Privacy of Outsourced Data

Sabrina De Capitani di Vimercati[1], Sara Foresti[1], Stefano Paraboschi[2], Pierangela Samarati[1]

1 University of Milan - 26013 Crema, Italy

{decapita,foresti,samarati}@dti.unimi.it

2 University of Bergamo - 24044 Dalmine, Italy

parabosc@unibg.it

Contents

1 Introduction

The amount of information held by organizations' databases is increasing very quickly. To respond to this demand, organizations can either add data storage and skilled administrative personnel (at a high rate), or, a solution becoming increasingly popular, delegate database management to an external service provider (*database outsourcing*). In database outsourcing, usually referred to as *Database As a Service* (DAS) the external service provider provides mechanisms for clients to access the outsourced databases. A major advantage of database outsourcing is related to the high costs of in-house versus outsourced hosting. Outsourcing provides significant cost savings and promises higher availability and more effective disaster protection than in-house operations. On the other hand, database outsourcing poses a major security problem, due to the fact that the external service provider, which is relied upon for ensuring high availability of the outsourced database (i.e., it is trustworthy), cannot always be trusted with the confidentiality of database content.

Besides well-known risks of confidentiality and privacy breaks, threats to outsourced data include improper use of database information: the server could extract, resell, or commercially use parts of a collection of data gathered and organized by the data owner, potentially harming the data owner's market for any product or service that incorporates that collection of information. Traditional database access control techniques cannot prevent the server itself from making unauthorized access to the data stored in the database. Alternatively, to protect against "honest but curious" servers, a protective layer of encryption can be wrapped around specific sensitive data, preventing outside attacks as well as infiltration from the server itself [1]. Data encryption, however, raises the problem of efficiently querying the outsourced database (now encrypted). Since confidentiality demands that data decryption must be possible only at the client-side, techniques have been proposed, enabling external servers to directly execute queries on encrypted data. Typically, these solutions consist mainly in adding a piece of information, called *index*, to the encrypted data. An index is computed based on the plaintext data and it preserves some of the original characteristics of the data.

Several approaches have been proposed for encrypting and indexing outsourced databases and for querying them. Also, these proposals assume all users to have complete access to the whole database. This assumption does not fit current scenarios where different users may need to see different portions of the data, that is, where selective access needs to be enforced. Adding a traditional authorization layer to the current outsourcing scenarios requires that when a client posed a query, both the query and its result have to be filtered by the data owner (who is in charge of enforcing the access control policy), a solution that however is not applicable in a real life scenario. More recent research has addressed the problem of enforcing selective access on the outsourced encrypted data by combining cryptography with authorizations, thus enforcing access control

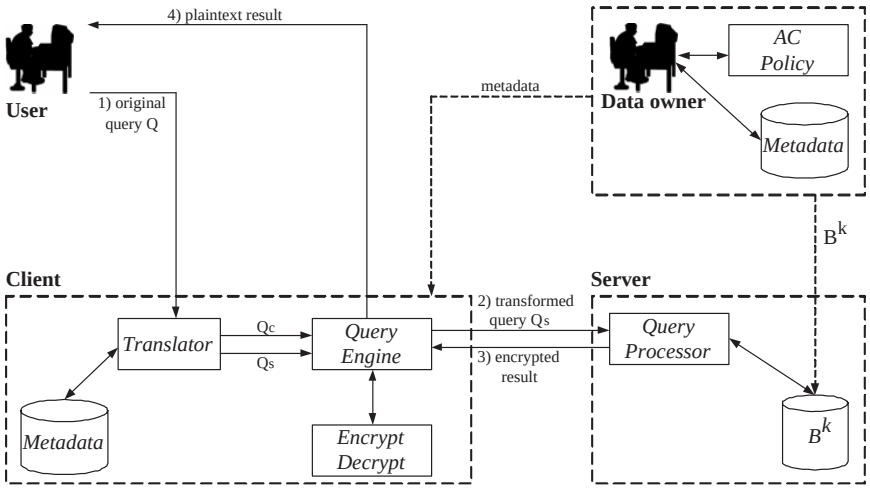


Figure 1: DAS Scenario

via *selective encryption*. Basically, the idea is to use different keys for encrypting different portions of the database. These keys are then distributed to users according to their access rights. The challenge is then to limit the amount of cryptographic information that needs to be stored and managed.

In this chapter, we survey the main proposals addressing the data access and security issues arising in the database outsorcing scenario. The remainder of the chapter is organized as follows. Section 2 gives an overview of the entities involved in the DAS scenario and of their typical interactions. Section 3 describes the main indexing methods proposed in the literature for supporting queries over encrypted data. Section 4 presents the main proposals for enforcing selective access on the outsourced data. Finally, Section 5 concludes the chapter.

2 Basic Scenario and Data Organization

We describe the entities involved in the DAS scenario, how data are organized in the outsourced database, and the interaction among the entities for query execution.

2.1 Parties Involved

There are four distinct entities interacting in the DAS scenario (Figure 1):

- a *data owner* (person or organization) produces and outsources resources to make them available for controlled external release;
- a *user* (human entity) presents requests (queries) to the system;

- a *client* front-end transforms the queries posed by users into equivalent queries operating on the encrypted data stored on the server;
- a *server* receives the encrypted data from one or more data owners and makes them available for distribution to clients.

Clients and data owners, when outsourcing data, are assumed to trust the server to faithfully maintain outsourced data. The server is then relied upon for the availability of outsourced data, so the data owner and clients can access data whenever requested. However, the server (which can be "honest but curious") is not trusted with the confidentiality of the actual database content, as outsourced data may contain sensitive information that the data owner wants to realease only to authorized users. Consequently, it is necessary to preserve the server from making unauthorized access to the database. To this purpose, the data owner encrypts her data with a key known only to trusted clients, and sends the encrypted database to the server for storage.

2.2 Data Organization

A database can be encrypted according to different strategies. In principle, both symmetric and asymmetric encryption can be used at different granularity levels. Symmetric encryption, being cheaper than asymmetric encryption, is usually adopted. The granularity level at which database encryption is performed can depend on the data that need to be accessed. Encryption can then be at the finer grain of [2, 3]:

- *table*: each table in the plaintext database is represented through a single encrypted value in the encrypted database. Consequently, tuples and attributes are indistinguishable in the released data, and cannot be specified in a query on the encrypted database;
- *attribute*: each column (attribute) in the plaintext table is represented by a single encrypted value in the encrypted table;
- *tuple*: each tuple in the plaintext table is represented by a single encrypted value in the encrypted table;
- *element*: each cell in the plaintext table is represented by a single encrypted value in the encrypted table.

Both table level and attribute level encryption imply the communication to the requesting client of the whole table involved in a query, as it is not possible to extract any subset of the tuples in the encrypted representation of the table. On the other hand, encrypting at element level would require an excessive

Employee

Emp-Id	Name	YoB	Dept	Salary
P01	Ann	1980	Production	10
R01	Bob	1975	R&D	15
F01	Bob	1985	Financial	10
P02	Carol	1980	Production	20
F02	Ann	1980	Financial	15
R02	David	1978	R&D	15

(a)

Employee^k

Counter	Etuple	I_1	I_2	I_3	I_4	I_5
1	ite6*+8wc	π	α	γ	ε	λ
2	8(nfeua4!=	ϕ	β	δ	θ	λ
3	Q73gnew321*/	ϕ	β	γ	μ	λ
4	-1vs9e892s	π	α	γ	ε	ρ
5	e32rfs4+@	π	α	γ	μ	λ
6	r43arg*5[)	ϕ	β	δ	θ	λ

(b)

Figure 2: An example of plaintext (a) and encrypted (b) table

workload for data owners and clients in encrypting/decrypting data. For balancing client workload and query execution efficiency, most proposals assume that the database is encrypted at tuple level.

While database encryption provides an adequate level of protection for data, it makes impossible for the server to directly execute the users' queries on the encrypted database. Upon receiving a query, the server can only send to the requestor the encrypted tables involved in the query; the client can then decrypt such tables and execute the query on them. To allow the server to select a set of tuples to be returned in response to a query, a set of indexes can be associated with the encrypted table. In this case, the server stores an encrypted table with an index for each attribute on which conditions may need to be evaluated. For simplicity, we assume an index for each attribute in each table of the database. Different kinds of indexes can be defined, depending on the clauses and conditions that need to be remotely evaluated for the different attributes. Given a plaintext database $\mathcal{B}$, each table $\mathtt{r_i}$ over schema $\mathtt{R_i}(\mathtt{A_{i1}}, \mathtt{A_{i2}}, \ldots, \mathtt{A_{in}})$ in $\mathcal{B}$ is mapped onto a table $\mathtt{r_i^k}$ over schema $\mathtt{R_i^k}(\underline{\mathtt{Counter}}, \mathtt{Etuple}, \mathtt{I_1}, \mathtt{I_2}, \ldots, \mathtt{I_n})$ in the corresponding encrypted database $\mathcal{B}^k$. Here, `Counter` is a numerical attribute added as primary key of the encrypted table; `Etuple` is the attribute containing the encrypted tuple whose value is obtained applying an encryption function E_k to the plaintext tuple, where k is the secret key; and $\mathtt{I_j}$ is the index associated with the $\mathtt{j}$-th attribute in $\mathtt{R_i}$. While we assume encrypted tuples and indexes to be in the same relation, we note that indexes can be stored in a separate table [4].

To illustrate, consider table `Employee` in Figure 2(a). The corresponding encrypted table is shown in Figure 2(b), where index values are conventionally represented with Greek letters. The encrypted table has exactly the same number of tuples as the original table. For the sake of readability, the tuples in the encrypted table are listed in the order with which they appear in the corresponding plaintext table. The same happens for the order of indexes, which are listed in the same order as the plaintext attributes to which they refer.

2.3 Interactions

The introduction of indexes allows to partially evaluate any query Q at the server-side, provided it is previously translated in an equivalent query operating on the encrypted database. Figure 1 summarizes the most important steps necessary for the evaluation of a query submitted by a user.

1. The user submits her query Q referring to the schema of the plaintext database $\mathcal{B}$, and passes it to the client front-end. The user needs not be aware that data have been outsourced to a third party.

2. The client maps the user's query onto: *1)* an equivalent query Q_s, working on the encrypted tables through indexes, and *2)* an additional query Q_c working on the results of Q_s. Query Q_s is then passed on to the remote server. Note that the client is the unique entity in the system that knows the structure of both $\mathcal{B}$ and $\mathcal{B}^k$ and that can translate the queries the user may submit.

3. The remote server executes the received query Q_s on the encrypted database and returns the result (i.e., a set of encrypted tuples) to the client.

4. The client decrypts the tuples received and eventually discards spurious tuples (i.e., tuples that do not satisfy the query submitted by the user). These spurious tuples are removed by executing query Q_c. The final plaintext result is then returned to the user.

Since a client may have a limited storage and computation capacity, one of the primary goals of the query execution process is to minimize the workload at the client side, while maximizing the operations that can be computed at the server side [2, 3, 5, 6].

Iyer et al. [2, 3] present a solution for minimizing the client workload that is based on a graphical representation of queries as trees. The tree representing a query is split in two parts: the lower part includes all the operations that can be executed by the server, while the upper part contains all the operations that cannot be delegated to the server and therefore needs to be executed by the client. In particular, since a query can be represented with different, but equivalent, trees by simply pushing down selections and postponing projections, the basic idea of the proposed solution is to determine an equivalent tree representation of the query, where the operations that only the client can execute are in the highest levels of the tree. For instance, if there are two ANDed conditions in the query and only one can be evaluated on the server-side, the selection operation is split in such a way that one condition is evaluated server-side and the other client-side.

Hacigümüs et al. [5] show a method for splitting the query Q_s to be executed on the encrypted data into two sub-queries, Q_{s1} and Q_{s2}, where Q_{s1} returns only tuples that will belong to the final result, and query Q_{s2} may contain also spurious tuples. This distinction allows the execution of Q_c over the result of Q_{s2}

only, while tuples returned by Q_{s1} can be immediately decrypted. To further reduce the client's workload, Damiani et al. [6] propose an architecture that minimizes storage at the client and introduce the idea of selective decryption of Q_s. With selective decryption, the client decrypts the portion of the tuples needed for evaluating Q_c, while complete decryption is executed only for tuples that belong to the final result and that will be returned to the final user. The approach is based on a block-cipher encryption algorithm, operating at tuple level, that allows the detection of the blocks containing the attributes necessary to evaluate the conditions in Q_c, which are the only ones that need decryption.

It is important to note that the process of transforming Q in Q_s and Q_c greatly depends both on the indexing method adopted and on the kind of query Q. There are operations that need to be executed by the client, since the indexing method adopted does not support the specific operations (e.g., range queries are not supported by all types of indexes) and the server is not allowed to decrypt data. Also, there are operations that the server could execute over the index, but that require a pre-computation that only the client can perform and therefore must be postponed in Q_c (e.g., the evaluation of a condition in the HAVING clause, which needs a grouping over an attribute whose corresponding index has been created by using a method that does not support the GROUP BY clause).

3 Querying Encrypted Data

When designing a solution for querying encrypted data, one of the most important goals is to minimize the computation at the client-side and to reduce communication overhead. The server therefore should be responsible for the majority of the work. Different index approaches allow the execution of different types of queries at server-side.

We now describe in more detail the methods initially proposed to efficiently execute simple queries at the server side, and we give an overview of more recent methods that improve the server's ability to query encrypted data.

3.1 Bucket-Based Index

Hacigümüs et al. [7] propose the first method to query encrypted data, which is based on the definition of a number of *buckets* on the attribute domain.

Let $\mathtt{r_i}$ be a plaintext relation over schema $\mathtt{R_i}(\mathtt{A_{i1}}, \mathtt{A_{i2}}, \ldots, \mathtt{A_{in}})$ and $\mathtt{r_i^k}$ be the corresponding encrypted relation over schema $\mathtt{R_i^k}(\underline{\mathtt{Counter}}, \mathtt{Etuple})$.

Considering an arbitrary plaintext attribute $\mathtt{A_{ij}}$ in $\mathtt{R_i}$, with domain $\mathtt{D_{ij}}$, bucket-based indexing methods partition $\mathtt{D_{ij}}$ in a number of non-overlapping subsets of values, called *buckets*, containing contiguous values.

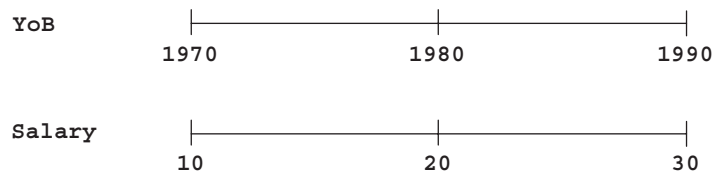


Figure 3: An example of bucketization

This process, called *bucketization*, usually generates buckets that are all of the same size.
Each bucket is then associated with a unique value and the set of these values is the domain for index $\mathtt{I_j}$ associated with $\mathtt{A_{ij}}$. Given a plaintext tuple $\mathtt{t}$ in $\mathtt{r_i}$, the value of attribute $\mathtt{A_{ij}}$ for $\mathtt{t}$ belongs to a bucket. The corresponding index value is then the unique value associated with the bucket to which the plaintext value $\mathtt{t}[\mathtt{A_{ij}}]$ belongs. It is important to note that, for better preserving data secrecy, the domain of index $\mathtt{I_j}$ may not follow the same order as the one of the plaintext attribute $\mathtt{A_{ij}}$. Attributes $\mathtt{I}_3$ and $\mathtt{I}_5$ in Figure 2(b) are the indexes obtained by applying the bucketization method described in Figure 3 to attributes `YoB` and `Salary` in Figure 2(a).

Bucket-based indexing methods allow the server-side evaluation of equality conditions appearing in the WHERE clause, as these conditions can be mapped into equivalent conditions operating on indexes. Given a plaintext condition of the form $\mathtt{A_{ij}}=v$, where v is a constant value, the corresponding condition operating on index $\mathtt{I_j}$ is $\mathtt{I_j}=\beta$, where β is the value associated with the bucket containing v. As an example, with reference to Figure 3, condition `YoB`=1985 is transformed into $\mathtt{I}_3=\gamma$. Also, equality conditions involving attributes defined on the same domain can be evaluated by the server, provided that attributes are indexed using the same bucketization. In particular, a plaintext condition of the form $\mathtt{A_{ij}}=\mathtt{A_{ik}}$ is translated into condition $\mathtt{I_j}=\mathtt{I_k}$ operating on indexes.
Bucket-based methods do not easily support range queries. Since the index domain does not necessarily preserve the plaintext domain ordering, a range condition of the form $\mathtt{A_{ij}} \geq v$, where v is a constant value, must be mapped into a series of equality conditions operating on index $\mathtt{I_j}$ of the form $\mathtt{I_j}=\beta_1$ OR $\mathtt{I_j}=\beta_2$ OR ... OR $\mathtt{I_j}=\beta_k$, where $\beta_1, \ldots, \beta_k$ are the values associated with buckets that correspond to plaintext values greater than or equal to v. As an example, with reference to Figure 3, condition `YoB`>1977 must be translated into $\mathtt{I}_3=\gamma$ OR $\mathtt{I}_3=\delta$, as both values represent years greater than 1977.

Since the same index value is associated with more than one plaintext value, bucket-based indexing usually produces spurious tuples that need to be filtered out by the client front-end. Spurious tuples are tuples that satisfy the condition over the indexes, but that do not satisfy the original plaintext condition. For instance, with reference to the tables in Figure 2, query "SELECT * FROM `Employee` WHERE `YoB`=1985"

is translated into "SELECT Etuple FROM $\texttt{Employee}^{\texttt{k}}$ WHERE $\texttt{I}_3{=}\gamma$". The result of the query executed by the server contains tuples 1, 3, 4, and 5; however, only tuple 3 satisfies the original condition as written by the user. Tuples 1, 4, and 5 are spurious and must be discarded by the client.

Hore et al. [8] propose an improvement to bucket-based index methods by introducing an efficient way for partitioning the domain of attributes. Given an attribute and a query profile of it, the authors present a method for building an efficient index, which tries to minimize the number of spurious tuples in the result of range and equality queries.

One of the main disadvantages of bucket-based indexing methods is that they expose data to inference attacks (see Section 3.5).

3.2 Hash-Based Index

Hash-based index methods are similar to bucket-based methods and are based on the concept of *one-way hash function* [4].

Let $\texttt{r}_\texttt{i}$ be a plaintext relation over schema $\texttt{R}_\texttt{i}(\texttt{A}_{\texttt{i1}}, \texttt{A}_{\texttt{i2}}, \ldots, \texttt{A}_{\texttt{in}})$ and $\texttt{r}_\texttt{i}^\texttt{k}$ be the corresponding encrypted relation over schema $\texttt{R}_\texttt{i}^\texttt{k}(\underline{\texttt{Counter}}, \texttt{Etuple})$. For each attribute $\texttt{A}_{\texttt{ij}}$ in $\texttt{R}_\texttt{i}$ to be indexed, a secure one-way hash function $h : \texttt{D}_{\texttt{ij}} \rightarrow \texttt{B}_{\texttt{ij}}$ is defined, where $\texttt{D}_{\texttt{ij}}$ is the domain of $\texttt{A}_{\texttt{ij}}$ and $\texttt{B}_{\texttt{ij}}$ is the domain of index $\texttt{I}_\texttt{j}$ associated with $\texttt{A}_{\texttt{ij}}$.

Given a plaintext tuple $\texttt{t}$ in $\texttt{r}_\texttt{i}$, the index value corresponding to attribute $\texttt{A}_{\texttt{ij}}$ for $\texttt{t}$ is computed by applying function h to the plaintext value $\texttt{t}[\texttt{A}_{\texttt{ij}}]$.

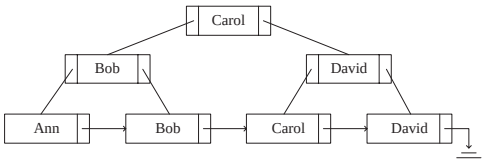
An important property of any secure hash function h is its *determinism*; formally, $\forall x, y \in \texttt{D}_{\texttt{ij}} : \; x = y \;\Rightarrow\; h(x) = h(y)$. Another interesting property of secure hash functions is that the codomain of h is smaller than its domain, so there is the possibility of *collisions*; a collision happens when given two values $x, y \in \texttt{D}_{\texttt{ij}}$ with $x \neq y$, we have that $h(x) = h(y)$. A further property is that h must produce a strong mixing, that is, given two distinct but near values x, y ($\mid x - y \mid < \epsilon$) chosen randomly in $\texttt{D}_{\texttt{ij}}$, the discrete probability distribution of the difference $h(x) - h(y)$ is uniform (the results of the hash function can be arbitrarily different, even for very similar input values). A consequence of strong mixing is that the hash function does not preserve the domain order of the attribute on which it is applied. As an example, consider the relations in Figure 2. Here the indexes corresponding to attributes Emp-Id, Name, and Dept in relation Employee are computed by applying a hash-based method. The values of attribute Name have been mapped onto two distinct values, namely α and β; the values of attribute Emp-Id have been mapped onto two distinct values, namely π and ϕ; and the values of attribute Dept have been mapped onto three distinct values, namely ε, θ, and μ. Like for bucket-based methods, hash-based methods allow an efficient evaluation of

equality conditions of the form $\texttt{A}_{\texttt{ij}}$=v, where v is a constant value. Each condition $\texttt{A}_{\texttt{ij}}$=v is transformed into a condition $\texttt{I}_{\texttt{j}}$=$h(v)$, where $\texttt{I}_{\texttt{j}}$ is the index corresponding to $\texttt{A}_{\texttt{ij}}$ in the encrypted table. For instance, condition `Name`="Alice" is transformed into $\texttt{I}_2$=α.

Also, equality conditions involving attributes defined on the same domain can be evaluated by the server, provided that these attributes are indexed using the same hash function. The main drawback of hash-based methods is that they do not support range queries, for which a solution similar to the one adopted for bucket-based methods is not viable: colliding values in general are not contiguous in the plaintext domain. Index collisions produce spurious tuples in the result. A collision-free hash function guarantees absence of spurious tuples, but may expose data to inference (see Section 3.5). For instance, assuming that the hash function adopted for attribute `Dept` is collision-free, condition `Dept`="Financial" is translated into $\texttt{I}_4$=μ, that will return only the tuples (in our example, tuples 3 and 5) that belong to the result set of the query that contains the corresponding plaintext condition.

3.3 B+ Trees Index

Both bucket-based and hash-based indexing methods do not easily support range queries, since both these solutions are not order preserving. However, there is frequently the need for range queries. Damiani et al. [4] propose an indexing method that, while granting data privacy, preserves the order of plaintext data. This indexing method exploits the traditional B+ tree data structure used by relational DBMSs for physically indexing data. A B+ tree with fan out n is a tree where every vertex can store up to $n-1$ search key values and n pointers and, except for the root and leaf vertices, has at least $\lceil n/2 \rceil$ children. Given an internal vertex storing p key values $k_1, \ldots, k_p$ with $p \leq n-1$, each k_i is followed by a pointer a_i and k_1 is preceded by a pointer a_0. Pointer a_0 points to the subtree that contains keys with values lower than k_1, a_p points to the subtree that contains keys with values greater than or equal to k_p, and each a_i points to the subtree that contains keys with values included in the interval $[k_i, k_{i+1})$. Internal vertices do not directly refer to tuples in the database, but just point to other vertices in the structure; on the contrary, leaf vertices do not contain pointers, but directly refer to the tuples in the database having a specific value for the indexed attribute. Leaf vertices are linked in a chain that allows the efficient execution of range queries. As an example, Figure 4(a) represents the B+ tree index built for attribute `Name` of table `Employee` in Figure 2(a). To access a tuple with key value k, value k is first searched in the root vertex of the B+ tree. The tree is then traversed by using the following scheme: if $k < k_1$, pointer a_0 is chosen; if $k \geq k_p$, pointer a_p is chosen, otherwise if $k_i \leq k < k_{i+1}$, pointer a_i is chosen. The process continues until a leaf vertex has been examined. If k is not found in any leaf vertex, then the table does not contain any tuple having, for the



(a)

Id	VertexContent
1	2, Carol, 3
2	4, Bob, 5
3	6, David, 7
4	Ann, 5, 1, 5
5	Bob, 6, 2, 3
6	Carol, 7, 4
7	David, NIL, 6

(b)

Id	C
1	gtem945/*c
2	8dq59wq*d'
3	ue63/)¡w
4	8/*5sym,p
5	mw39wio[
6	=wco21!ps
7	oieb5(p8*

(c)

Figure 4: An example of B+ tree indexing structure

indexed attribute, value k.

A B+ tree index can be usefully adopted for each attribute $\mathtt{A_{ij}}$ in schema $\mathtt{R_i}$ and defined over a partially ordered domain. The index is built by the client over the plaintext values of the attribute, and then stored on the remote server, together with the encrypted database. To this purpose, the B+ tree structure is translated into a specific table with just two attributes: the vertex identifier and the vertex content. The table has a row for each vertex in the tree and pointers are represented through cross references from the vertex content to other vertex identifiers in the table. For instance, the B+ tree structure depicted in Figure 4(a) is represented in the encrypted database by the relation in Figure 4(b).

Since the relation representing the B+ tree contains sensitive information (i.e., the plaintext values of the attribute on which the B+ tree is built) this relation has to be protected by encrypting its content. To this purpose, encryption is also applied at the level of vertex, to protect the order relationship between plaintext and index values. The corresponding encrypted table has therefore two attributes: attribute `Id` that, as before, is the identifier of the vertex, and attribute `C` that is the encrypted vertex. Figure 4(c) illustrates the encrypted B+ tree table that corresponds to the plaintext B+ tree table in Figure 4(b).

The B+ tree based indexing method allows the evaluation of both equality and range conditions appearing in the WHERE clause. Moreover, being order preserving, it also allows the evaluation of ORDER BY and GROUP BY clauses, and of most of the aggregate operators, directly on the encrypted database.

Given the plaintext condition $\mathtt{A_{ij}} > v$, where v is a constant value, the client needs to traverse the B+ tree stored on the server to find out the leaf vertex representing v. To this purpose, the client queries the B+ tree table to retrieve the root, which is the tuple with *Id* equal to 1. It then decrypts it, evaluates its content, and according to the search process above-mentioned queries again the remote server to retrieve the next vertex that has to be checked. The search process continues until a leaf vertex containing v is found (if any). The client then follows the chain of leaf vertices starting from the retrieved leaf. As an example, consider the B+ tree in Figure 4(a) defined for attribute `Name`. A query asking for tuples where the value of attribute

`Name` follows Bob in the lexicographic order is evaluated as follows. First, the root is retrieved and evaluated: since Bob precedes Carol, the first pointer is chosen and vertex 2 evaluated. Since Bob is then equal to the value in the vertex, the second pointer is chosen and vertex 5 evaluated. Vertex 5 is a leaf, and all tuples in vertices 5, 6, and 7 are returned to the final user.

It is important to note that B+ tree indexes do not produce spurious tuples when executing a query, but the evaluation of conditions is much more expensive for the client with respect to bucket and hash-based methods. For this reason, it may be advisable to combine the B+ tree method with either hash-based or bucket-based indexing, and use B+ tree only for evaluating conditions based on intervals. Compared with traditional B+ tree structures used in DBMSs, the vertices do not have to be of the same size as a disk block; a cost model can then be used to optimize the number of children of a vertex, potentially producing vertices with a large number of children and trees with limited depth. Finally, we note that since the B+ tree content is encrypted, the method is secure against inference attacks.

3.4 Other Approaches

In addition to the three main indexing methods previously presented, many other solutions have been proposed to support queries on encrypted data. These methods try to better support SQL clauses or to reduce the amount of spurious tuples in the result produced by the remote server.

Wang et al. [9, 10] propose a new indexing method, specific for attributes whose domain is a set of characters, which adapts the hash-based indexing methods to permit direct evaluation of LIKE conditions. The index value associated with any string s, composed of n characters $c_1c_2 \ldots c_n$, is obtained by applying a secure hash function to each couple of subsequent characters in s. Specifically, given a string $s = c_1c_2 \ldots c_n = s_1s_2 \ldots s_{n/2}$, where $s_i = c_{2i}c_{2i+1}$, the corresponding index is $i = h(s_1)h(s_2) \ldots h(s_{n/2})$.

Hacigümüs et al. [5] study a method to remotely support aggregation operators, such as COUNT, SUM, AVG, MIN, and MAX. The method is based on the concept of *privacy homomorphism* [11, 12], which exploits properties of modular algebra to allow the execution over index values of sum, subtraction, and product, while not preserving domain ordering. Evdokimov et al. [13] formally analyze the security of the method based on privacy homomorphism with respect to the degree of confidentiality assigned to the remote server. Specifically, a definition of intrinsic security is given for encrypted databases, and it is proved that almost all indexing methods are not intrinsically secure; in particular, methods that do not cause spurious tuples to belong to the result of a query inevitably are exposed to attacks coming from a malicious third party or from the service provider.

The *Partition Plaintext and Ciphertext* (PPC) is a new model for storing server-side outsourced data [3].

This model proposes outsourcing of both plaintext and encrypted information, which need to be stored on the remote server. In this model, only sensitive attributes are encrypted and indexed, while the other attributes are released in plaintext form. The authors propose an efficient architecture for the DBMS to store together, and specifically in the same page of memory, both plaintext and encrypted data.

To support equality and range queries over encrypted data without adopting B+ tree structures, Agrawal et al. [14] present an *Order Preserving Encryption Schema* (OPES). An OPES function has the advantage of flattening the frequency spectrum of index values, thanks to the introduction of new buckets when needed. It is important to note here that queries executed over this kind of indexes do not return spurious tuples. Also, OPES provides data secrecy only if the intruder does not know the plaintext database or the domain of original attributes.

Aggarwal et al. [15] discuss a new solution for querying remotely stored data, while preserving their privacy. The authors assume that some security constraints are defined on outsourced data, specifying which sets of attributes cannot be released together and which attributes cannot appear in plaintext. To guarantee constraint satisfaction, the authors propose to vertically fragment the universal relation R, decomposing the database into two fragments that are then stored on two different servers. The method is based on the assumption that the two servers do not exchange information and that all the constraints can be satisfied by encrypting just a few attributes in each fragment.

Different working groups [16, 17, 18, 19, 20] introduce other approaches for searching keywords in encrypted documents. These methods are based on the definition of a *secure index data structure*. The secure index data structure allows the server to retrieve all documents containing a particular keyword without the need to know any other information. This is possible because a trapdoor is introduced when encrypting data, and such a trapdoor is then exploited by the client when querying data. Other similar proposals are based on *Identity Based Encryption* technique for the definition of secure indexing methods. Boneh and Franklin [21] present an encryption method allowing searches over ciphertext data, while not revealing anything about the original data. This method is shown to be secure through rigorous proofs. Although these methods for searching keywords over encrypted data have been originally proposed for searching over audit logs or email repositories, they are also well suited for indexing data in the outsourced database scenario.

To summarize, Figure 5 shows, for each indexing method discussed, what type of query is supported. Here, an hyphen means that the query is not supported, a black circle means that the query is supported, and a white circle means that the query is partially supported.

Index	Query		
	Equality	**Range**	**Aggregation**
Bucket-based [7]	•	○	–
Hash-based [4]	•	–	○
B+ Tree [4]	•	•	•
Character oriented [9, 10]	•	○	–
Privacy homomorphism [5]	•	–	•
PPC [3]	•	•	•
OPES [14]	•	•	○
Secure index data structures [16, 17, 18, 19, 20]	•	○	–
Fragmentation based [15]	•	•	•

• fully supported; ○ partially supported; – not supported

Figure 5: Indexing methods supporting queries

3.5 Evaluation of Inference Exposure

Given a plaintext relation $\mathtt{r}$ over schema $\mathtt{R}(\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n)$, it is necessary to decide which attributes need to be indexed, and how the corresponding indexes can be defined. In particular, when defining the indexing method for an attribute, it is important to consider two conflicting requirements: on one side, the indexing information should be related with the data well enough to provide for an effective query execution mechanism; on the other side, the relationship between indexes and data should not open the door to *inference and linking attacks* that can compromise the protection granted by encryption. Different indexing methods can provide a different trade-off between query execution efficiency and data protection from inference. It is therefore necessary to define a measure of the risk of exposure due to the publication of indexes on the remote server.

Although many techniques to support various types of queries in the DAS scenario have been developed, a deep analysis of the level of protection provided by all these methods against inference and linking attacks is missing. In particular, only inference exposure of a few indexing methods has been evaluated [4, 8, 22, 23].

Hore et al. [8] analyze the security issues related to the use of bucket-based indexing methods. The authors consider data exposure problems in two situations: *1)* the release of a single attribute, and *2)* the publication of all the indexes associated with a relation. To measure the protection degree granted to the original data by the specific indexing method, the authors propose to exploit two different measures. The first measure is the *variance* of the distribution of values within a bucket b. The second measure is the *entropy* of the distribution of values within a bucket b. The higher is the variance, the higher is the protection level granted to the data. Therefore, the data owner should maximize, for each bucket in the relation, the corresponding variance. Analogously, the higher is the entropy of a bucket, the higher is the protection level of the relation. The optimization problem that the data owner has to solve, while planning the bucketization process on a

table, is the *maximization of minimum variance and minimum entropy*, while maximizing query efficiency. Since such an optimization problem is *NP-hard*, Hore et al. [8] propose an approximation method, which fixes a maximum allowed performance degradation. The objective of the algorithm is then to maximize both minimum variance and entropy, while guaranteeing performances not to follow under an imposed constraint. To the aim of taking into consideration also the risk of exposure due to association, Hore et al. [8] propose to adopt, as a measure of the privacy granted by indexes when posing a multi-attribute range query, the well known *k-anonymity* concept [24].

Damiani et al. [4, 22, 23] evaluate the exposure to inference due to the adoption of hash-based indexing methods. Inference exposure is measured by taking into account the prior knowledge of the attacker that introduces two different scenarios. In the first scenario, called Freq+DB^k, the attacker is supposed to know, in addition to the encrypted database (DB^k), the domains of the plaintext attributes and the distribution of plaintext values (Freq) in the original database. In the second scenario, called DB+DB^k, the attacker is supposed to know both the encrypted (DB^k) and the plaintext database (DB). In both scenarios, the exposure measure is computed as the probability for the attacker to correctly map index values onto plaintext attribute values. The authors show that to guarantee a higher degree of protection against inference, it is convenient to use a hash-based method that generates collisions. In case of a hash-based method where the collision factor is equal to 1, meaning that there are no collisions, the inference exposure depends on the number of attributes used for indexing. In the DB+DB^k scenario, the exposure grows as the number of attributes used for indexing grows. In the Freq+DB^k scenario, the attacker can discover the correspondences between plaintext and indexing values by comparing their occurrence profiles. Intuitively, the exposure grows as the number of attributes with a different occurrence profile grows. For instance, considering relation `Employee` in Figure 2(a), we can notice that both `Salary` and the corresponding index $\mathtt{I_5}$ have a unique value, that is, 20 and ρ, respectively. We can therefore conclude that the index value corresponding to 20 is ρ, and that no other salary value is mapped into ρ as well.

Damiani et al. [23] extend the inference exposure measures presented in [4, 22] to produce an inference measure that can be associated with the whole table instead of with single attributes. Specifically, the authors propose two methods for aggregating the exposure risk measures computed at attribute level. The first method exploits the weighted mean operator and weights each attribute $\mathtt{A_i}$ proportionally with the risk connected with the disclosure of the values of $\mathtt{A_i}$. The second one exploits the *OWA* (Ordered Weighted Averaging) operator, which allows the assignment of different importance values to different sets of attributes, depending on the degree of protection guaranteed by the indexing method adopted for the specific subset of attributes.

4 Security Issues

The emerging DAS scenario introduces also numerous research challenges related to data security. We now describe the main proposals that aim at ensuring the confidentiality of the outsourced data.

4.1 Access Control Enforcement

All the existing proposals for designing and querying encrypted/indexing outsourced databases assume that the client has complete access to the query result. However, this assumption does not fit real world scenarios, where different users may have different access privileges. A trivial solution for implementing selective access in the DAS scenario consists in explicitly defining authorizations at the data owner site. The main drawback of this method is that the server cannot directly send the result of a query to the client because the data owner first has to remove all the tuples that the final user cannot access (this task cannot be delegated to the remote server, which may not be allowed to know the access control policy defined by the data owner). Such an approach however puts much of the work on the data owner introducing a bottleneck for computation and communication.

A promising direction consists in selectively encrypting data so that users (or groups thereof) can decrypt only the data they are authorized to access. Intuitively, selective encryption means that data are encrypted by using different keys and that users can decrypt only data for which they know the corresponding encryption key. Although it is usually advisable to leave authorization-based access control and cryptographic protections separate, as encryption is traditionally considered a mechanism and should not be adopted in model definition, such a combination proves successful in the DAS scenario [30]. In particular, since neither the data owner nor the remote server can enforce the access control policy, for either security or efficiency reasons, the data themselves need to implement selective access. We now describe the proposals supporting selective access in more details.

4.1.1 Selective Encryption

Given a system composed of a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources, the data owner may want to define and enforce a policy, stating which user $u_i \in \mathcal{U}$ is allowed to access which resource $r_j \in \mathcal{R}$ in the outsourced database. In the DAS scenario, a resource may be a table, an attribute, a tuple, or even a cell, depending on the granularity at which the data owner wishes to define her policy. Since existing solutions do not depend on the granularity level to which the access control policy is defined [30], in the remainder of this section, we will continue to use the generic term resource to generically indicate any database element on which authorizations can be specified.

	$\mathbf{r_1}$	$\mathbf{r_2}$	$\mathbf{r_3}$	$\mathbf{r_4}$
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	0	1	1	1

Figure 6: An example of access matrix

The set of authorizations defined by the data owner are represented through a traditional *access matrix* $\mathcal{A}$ having a row for each user in $\mathcal{U}$ and a column for each resource in $\mathcal{R}$. Since only read privileges are considered (the enforcement of write privileges is still an open issue), each cell $\mathcal{A}[u_i,r_j]$ may assume two values: 1, if u_i is allowed to access r_j; 0, otherwise. Given an access matrix $\mathcal{A}$ over sets $\mathcal{U}$ and $\mathcal{R}$, $acl(r_j)$ denotes the *access control list* of resource r_j (i.e., the set of users that can access r_j), and $cap(u_i)$ denotes the *capability list* of user u_i (i.e., the set of resources that u_i can access). For instance, Figure 6 represents an access matrix for a system with four users (A, B, C, and D), and four resources (r_1, r_2, r_3, and r_4). Here, for example, $acl(r_1)$=$\{B,C\}$ and $cap(B)$=$\{r_1,r_3\}$.

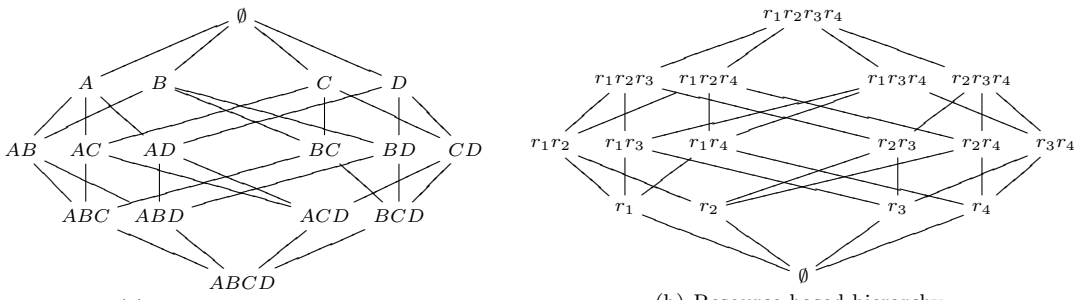
The naive solution for enforcing access control through selective encryption consists in using a different key for each resource in the system, and in communicating to each user the set of keys associated with the resources she can access. This solution correctly enforces the policy, but it is very expensive since each user needs to keep a number of keys that depends on her privileges. That is, users having many privileges and, probably, often accessing the system, will have a greater number of keys than users having a few privileges and, probably, accessing only rarely the system. To reduce the number of keys a user has to manage, the authors propose to use a *key derivation method*. A key derivation method is basically a function that, given a key and a piece of publicly available information, allows the computation of another key. The basic idea is that each user is given a small number of keys, from which she can derive all the keys needed to access the resources she is authorized to access.

To the aim of using a key derivation method, it is necessary to define which keys can be derived from another key and how. Key derivation methods proposed in the literature are based on the definition of a *key derivation hierarchy*. Given a set of keys $\mathcal{K}$ in the system and a partial order relation $\preceq$ defined on it, the corresponding key derivation hierarchy is usually represented as a pair ($\mathcal{K}$,$\preceq$), where $\forall k_i, k_j \in \mathcal{K}$, $k_j \preceq k_i$ iff k_j is derivable from k_i. Any key derivation hierarchy can be graphically represented through a directed graph, having a vertex for each key in $\mathcal{K}$, and a path from k_i to k_j only if k_j can be derived from k_i. Depending on the partial order relation defined on $\mathcal{K}$, the key derivation hierarchy can be: a *chain* (i.e., $\preceq$ defines a total order relation); a *tree*; or a *directed acyclic graph* (DAG). The different key derivation methods can be classified on the basis of the kind of hierarchy they are able to support, as follows.

- The hierarchy is a *chain of vertices* [31]. Key k_j of a vertex is computed on the basis of key k_i of its (unique) parent (i.e., $k_j = f(k_i)$) and no public information is needed.
- The hierarchy is a *tree* [31, 32, 33]. Key k_j of a vertex is computed on the basis of key k_i of its (unique) parent and on the publicly available label l_j associated with k_j (i.e., $k_j = f(k_i, l_j)$).
- The hierarchy is a *DAG* [34, 35, 36, 37, 38, 39, 40, 41, 42]. Since each vertex in a DAG can have more than one parent, the derivation methods are in general more complex than the methods used for chains or trees. There are many proposals that work on DAGs; typically they exploit a piece of public information associated with each vertex of the key derivation hierarchy. In [35], Atallah et al. introduce a new class of methods. The method in [35] maintains a piece of public information, called *token*, associated with each edge in the hierarchy. Given two keys, k_i and k_j arbitrarily assigned to two vertices, and a public label l_j associated with k_j, a token from k_i to k_j is defined as $T_{i,j}{=}k_j \oplus h(k_i, l_j)$, where $\oplus$ is the n-ary `xor` operator and h is a secure hash function. Given $T_{i,j}$, any user knowing k_i and with access to public label l_j, can compute (derive) k_j. All tokens $T_{i,j}$ in the system are stored in a *public catalog*.

It is important to note that the methods operating on trees can be used for chains of vertices, even if the contrary is not true. Analogously, the methods operating on DAGs can be used for trees and chains, while the converse is not true.

When choosing a key derivation method for the DAS scenario, it is necessary to take into consideration two different aspects: *1)* the client overhead and *2)* the cost of managing access control policy updates. The client overhead is mainly the communication and computation time for getting from the server the public information that is needed in the derivation process (e.g., tokens in [35]). The cost of enforcing access control policy updates is the cost of updating the key derivation hierarchy. As we will see later on, the key derivation hierarchy is used to correctly enforce the access control policy specified by the data owner, and therefore its definition is based on the access control policy itself. Intuitively, since the access control policy is likely to change over time, the hierarchy needs to re-arrange accordingly (i.e., insert or delete vertices, and modify keys). An important requirement is then to minimize the amount of re-encrypting and re-keying need in the hierarchy re-arrangement. Indeed, any time the key of a vertex is changed, at least the tuples encrypted with that key need to be re-encrypted by the data owner, and the new key should be given to all users knowing the old one. By analyzing the most important key derivation methods, we can observe that the key derivation methods operating on trees allow insertion and deletion of leaf vertices, without need of changing other keys in the tree. If, instead, an internal vertex v is inserted or deleted, all the keys of the vertices in the subtree rooted at v must be updated accordingly. Analogously, methods operating on DAGs and associating



(a) User-based hierarchy

(b) Resource-based hierarchy

Figure 7: Examples of key derivation hierarchies

public information with edges in the graph (e.g., Atallah's et al. [35]) allow insertion and deletion of vertices without need of re-keying operations. By contrast, all the other key derivation methods operating on DAGs require both to modify all keys derivable from the key that has been changed, and to re-encrypt all tuples previously encrypted by using the old keys. Among all the key derivation methods proposed, the key method proposed in [35] seems the method that better suits the DAS scenario.

Key derivation hierarchies. We now describe how it is possible to define a key derivation hierarchy that allows the correct enforcement of the access control policy defined by the data owner.

An access control policy $\mathcal{A}$ can be enforced by defining different key derivation hierarchies. In particular, a key derivation hierarchy can be defined according to two different strategies: *user-based* and *resource-based*. In the user-based strategy, the access control policy $\mathcal{A}$ is modeled as a set of access control lists, while in the resource-based strategy, it is modeled as a set of capabilities. We can then define a user-based or a resource-based hierarchy as follows.

A user-based hierarchy, denoted `UH`, is defined as a pair $(P(\mathcal{U}), \preceq)$, where $P(\mathcal{U})$ is the set containing all possible sets of users in the system, and $\preceq$ is the partial order relation induced by the set containment relation ($\subseteq$). More precisely, $\forall a, b \in P(\mathcal{U})$, $a \preceq b$ if and only if $b \subseteq a$. The user-based hierarchy contains therefore the set of all subsets of $\mathcal{U}$ and the corresponding DAG has $2^{|\mathcal{U}|}$ vertices. For instance, Figure 7(a) represents a user-based hierarchy built over a system with four users A, B, C, and D. To correctly enforce the access control policy, each vertex in the hierarchy is associated with a key, each resource in the system is encrypted by using the key of the vertex representing its *acl*, and each user is given the key of the vertex representing herself in the hierarchy. From the key of vertex u_i, user u_i can then derive the keys of the vertices representing groups of users containing u_i and therefore she can decrypt all the resources she can access (i.e., belonging to her capability list). Note that the empty set vertex represents a key known only to the data owner, and it is used to encrypt resources that nobody can access. As an example, consider the policy in Figure 6 and the hierarchy in Figure 7(a). Resource r_1 is encrypted with key k_{BC} of vertex

BC, r_2 with k_{ACD}, r_3 with k_{ABD}, and r_4 with k_{ACD}. Each user knows the key associated with the vertex representing herself and there is a path connecting each user's vertex with all the vertices representing a group containing the user. For instance, if we consider user A, from vertex A it is possible to reach vertices AB, AC, AD, ABC, ABD, ACD, and $ABCD$. Consequently, user A can decrypt r_2, r_3, and r_4, which are exactly the resources in her capability list.

A resource-based hierarchy, denoted `RH`, can be built in a dual way. The resource-based hierarchy is therefore defined as pair $(P(\mathcal{R}), \preceq)$, where $P(\mathcal{R})$ is the set of all subsets of $\mathcal{R}$, and order relation $\preceq$ is based on the set containment relation ($\subseteq$). In other words, $\forall a, b \in P(\mathcal{R})$, $a \preceq b$ if and only if $a \subseteq b$. For instance, Figure 7(b) represents a resource-based hierarchy built over a system with four resources r_1, r_2, r_3, and r_4. Like for the user-based hierarchy, to correctly enforce the access control policy, each vertex in the hierarchy is associated with a key, each resource in the system is encrypted by using the key of the vertex representing the resource itself in the hierarchy, and each user is given the key of the vertex representing her *capability*. From the key of the vertex corresponding to her capability, each user can compute the keys of the vertices representing resources belonging to her capability list. For instance, consider users B and C and resource r_1. Vertex r_1 can be reached starting from vertices $r_1r_2r_3r_4$, $r_1r_2r_3$, $r_1r_2r_4$, $r_1r_3r_4$, r_1r_2, r_1r_3, and r_1r_4. Users B and C know the key of vertices r_1r_3 and $r_1r_2r_4$, respectively, which are the vertices corresponding to their capability lists. Consequently, since there is a path from the key of B and C to the key used for encrypting r_1, B and C, which are exactly the users in the access control list of r_1, can access r_1.

Both the user-based and the resource-based hierarchies here defined correctly enforce the policy described in the access matrix $\mathcal{A}$, and both of them assign a unique secret key to each user of the system and define a unique key for encrypting each resource in the system. The most important difference between these approaches lays in the *key assignment method*. In a user-based hierarchy, a different key is assigned to each user, while tuples may share the same key (if they have the same access control list). By contrast, with a resource-based hierarchy, a different key is associated with each resource, while users may share the same key (if they have the same capability list). When deciding whether to adopt a user-based or a resource-based hierarchy, it is therefore important to determine whether users can share the same key for accessing the system. It is also important to note that the number of keys in the system depends on the number of users and resources in the system, respectively. Consequently, if the number of users is lower than the number of resources, it may be convenient to adopt `UH`.

Hierarchy reduction. For simplicity, we now focus our attention on `UH` (however, the following considerations are valid also for the resource-based hierarchy). It is easy to see that the solution described above defines more keys than actually needed and requires the publication of a great amount of information on the

remote server, thus causing an expensive key derivation process at the client-side. The higher is the number of users, the deeper is the key derivation hierarchy (the hierarchy height is equal to the number of users in the system). As an example, consider the user-based hierarchy in Figure 7(a) and, in particular, consider user A. To access resource r_3, A has to first derive k_{AD} that in turn can be used for deriving k_{ABD}, which is the key needed for decrypting r_3. However, in this case, vertex AD makes only the derivation process longer than needed and therefore it can be removed without compromising the correctness of the derivation process.

Since an important goal is to reduce the client's overhead, it is possible to simplify the key derivation hierarchy, removing non necessary vertices, while ensuring a correct key derivability. Therefore, instead of representing all the possible groups of users in the DAG, it is sufficient to represent those sets of users whose key is relevant for access control enforcement. Intuitively, these groups are those corresponding either to the *acl* values or singleton sets of users. The vertices corresponding to *acl*s and to users are necessary because their keys are used for resource encryption and allow users to correctly derive all the other keys used for encrypting resources in their capabilities, respectively. This set of vertices needs then to be correctly connected in the hierarchy. In particular, from the key of any user u_i it must be possible to derive the keys of all those vertices representing a group that contains u_i. Since in [30] the authors propose to use a user-based hierarchy in combination with the Atallah's et al. key derivation method, it is not advisable to connect each user's key directly with each group containing the user itself. Indeed, any time a client needs to derive a key, it queries the remote server to gain the tokens necessary for derivation. Another important observation is that when building the key derivation hierarchy, other vertices can be inserted, which are useful for reducing the size of the public catalog, even if their keys are not used for derivation. As an example, consider a system with five users and three *acl* values: ACD, ABD, and ADE. If vertices A, B, C, D, and E are connected directly with ACD, ABD, and ADE, the system needs nine tokens. If instead a new vertex AD is inserted and connected with the three *acl* values, A and D do not need an edge connecting them directly to each *acl* value, but they only need an edge connecting them with AD. In this case, the system needs eight tokens. Therefore, any time three or more vertices share a common parent, it is useful to insert such a vertex for saving tokens in the public catalog. Figure 8 illustrates the hierarchy corresponding to the access control policy in Figure 6 and containing only the vertices needed for a correct enforcement of the policy. The problem of correctly enforcing a policy through a key derivation graph while minimizing the number of edges in the DAG is however *NP-hard*. In [30] the authors have solved this problem through an approximation algorithm.

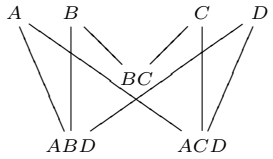


Figure 8: An example of reducted hierarchy enforcing the access control policy in Figure 6

4.1.2 Other Approaches

Damiani et al. [30] propose a method based on key derivation methods operating on trees and transform the original user-based hierarchy in a tree enforcing the same policy. In this case, each user has to manage more than one key. They propose therefore an approximation algorithm to the aim of limiting the average number of keys assigned to each user in the system.

Zych and Petkovic [43] propose an alternative access control enforcement method for outsourced databases, which exploits the Diffie-Hellman key generation scheme and asymmetric encryption. They define a user-based hierarchy that is then transformed into a *V-graph*. For each vertex in the V-graph, the number of incoming edges is either 2 or 0, and for any two vertices, there is at most one common parent vertex. The resulting structure is a *binary tree*, whose leaves represent singleton sets of users, and whose root represents the group containing all the users in the system. Here, the key derivation process goes from leaf vertices to the root. Each user is given the private key of the vertex representing herself in the hierarchy, while each resource is encrypted with the public key of the vertex representing its *acl*. Consequently, each user can compute, through derivation, the keys necessary to decrypt the resources she is authorized to access. This method requires: $O(E)$ public space, where E is the set of edges in the tree; $O(N)$ private space on clients, where N is the number of cells equal to 1 in the access matrix; and $O(n)$ derivation time, where n is the number of users in the system.

Key derivation hierarchies have also been adopted for access control enforcement in contexts different from the one introduced in this chapter. For instance, pay-tv systems usually adopt selective encryption for selective access enforcement and key hierarchies to easily distribute encryption keys [44, 45, 46, 47, 48]. Although these applications have some similarities with the DAS scenario, there are important differences that do not make them applicable in the DAS scenario. First, in the DAS scenario we need to protect stored data, while in the pay-tv scenario streams of data are the resources that need to be protected. Second, in the DAS scenario key derivation hierarchies are used to reduce the number of keys each user has to keep secret, while in the pay-tv scenario a key derivation hierarchy is exploited for session key distribution.

To guarantee security in the DAS scenario, physical devices have also been studied, both operating client-side [49] and server-side [50]. However, the usage of smart cards for clients and of secure co-processors for

the remote server has not been deeply studied. These methods can be adopted together with the security and querying solutions presented in this chapter.

5 Conclusions

Database outsourcing is becoming an emerging data management paradigm that introduces many research challenges. In this chapter, we focused on the problems related to query execution and access control enforcement. For query execution, different indexing methods have been discussed. These methods mainly focuses on supporting a specific kind of conditions or a specific SQL clause and on minimizing the client burden in query execution. Access control enforcement is instead a new issue for the DAS scenario and has not been deeply studied jet. The most important proposal for enforcing selective access on outsourced encrypted data is based on selective encryption. This method exploits cryptography for access control enforcement by using different keys to protect data. Each user is then given the set of keys allowing her to access exactly the resources belonging to her capability list.

There are however many other issues that need to be further investigated. The identification of techniques able to enforce updates that can modify the set of users, the set of resources, or their authorizations while maintaining a limited cost in terms of key reassignment or decryption/encryption is again an open issue. Another interesting issue is related to the management of write privileges; although there are solutions that provide data integrity by detecting non authorized modifications of database content, these solutions do not prevent unathorized modifications.

Acknowledgements

This work was supported in part by the European Union under contract IST-2002-507591, by the Italian Ministry of Research Fund for Basic Research (FIRB) under project RBNE05FKZ2, and by the Italian MIUR under project 2006099978.

References

[1] Davida, G., Wells, D., and Kam, J., A database encryption system with subkeys. *ACM Transactions on Database Systems*, 6:312, 1981.

[2] Hacigümüs, H., Iyer, B., and Mehrotra, S., Providing database as a service. In *Proc. of the 18th International Conference on Data Engineering*, San Jose, California, USA. IEEE Computer Society, 2002, 29.

[3] Iyer, B. et al., A framework for efficient storage security in RDBMS. In Bertino, E. et al., Eds., *Proc. of the International Conference on Extending Database Technology (EDBT 2004)*, volume 2992 of *Lecture Notes in Computer Science*, Crete, Greece. Springer, 2004, 147.

[4] Damiani, E. et al., Balancing confidentiality and efficiency in untrusted relational DBMSs. In Jajodia, S., Atluri, V., and Jaeger, T., Eds., *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS03)*, Washington, DC, USA. ACM, 2003, 93.

[5] Hacigümüs, H., Iyer, B., and Mehrotra, S., Efficient execution of aggregation queries over encrypted relational databases. In Lee, J., Li, J. Wndhang, K., and Lee, D., Eds., *Proc. of the 9th International Conference on Database Systems for Advanced Applications*, volume 2973 of *Lecture Notes in Computer Science*, Jeju Island, Korea. Springer, 2004, 125.

[6] Damiani, E. et al., Implementation of a storage mechanism for untrusted DBMSs. In *Proc. of the Second International IEEE Security in Storage Workshop*, Washington DC, USA. IEEE Computer Society, 2003, 38.

[7] Hacigümüs, H. et al., Executing SQL over encrypted data in the database-service-provider model. In *Proc. of the ACM SIGMOD 2002*, Madison, Wisconsin, USA. ACM Press, 2002, 216.

[8] Hore, B., Mehrotra, S., and Tsudik, G., A privacy-preserving index for range queries. In Nascimento, M. et al., Eds., *Proc. of the 30th International Conference on Very Large Data Bases*, Toronto, Canada. Morgan Kaufmann, 2004, 720.

[9] Wang, Z. et al., Fast query over encrypted character data in database. *Communications in Information and Systems*, 4:289, 2004.

[10] Wang, Z., Wang, W., and Shi, B., Storage and query over encrypted character and numerical data in database. In *Proc. of the Fifth International Conference on Computer and Information Technology (CIT'05)*, Shanghai, China. IEEE Computer Society, 2005, 77.

[11] Boyens, C. and Gunter, O., Using online services in untrusted environments - a privacy-preserving architecture. In *Proc. of the 11th European Conference on Information Systems (ECIS '03)*, Naples, Italy, 2003.

[12] Domingo-Ferrer, J. and Herrera-Joancomart, J., A privacy homomorphism allowing field operations on encrypted data, *Jornades de Matematica Discreta i Algorismica*, 1998.

[13] Evdokimov, S., Fischmann, M., and Gunther, O., Provable security for outsourcing database operations. In Liu, L. Reuter, A., Whang, K., and Zhang, J., Eds., *Proc. of the 22nd International Conference on Data Engineering (ICDE '06)*, Atlanta, GA, USA. IEEE Computer Society, 2006, 117.

[14] Agrawal, R. et al., Order preserving encryption for numeric data. In Weikum, G., König, A., and Deßloch, S., Eds., *Proc. of the ACM SIGMOD 2004*, Paris, France. ACM, 2004, 563.

[15] Aggarwal, G. et al.. Two can keep a secret: a distributed architecture for secure database services. In *Proc. of the Second Biennal Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, 2005, 186.

[16] Boneh, D. et al., Public-key encryption with keyword search. In *Proc. of the Eurocrypt 2004*, volume 3027 of *Lecture Notes in Computer Science*, Interlaken, Switzerland. Springer, 2004, 506.

[17] Brinkman, R., Doumen, J., and Jonker, W., Using secret sharing for searching in encrypted data. In Jonker, W. and M., P., Eds., *Proc. of the Secure Data Management Workshop*, volume 3178 of *Lecture Notes in Computer Science*, Toronto, Canada. Springer, 2004, 18.

[18] Goh, E., *Secure indexes*. http://eprint.iacr.org/2003/216/, 2003.

[19] Song, D., Wagner, D., and Perrig, A., Practical techniques for searches on encrypted data. In *Proc. of the 21st IEEE Computer Society Symposium on Research in Security and Privacy*, Berkeley, CA, USA. IEEE Computer Society, 2000, 44.

[20] Waters, B. et al., Building an encrypted and searchable audit log. In *Proc. of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA. Internet Society, 2004.

[21] Boneh, D. and Franklin, M., Identity-based encryption from the weil pairing. *SIAM Journal on Computing*, 32:586, 2003.

[22] Ceselli, A. et al., Modeling and assessing inference exposure in encrypted databases. *ACM Transactions on Information and System Security (TISSEC)*, 8:119, 2005.

[23] Damiani, E. et al., Measuring inference exposure in outsourced encrypted databases. In Gollmann, D. Massacci, F. and Yautsiukhin, A., Eds., *Proc. of the First Workshop on Quality of Protection*, volume 23 of *Advances in Information Security*, Milan, Italy. Springer, 2005.

[24] Samarati, P., Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13:1010, 2001.

[25] Hacigümüs, H., Iyer, B., and Mehrotra, S., Ensuring integrity of encrypted databases in database as a service model. In De Capitani di Vimercati, S., Ray, I., and Ray, I., Eds., *Proc. of the IFIP Conference on Data and Applications Security*, Estes Park Colorado. Kluwer, 2003, 61.

[26] Mykletun, E., Narasimha, M., and Tsudik, G., Authentication and integrity in outsourced database. In *Proc. of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, USA. The Internet Society, 2004.

[27] Narasimha, M. and Tsudik, G., DSAC: integrity for outsourced databases with signature aggregation and chaining. In Herzog, O., Schek, H., Fuhr, N., Chowdhury, A., and Teiken, W., Eds., *Proc. of the 14th ACM International Conference on Information and knowledge management*, Bremen, Germany. ACM, 2005, 235.

[28] Sion, R., Query execution assurance for outsourced databases. In Böhm, K., Jensen, C., Haas, L., Kersten, M., Larson, P., and Ooi, B., Eds., *Proc. of the 31st International Conference Very Large Data Bases*, Trondheim, Norway. ACM, 2005, 601.

[29] Mykletun, E., Narasimha, M., and Tsudik, G., Signature bouquets: Immutability for aggregated/condensed signatures. In Samarati, P., Ryan, P., Gollmann, D., and Molva, R., Eds., *Proc. of the European Symposium On Research in Computer Security (ESORICS 2004)*, volume 3193 of *Lecture Notes in Computer Science*, Sophia Antipolis, France. Springer-Verlag, 2004, 160.

[30] Damiani, E. et al., Selective data encryption in outsourced dynamic environments. In *Proc. of the Second International Workshop on Views On Designing Complex Architectures (VODCA 2006)*, Electronic Notes in Theoretical Computer Science, Bertinoro, Italy. Elsevier, 2006.

[31] 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, Texas, USA. IEEE Computer Society, 1987, 405.

[32] Gudes, E., The design of a cryptography based secure file system. *IEEE Transactions on Software Engineering*, 6:411, 1980.

[33] Sandhu, R., Cryptographic implementation of a tree hierarchy for access control. *Information Processing Letters*, 27:95, 1988.

[34] Akl, S. and Taylor, P., Cryptographic solution to a problem of access control in a hierarchy. *ACM Transactions on Computer System*, 1:239, 1983.

[35] Atallah, M., Frikken, K., and Blanton, M., Dynamic and efficient key management for access hierarchies. In Atluri, V., Meadows, C., and Juels, A., Eds., *Proc. of the 12th ACM conference on Computer and Communications Security (CCS05)*, Alexandria, VA, USA. ACM SIGSAC, 2005, 190.

[36] De Santis, A., Ferrara, A.L., and Masucci, B., Cryptographic key assignment schemes for any access control policy. Inf. Process. Lett., 92(4):199205, 2004.

[37] Harn, L. and Lin, H., A cryptographic key generation scheme for multilevel data security. *Computers and Security*, 9:539, 1990.

[38] Hwang, M. and Yang, W., Controlling access in large partially ordered hierarchies using cryptographic keys. *The Journal of Systems and Software*, 67:99, 2003.

[39] Liaw, H., Wang, S., and Lei, C., On the design of a single-key-lock mechanism based on newton's interpolating polynomial. *IEEE Transaction on Software Engineering*, 15:1135, 1989.

[40] MacKinnon, S. et al., An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Transactions on Computers*, 34:797, 1985.

[41] Shen, V. and Chen, T., A novel key management scheme based on discrete logarithms and polynomial interpolations. *Computer and Security*, 21:164, 2002.

[42] Crampton, J., Martin, K., and Wild, P., On key assignment for hierarchical access control. In *Proc. of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, Los Alamitos, CA, USA. IEEE Computer Society, 2006, 98.

[43] Zych, A. and Petkovic, M., Key management method for cryptographically enforced access control. In *Proc. of the 1st Benelux Workshop on Information and System Security*, Antwerpen, Belgium, 2006.

[44] Birget, J. et al., Hierarchy-based access control in distributed environments. In *Proc. of the IEEE International Conference on Communications*, volume 1, Helsinki, Finland. IEEE Computer Society, 2002, 229.

[45] Ray, I., Ray, I., and Narasimhamurthi, N., A cryptographic solution to implement access control in a hierarchy and more. In *Proc. of the 11th ACM Symposium on Access control Models and Technologies (SACMAT'02)*, Monterey, California, USA. ACM Press, 2002, 65.

[46] Tsai, H. and Chang, C., A cryptographic implementation for dynamic access control in a user hierarchy. *Computer and security*, 14:159, 1995.

[47] Wong, C., Gouda, M., and Lam, S., Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8:16, 2000.

[48] Sun, Y. and Liu, K., Scalable hierarchical access control in secure group communications. In *Proc. of the IEEE Infocom*, volume 2, Hong Kong, China. IEEE Computer Society, 2004, 1296.

[49] Bouganim, L. and Pucheral, P., Chip-secured data access: confidential data on untrusted servers. In Bernstein, P. et al., Eds., *Proc. of the 28th International Conference on Very Large Data Bases*, Hong Kong, China. Morgan Kaufmann, 2002, 131.

[50] Bouganim, L. et al., Chip-secured data access: Reconciling access rights with data encryption. In Freytag, J., Lockemann, P., Abiteboul, S., Carey, M., Selinger, P., and Heuer, A., Eds., *Proc. of the 29th VLDB conference*, Berlin, Germany. Morgan Kaufmann, 2003, 1133.