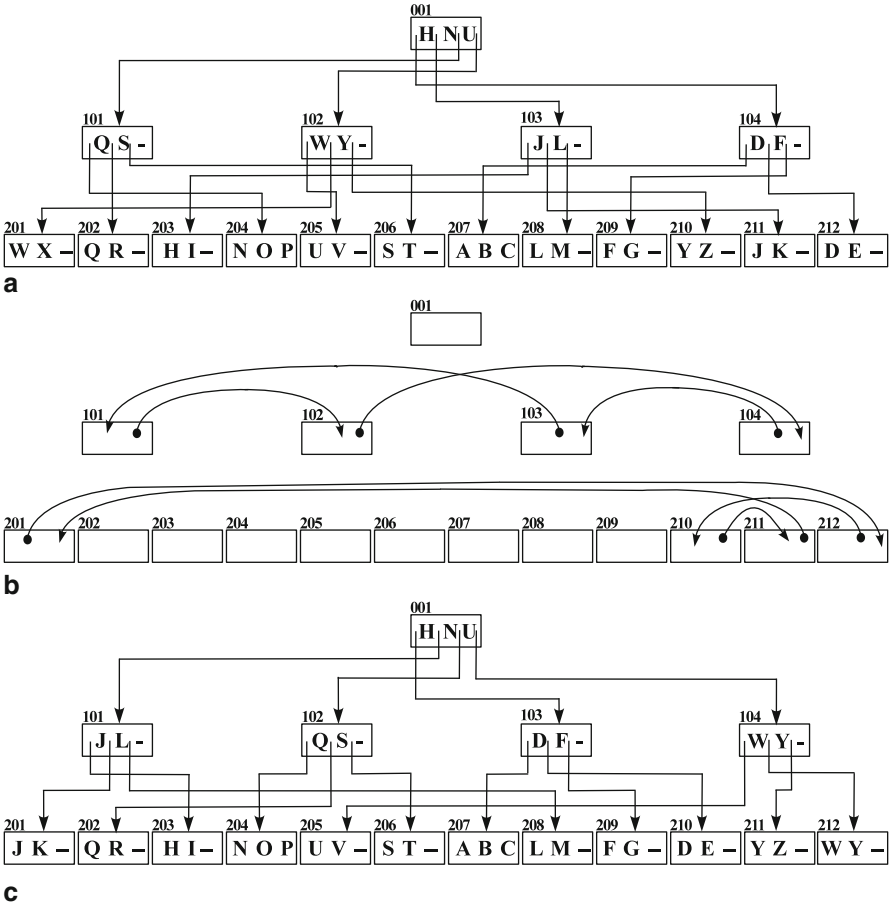


Fig. 13 An example of evolution of the logical shuffle index in Fig. 12b as a consequence of a search for value 'W' with 'E' and 'Q' as covers

Extensions of the Shuffle Index The original shuffle index proposal has been extended in several directions to support: *(i)* concurrent accesses to the data; *(ii)* searches over attributes different from K; and *(iii)* data storage at different servers. Concurrency is provided by the adoption of *delta versions* [17], which are copies of portions of the shuffle index that are dynamically created/updated by subsequent accesses. Each access to the shuffle index is assigned to a different delta version with exclusive write lock. Accessed blocks are downloaded from the delta version (if the delta version includes the node of interest) or from the shuffle index (otherwise), while shuffled blocks are written on the delta version. Periodically, delta versions are reconciled and applied to the shuffle index, to preserve the effects of the different shuffling operations performed by different users.

To efficiently support private accesses to r based on attributes different from K, in [17] the authors propose to complement the primary shuffle index with different *secondary shuffle indexes*, built on candidate keys that are expected to be often involved in query evaluation. A secondary index defined on attribute a is a shuffle index that stores, in association with value val for a, the values that attribute K has in tuple t (i.e., $t[K]$), such that $t[a] = val$. A search for the tuples in r with value val for attribute a proceeds then in two steps: *(1)* search for value val in the secondary index, retrieving the value val_K of attribute K in the tuples of interest; and then *(2)* search for value val_K in the primary index, retrieving the tuple of interest.

The distribution of the shuffle index over different servers, which are not aware one of each other, increases the protection offered to the confidentiality of users' accesses. According to the proposal in [16], in a distributed scenario cover searches, cached searches, and shuffling protection techniques can be complemented with *shadowing*. Shadowing guarantees that the observations by each server of accessed blocks make it believe to be the only server storing the whole data collection. In fact, each server observes the same number of blocks read (written, respectively) at each level of the tree.

5 Combining Access Control and Indexing Techniques

Access control enforcement and query evaluation over encrypted outsourced data has been widely studied, as testified by the different approaches illustrated in the previous sections of this chapter. However, the problem of combining them is still an open issue. The joint adoption of selective encryption (Sect. 2) and indexing techniques (Sect. 3) may permit authorized users to infer information they are not entitled to access. In fact, authorized users can infer the values that attributes have in tuples they should not be able to read, by exploiting their visibility over the index values for the tuples they are entitled to access. For instance, with reference to the encrypted relation in Fig. 11b and the access control policy regulating it in Fig. 3, user B can infer that $t_7[\texttt{ZIP}] = 22010$ even if $B \notin acl(t_7)$, because $t_7^k[I_z] = t_1^k[I_z]$ and B knows that $t_1[\texttt{ZIP}] = 22010$ since B belongs to $acl(t_1)$.

The problem of jointly adopting selective encryption and indexing techniques has recently been investigated, leading to the identification of different privacy risks that vary depending on the technique adopted for index definition (see Sect. 3) [13]. Before illustrating these risks, we summarize the knowledge of an authorized user u (i.e., a user who can access a subset of the tuples in r). Each authorized user knows: *(i)* index function ι used to define index I over attribute a (necessary for query evaluation); *(ii)* the plaintext tuples that the user can access; *(iii)* all the encrypted tuples in R^k (they are publicly available). For instance, consider the encrypted relation in Fig. 11b and the access control policy regulating it in Fig. 3. User A knows the index functions used by the data owner to define I_z, I_m, and I_i; all the plaintext tuples but t_3 and t_8; and the encrypted relation in Fig. 11b. The knowledge of user A

a

t	acl(t)
t_1	*ABC*
t_2	*ABC*
t_3	*BC*
t_4	*ABDE*
t_5	*ABCDE*
t_6	*ACDE*
t_7	*A*
t_8	*D*

b PATIENTS

	SSN	Name	ZIP	MarStatus	Illness
t_1	123456789	Ann	22010	single	gastritis
t_2	234567891	Barbara	24027	widow	neuralgia
t_3					
t_4	456789123	Daniel	20100	married	gastritis
t_5	567891234	Emma	21048	single	neuralgia
t_6	678912345	Fred	23013	married	hypertension
t_7	789123456	Gary	22010	widow	gastritis
t_8					

c PATIENTS^k

tid	enc	I_z	I_m	I_i
1	aD4%l	α	ζ	η
2	7Eoi)	β	κ	ξ
3	Gx?b3	α	θ	μ
4	dn4$z	γ	θ	η
5	2Cyl=	δ	ζ	ξ
6	Joi2s	ε	θ	ρ
7	(ceWm	α	κ	μ
8	w2!qk	β	κ	ρ

Fig. 14 Access control lists (**a**), knowledge of user *A* over relation PATIENTS (**b**), and over relation PATIENTS^k (**c**)

is summarized in Fig. 14, where gray cells denote plaintext values that user *A* is not authorized to read.

The inferences that an authorized user can draw on index *I* representing attribute *a* can be summarized as follows.

- *Direct index*. Since each plaintext value is associated with one index value and viceversa, if $t_i^k[I] = t_j^k[I]$ then also $t_i[a] = t_j[a]$ and viceversa. Hence, each user *u* can infer the plaintext value of attribute *a* for all those tuples in *r* that have the same value as a tuple that *u* is authorized to access. Consider, as an example, direct index I_z in relation PATIENTS^k in Fig. 14c. User *A* knows that $t_1[\texttt{ZIP}] = t_3[\texttt{ZIP}] = t_7[\texttt{ZIP}] = 22010$ even if she cannot read t_3, since all these tuples have the same value for index I_z.
- *Bucket-based index*. Since different plaintext values are mapped to the same index value, the information leakage illustrated for direct indexes is mitigated by the presence of collisions. Hence, if $t_i^k[I] = t_j^k[I]$ there is a certain (greater than zero) probability that also $t_i[a] = t_j[a]$, but there is no guarantee that this equality condition holds. Consider, as an example, index I_m in relation PATIENTS^k in Fig. 14c. Since the value for I_m is the same for t_2, t_7, and t_8, user *A* can infer that probably $t_2[\texttt{MarStatus}] = t_7[\texttt{MarStatus}] = t_8[\texttt{MarStatus}] =$ widow. We note however that plaintext values 'widow' and 'divorced' are represented by the same index value κ.
- *Flattened index*. Although less straightforward, the inference risk caused by flattened indexes is the same as illustrated for direct indexes. In fact, each index value represents one plaintext value only and then if $t_i^k[I] = t_j^k[I]$, also $t_i[a] = t_j[a]$. The viceversa is instead not true, that is, not all the occurrences of a value *val* are represented by the same index value. However, each authorized user knows the index function ι adopted by the data owner and can then compute $\iota(val)$, retrieving all the index values representing *val*. For instance, consider flattened index I_i in relation PATIENTS^k in Fig. 14c. Although user *A* is not authorized to read tuple t_3, she can infer that $t_3[\texttt{Illness}] =$ gastritis as t_3 and t_7 have the same value for

PATIENTS^k

tid	enc	I_z
1	aD4%l	α'_A, α_B, α_C
2	7Eoi)	β_A, β_B, β_C
3	Gx?b3	α'_B, α'_C
4	dn4$z	γ_A, γ_B, γ_D, γ_E
5	2Cyl=	δ_A, δ_B, δ_C, δ_D, δ_E
6	Joi2s	ε_A, ε_C, ε_D, ε_E
7	(ceWm	α_A
8	w2!qk	β_D

Fig. 15 An example of encrypted and indexed version of relation PATIENTS with index I_z over ZIP defined according to a salted user-dependent function

index I_i. Also, since she can compute ι(gastritis)= $\{\eta, \mu\}$, she can infer that also t_1 and t_4 have this value for attribute Illness.

From the observations above, it is easy to see that attribute values are exposed when the same index value appears in association with tuples characterized by different access control lists. Consider two tuples t_i and t_j in r such that $acl(t_i) \neq acl(t_j)$, and $t_i^k[I] = t_j^k[I]$. All the users in $acl(t_i)$ ($acl t_j$, respectively) can draw inferences on the value of $t_j[a]$ ($t_i[a]$, respectively). For instance, tuples t_1, t_3, and t_7 have the same value for attribute ZIP, but different *acl*s. This permits A to infer that t_3[ZIP] = 22010 even if she should not be able to read such a tuple. A first solution to limit such a leakage of information is based on the idea that the index value representing value $t[a] = l$ should not only depend on l but also on $acl(t)$. In [13] the authors present a solution that operates on direct indexes, which represent the worst-case scenario. This approach associates a different index function ι_u with each user u (depending on a piece of secret information shared between u and the data owner). Function ι_u is salted (i.e., a randomly chosen salt is applied) to avoid that tuples with the same plaintext value v but different *acl* are associated with the same index value $\iota_u(v)$ for user u, which could easily be exploited for inferences. For instance, consider direct index I_z in relation PATIENTS^k in Fig. 14c. Figure 15 illustrates relation PATIENTS^k, where index I_z has been defined using a different (salted) index function for each user in the system. For readability, in the figure we use a subscript to indicate the user to which the index value refers (e.g., α_A is a value computed by ι_A) and symbol $'$ denotes the salted version of index values (e.g., α'_A is the salted version of α_A).

While interesting, the proposal illustrated in [13] and mentioned above considers one specific indexing technique only. Even if it can be easily extended to operate with bucket-based and flattened indexing functions, it cannot be combined with the privacy-preserving indexing approaches described in Sect. 4. Furthermore, user-based indexing techniques are suitable for static scenarios, as dynamic observations of repeated accesses to the data can reveal to an observer which index values represent the same plaintext value. In fact, index values representing the same plaintext value

are often accessed together by authorized users. For instance, with reference to relation PATIENTS^k in Fig. 15b, every time user A needs to access all the tuples with ZIP = 22010, she will query the encrypted relation with the condition $I_z = \alpha_A$ OR $I_z = \alpha'_A$. The server can then easily conclude that α_A and α'_A represent the same plaintext value.

6 Conclusions

Public and private organizations are more and more resorting to cloud systems for outsourcing their own data centers. While bringing intuitive benefits in terms of economies of scale, moving to the cloud raises new privacy risks, since data are no more under the direct control of their owner. The research and development communities have dedicated many efforts in the design and development of novel techniques for protecting outsourced data and accesses to them. In this chapter, we surveyed recent approaches that, while protecting confidentiality of the data to the eyes of the storing server through encryption, enforce access control restrictions and efficiently evaluate queries over encrypted data, possibly without even revealing to the server the target of accesses. We also described the main issues arising when these techniques are adopted in combination, illustrating a preliminary approach for their solution.

Acknowledgements This chapter is based on joint work with Sushil Jajodia, Gerado Pelosi, and Stefano Paraboschi. This work was supported in part by the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH), and by Google, under the Google Research Award program.

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 TOCS 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. Bertino, E., Jajodia, S., Samarati, P.: Database security: Research and practice. Information Systems 20(7), 537–556 (November 1995)
5. Ceselli, A., Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Modeling and assessing inference exposure in encrypted databases. ACM TISSEC 8(1), 119–152 (February 2005)
6. Chang, Y., Mitzenmacher, M.:Privacy preserving keyword searches on remote encrypted data. In: Proc. of ACNS 2005. New York, NY, USA(June 2005)
7. Crampton, J., Martin, K., Wild, P.: On key assignment for hierarchical access control. In: Proc. of CSFW 2006. Venice, Italy (July 2006)

8. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: Proc. of ACM CCS 2006. Alexandria, VA, USA (October - November 2006)
9. 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)
10. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G.: Enforcing subscription-based authorization policies in cloud scenarios. In: Proc. of DBSec 2012. Paris, France (July 2012)
11. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G., Paraboschi, S., Samarati, P.: Enforcing dynamic write privileges in data outsourcing. Computers & Security 39, 47–63 (November 2013)
12. 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)
13. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Private data indexes for selective access to outsourced data. In: Proc. of WPES 2011. Chicago, IL, USA (October 2011)
14. 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)
15. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Supporting concurrency in private data outsourcing. In: Proc. of ESORICS 2011. Leuven, Belgium (September 2011)
16. 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)
17. 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)
18. De Cristofaro, E., Lu, Y., Tsudik, G.: Efficient techniques for privacy-preserving sharing of sensitive information. In: Proc. of TRUST 2011. Pittsburgh, PA, USA (June 2011)
19. De Santis, A., Ferrara, A., Masucci, B.: Cryptographic key assignment schemes for any access control policy. IPL 92(4), 199–205 (November 2004)
20. Ding, X., Yang, Y., Deng, R.: Database access pattern protection without full-shuffles. IEEE TIFS 6(1), 189–201 (March 2011)
21. 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)
22. Gamassi, M., Lazzaroni, M., Misino, M., Piuri, V., Sana, D., Scotti, F.: Quality assessment of biometric systems: a comprehensive perspective based on accuracy and performance measurement. IEEE TIM 54(4), 1489–1496 (August 2005)
23. Gamassi, M., Piuri, V., Sana, D., Scotti, F.: Robust fingerprint detection for access control. In: Proc. of RoboCare Workshop. Rome, Italy (May 2005)
24. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proc. of STOC 2009. Bethesda, MA, USA (May 2009)
25. Goh, E.J.: Secure indexes. Tech. Rep. 2003/216, Cryptology ePrint Archive (2003), http://eprint.iacr.org/
26. Goldreich, O., Ostrovsky, R.: Software protection and simulation on Oblivious RAMs. JACM 43(3), 431–473 (May 1996)
27. Goodrich, M., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Practical oblivious storage. In: Proc. of CODASPY 2012. San Antonio, TX, USA (February 2012)
28. 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)

29. 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)
30. 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)
31. 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)
32. Jhawar, R., Piuri, V.: Fault tolerance management in IaaS clouds. In: Proc. of ESTEL 2012. Rome, Italy (October 2012)
33. Jhawar, R., Piuri, V.: Fault tolerance and resilience in cloud computing environments. Computer and Information Security Handbook 2nd Edition Vacca J. (ed.), Morgan Kaufmann (2013)
34. Jhawar, R., Piuri, V., Samarati, P.: Supporting security requirements for resource management in cloud computing. In: Proc. of CSE 2012. Paphos, Cyprus (December 2012)
35. Lin, P., Candan, K.: Hiding traversal of tree structured data from untrusted data stores. In: Proc. of WOSIS 2004. Porto, Portugal (April 2004)
36. Lu, Y., Tsudik, G.: Privacy-preserving cloud database querying. JISIS 1(4), 5–25 (November 2011)
37. Pang, H., Zhang, J., Mouratidis, K.: Enhancing access privacy of range retrievals over $B+$-trees. IEEE TKDE 25(7), 1533–1547 (July 2013)
38. 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)
39. Samarati, P., De Capitani di Vimercati, S.: Data protection in outsourcing scenarios: Issues and directions. In: Proc. of ASIACCS 2010. Beijing, China (April 2010)
40. 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)
41. Sandhu, R.: Cryptographic implementation of a tree hierarchy for access control. IPL 27(2), 95–98 (February 1988)
42. 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)
43. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: ObliviStore: High performance oblivious cloud storage. In: Proc. of ACM CCS 2013. Berlin, Germany (November 2013)
44. Wan, Z., Liu, J., Deng, R.H.: HASBE: A hierarchical attribute-based solution for flexible and scalable access control in cloud computing. IEEE TIFS 7(2), 743–754 (April 2012)
45. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. IEEE TPDS 23(8), 1467–1479 (August 2012)
46. Wang, H., Lakshmanan, L.: Efficient secure query evaluation over encrypted XML databases. In: Proc. of VLDB 2006. Seoul, Korea (September 2006)
47. Waters, B.: Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In: Proc. of PKC 2011. Taormina, Italy (March 2011)
48. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: Proc. of ACM CCS 2012. Raleigh, NC, USA (October 2012)
49. 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)
50. Williams, P., Sion, R., Tomescu, A.: PrivateFS: A parallel oblivious file system. In: Proc. of ACM CCS 2012. Raleigh, NC, USA (October 2012)
51. 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)
52. 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)
53. 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)