Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs

Ernesto Damiani
DTI - Università di Milano
26013 Crema - Italy
damiani@dti.unimi.it

S.De Capitani di Vimercati
DTI - Università di Milano
26013 Crema - Italy
decapita@dti.unimi.it

Sushil Jajodia
George Mason University
Fairfax, VA 22030-4444
jajodia@gmu.edu

Stefano Paraboschi
DIGI - Università di Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Pierangela Samarati
DTI - Università di Milano
26013 Crema - Italy
samarati@dti.unimi.it

ABSTRACT

The scope and character of today's computing environments are progressively shifting from traditional, one-on-one client-server interaction to the new cooperative paradigm. It then becomes of primary importance to provide means of protecting the secrecy of the information, while guaranteeing its availability to legitimate clients. Operating on-line querying services securely on open networks is very difficult; therefore many enterprises outsource their data center operations to external application service providers. A promising direction towards prevention of unauthorized access to outsourced data is represented by encryption. However, data encryption is often supported for the sole purpose of protecting the data in storage and assumes trust in the server, that decrypts data for query execution.

In this paper, we present a simple yet robust single-server solution for remote querying of encrypted databases on untrusted servers. Our approach is based on the use of indexing information attached to the encrypted database which can be used by the server to select the data to be returned in response to a query without the need of disclosing the database content. Our indexes balance the trade off between efficiency requirements in query execution and protection requirements due to possible inference attacks exploiting indexing information. We also investigate quantitative measures to model inference exposure and provide some related experimental results.

CCS'03, October 27–30, 2003, Washington, DC, USA.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*

General Terms

Security, Design

Keywords

Database service, cryptography, indexing

1. INTRODUCTION

In most organizations, databases hold a critical concentration of sensitive information. Ensuring an adequate level of protection to databases' content is therefore an essential part of any comprehensive security program. *Database encryption* [5] is a time-honored technique that introduces an additional layer to conventional network and application-level security solutions, preventing exposure of sensitive information even if the database server is compromised. Database encryption prevents unauthorized users, including intruders breaking into a network, from seeing the sensitive data in databases; similarly, it allows database administrators to perform their tasks without being able to access sensitive information (e.g., sales or payroll figures) in plaintext. Furthermore, encryption protects data integrity, as possible data tampering can be recognized and data correctness restored (e.g., by means of backup copies).

While much research has been made on the mutual influence of data and transmission security on organizations' overall security strategy [15], the influence of service outsourcing on data security has been less investigated. Conventional approaches to database encryption have the sole purpose of protecting the data in storage and assume trust in the server, which decrypts data for query execution. This

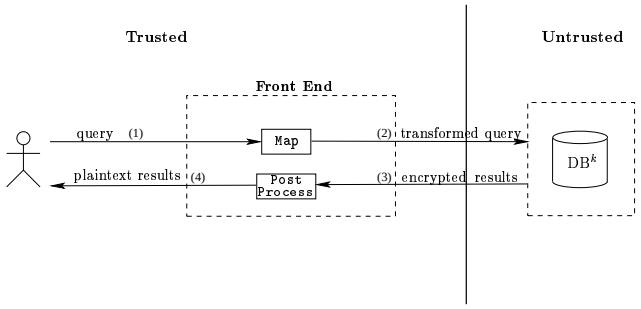


Figure 1: Overall scenario

assumption is less justified in the new cooperative paradigm, where multiple *Web services* cooperate and exchange information in order to offer a variety of applications. Effective cooperation between Web services and *data owners* often requires critical information to be made continuously available for on-line querying by other services or final users. To name but a few, *telemedicine* applications involve network transfers of medical data, *location-based services* require availability of users' cartographic coordinates, while *e-business decision support systems* often need to access sensitive information such as credit ratings.

Customers, partners, regulatory agencies and even suppliers now routinely need access to information originally intended to be stored deep within companies' information systems. Operating on-line querying services securely on open networks is very difficult; for this reason, many enterprises prefer to outsource their data center operations to external *application providers* rather than allowing direct access to their databases from potentially hostile networks like the Internet. Furthermore, outsourcing relational databases to external providers promises higher availability and more effective disaster protection than in-house operations. *Remote storage technologies* (e.g., storage area networks [16]) are used to place sensitive and even critical company information at a provider's site, on systems whose architecture is specifically designed for database publishing and access is controlled by the provider itself.

As a consequence of this trend toward outsourcing, highly sensitive data are now stored on systems run in locations that are not under the data owner's control, such as leased space and untrusted partners' sites. Therefore, data confidentiality and even integrity can be put at risk by outsourcing data storage and management. Adoption of security best practices in outsourced locations, such as the use of firewalls and intrusion detection tools, is not under the data owner's control. In addition, data owners may not entirely trust providers' discretion; on the other hand, preventing a provider from inspecting data stored on its own machines is very difficult. For this kind of services to work successfully it is therefore of primary importance to provide means of protecting the secrecy of the information remotely stored, while guaranteeing its availability to legitimate clients.

The requirement that the database content remains secret to the database server itself introduces several new interesting challenges. Conventional encrypted DBMSs assume trust in the DBMS itself, which can then decrypts data for query execution. In an outsourced environment scenario, such an assumption is not applicable anymore as the party to which the service is being outsourced cannot be granted full access to the plaintext data. Since confidentiality demands that data decryption must be possible only at the client side, techniques are needed enabling untrusted servers to execute queries on encrypted data. A first proposal towards the solution of this problem was presented in [8] were the authors proposed storing, together with the encrypted database, additional *indexing* information. Such indexes can be used by the untrusted DBMS to select the data to be returned in response to a query. The basic idea is illustrated in Figure 1. Each plaintext query (1) is mapped onto a corresponding query (2) on the indexing content and executed in that form at the untrusted server. The untrusted server returns the encrypted result (3), which is then decrypted at the trusted front end. If indexing information is not exact, an additional query (4) may need to be executed to eliminate spurious tuples that do not belong to the result set.

The major challenge in this scenario is how to compute and represent indexing information. Two conflicting requirements challenge the solution of this problem: on the 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 [6]. The indexing information provided in [8], based on using as indexes name of sets collecting together intervals of values, proves limited in this respect (see Section 5).

In this paper we provide an approach to indexing encrypted data constructed with efficiency and confidentiality in mind, providing a balance between the two. The contributions of this paper can be summarized as follows. First, we propose an approach to indexing encrypted data based on direct encryption and hashing. Second, we provide a measure of inference exposure of the encrypted/indexed data that nicely models the problem in terms of graph automorphisms. Finally, we enhance the indexing information to provide for efficient execution of interval-based queries.

2. DATA ORGANIZATION

We consider a relational DBMS where data are organized in tables (e.g., table ACCOUNTS in Figure 2) where the underlined attribute represents the key of the table. In principle, different granularity choices are possible for database encryption, such as encrypting at the level of whole tables, columns (i.e., attributes), rows (i.e., tuples) and cells (i.e., elements). Encrypting at the level of tables and columns implies that the whole table (column resp.) involved in a query should always be returned, providing therefore no means for selecting the data of interest and leaving to the client the burden of query execution on a possibly huge amount of data. On the other hand, supporting encryption at the finest possible level of single cells is also inapplicable as it would severely affect performance, since the client would be required to execute a possibly very large number of decrypt operations to interpret the results of queries [9]. In the same line as [8], we assume encryption to be performed at the tuple level. To provide the server with the ability to select a set of tuples to be returned in response to a query, we associate with each encrypted tuple a number of indexing attributes. An *index* can be associated with each attribute in the origi-

ACCOUNTS

Account	Customer	Balance
Acc1	Alice	100
Acc2	Alice	200
Acc3	Bob	300
Acc4	Chris	200
Acc5	Donna	400
Acc6	Elvis	200

ENC_ACCOUNTS1

Enc_tuple	I_A	I_C	I_B
x4Z3tfX2ShOSM	π	α	μ
mNHg1oC010p8w	ϖ	α	κ
WsIaCvfyF1Dxw	ξ	β	η
JpO8eLTVgwV1E	ϱ	γ	κ
qctG6XnFNDTQc	ς	δ	θ
4QbqC3hxZHklU	Γ	ϵ	κ

ENC_ACCOUNTS2

Enc_tuple	I_A	I_C	I_B
x4Z3tfX2ShOSM	π	α	μ
mNHg1oC010p8w	ϖ	α	κ
WsIaCvfyF1Dxw	ξ	β	μ
JpO8eLTVgwV1E	ϱ	β	κ
qctG6XnFNDTQc	ς	δ	μ
4QbqC3hxZHklU	Γ	δ	κ

Figure 2: A plaintext relation and possible corresponding encrypted relations

nal relation on which conditions need to be evaluated in the execution of queries.

Each plaintext relation is represented as a relation with an attribute for the encrypted tuple and as many attributes as indexes to be supported. More specifically, each plaintext tuple $t(A_1, \ldots, A_n)$ is mapped onto a tuple $t'(T_k, I_1, \ldots, I_m)$ where $m \leq n$, $t'[T_k] = E_k(t)$, with $E_k()$ denoting an invertible encryption function over key k, and each I_i corresponds to the index over some A_j. Figure 2 illustrates an example of a plaintext table ACCOUNTS and the corresponding encrypted/indexed[1] relation ENC_ACCOUNTS1 where `Enc_tuple` contains the encrypted triples, while I_A, I_C, and I_B are indexes over attributes `Account`, `Customer`, and `Balance` respectively. For the sake of readability we use easy-to-understand names for the attributes and table names in the encrypted schema and Greek letters as index values. Of course, in a real example, attributes and tables names would be obfuscated and actual values for indexes would be the results of an invertible encryption function and would then look like the ones reported for the encrypted tuples in Figure 2.

Let us now discuss how to represent indexing information.

A trivial approach to indexing would be to use the plaintext value of each cell. This approach is obviously not suitable as plaintext data would be disclosed.

An alternative approach providing the same fine-grained selection capability without disclosing plaintext values is to use the individual encrypted values as index. Then, for each indexed cell the outcome of an invertible encryption function over the cell value is used. Formally, $t[I_i] = E_k(t[A_i])$. Execution is simple: each plaintext query can be translated into a corresponding query on encrypted data by simply applying the encryption function to the values mentioned in the query. For instance, with reference to the tables in Figure 2, query "SELECT * FROM ACCOUNTS WHERE CUSTOMER = `Alice`" would be translated into "SELECT ENC_TUPLE FROM ENC_ACCOUNTS1 WHERE $I_C = \alpha$"

This solution has the advantage of preserving plaintext distinguishability and together with precision and efficiency in query execution, as all the tuples returned belong to the query set of the original query. In particular, the solution is convenient for queries involving equality constraints over the attributes. Also, since equality predicates are almost always used in the computation of joins, a join applied on two tables that use the same encryption function on the join attribute can be computed precisely.

As a drawback, however, in this approach encrypted values reproduce exactly the plaintext values distribution with respect to values' *cardinality* (i.e., the number of distinct values of the attribute) and frequencies; this could open the doors to frequency-based attacks (see next section).

A third alternative approach to counter these attacks, is to use as index the result of a *secure hash function* over the attribute values rather than straightforwardly encrypting the attributes; this way, the attribute values' distribution can be *flattened* by the hash function. A flexible characteristic of a hash function is the cardinality of its co-domain B, which allows us to adapt to the granularity of the represented data. When B is small compared with the cardinality of the attribute, the hash function can be interpreted as a mechanism that distributes tuples in B *buckets*; a good hash function (and a secure hash has to be good) distributes uniformly the values in the buckets. For instance, the ACCOUNTS table in Figure 2 can be indexed by hashing considering three buckets (α, β, δ) for I_C and two buckets (μ, κ) for I_B. The encrypted relation ENC_ACCOUNTS2 in Figure 2 can then be obtained when Alice is mapped onto α, `Bob` and `Chris` are both mapped onto β, while `Donna` and `Elvis` are both mapped onto δ. Also, `200` is mapped to κ while all other balance values are mapped onto μ. With respect to direct encryption, hash-based indexing provides more protection as different plaintext values are mapped onto the same index.

Using attribute hashes in remote tables permits an efficient evaluation of equality predicates within the remote server. If the same hash function is used to compute values of two attributes of different tables on which the equality predicate must be evaluated in the context of a join query, the join query itself can be efficiently computed at the remote server simply by combining all of the pairs of tuples characterized by the same hash value.

When direct encryption is used for indexing, the result returned by a query on the encrypted table is exactly the query set of the original query. The only task left for the front end is then decryption. By contrast, when hashing is used, the results will often include spurious tuples (all those belonging to the same bucket of the index) that will have to be removed by the front end receiving it. In this case, the additional burden on the front end consists in purging from the result returned by the remote server all the pairs of tuples that, once brought back in plaintext form, do not satisfy the equality predicate on the join attribute. Intuitively, every query Q of the front end corresponds to a query Q' to be passed onto the untrusted DBMS for execution over the encrypted database and a query Q'' to be executed at the front end on the results of Q'. To illustrate, consider the encrypted table ENC_ACCOUNTS2 in Figure 2 and the user query Q "SELECT BALANCE FROM ACCOUNTS WHERE CUSTOMER=`Bob`". The query is translated as Q' = "SELECT Enc_tuple FROM ENC_ACCOUNT2 WHERE $I_c = \beta$" for execution by the untrusted DBMS which returns the third and fourth encrypted tuple. The trusted front end then decrypts the two obtaining tuples third and fourth of the original table ACCOUNTS and eliminates the latter (whose presence was due to index collision) by reevaluating the condition.

[1]In the remainder of the paper, for the sake of simplicity, we shall designate this table format with the term encrypted.

3. INFERENCE EXPOSURE COEFFICIENTS

Being closely related to plaintext data, indexing information could open the door to inferences that exploit data analysis techniques to reconstruct the database content and/or break the indexing code. It is important to be able to evaluate quantitatively the level of exposure associated with the publication of certain indexes and therefore to determine the proper balance between index efficiency and protection.

There are different ways in which inference attacks could be modeled. We distinguish two notions that differ in the assumption about the attacker's prior knowledge. In common, the two scenarios have the fact that the attacker has complete access on the encrypted relations.

In the first case, which we call **Freq+DB**^{K} scenario, we assume the attacker is aware of the distribution of plaintext values in the original database. This knowledge can be exact (e.g., in a database storing accounting information the account holder list can be fully known) or approximated (e.g., the ZIP codes of the geographical areas of the account holders can be estimated based on population data). For the sake of simplicity, in the following we will assume exact knowledge (which represents the worst case scenario). In this scenario there are two possible inferences that the attacker can draw: *i)* the *plaintext content* of the database, that is, determine the existence of a certain tuple (or *association* of values) in the original database, and/or *ii)* the *indexing function*, that is, determine the correspondence between plaintext values and indexes.

In the second case, which we call **DB+DB**^{K} scenario, we assume the attacker has both the encrypted and the plaintext database. In this case the inference allows the attacker to *break the indexing function*, thus establishing the correlation between plaintext data and the corresponding index values. The hosting server has then available both the plaintext data and the corresponding indexes, by breaking indexing, the malicious server will then be able to decode any additional encrypted tuple that can be inserted in the database.

In the remainder of this section we introduce two coefficients to assess the exposure of indexes in the two scenarios above. The indexing code we refer to is direct encryption; besides being easier to understand, it can be regarded as a worst case situation of the general hashing indexing.

3.1 Freq+DB^K Exposure

To illustrate this scenario, let us consider the example in Figure 2. The attacker knows the encrypted ENC_ACCOUNTS1; also, she knows that attribute `Account` has unique values and she knows the values (and their occurrences) appearing independently in attributes `Customer` and `Balance`:

Customer={`Alice`,`Alice`,`Bob`,`Chris`,`Donna`,`Elvis`}

Balance={100,200,200,200,300,400}.

Although the attacker does not know which index corresponds to which plaintext attribute, she can determine the actual correspondence by comparing their occurrence profiles. Namely, she can determine that I_A, I_C, and I_B correspond to attributes `Account`, `Customer` and `Balance` respectively. The attacker can then infer that κ represents value 200 and index α represents value `Alice` (*indexing inference*). She can also infer that the plaintext table contains a tuple associating values `Alice` and 200 (*association inference*). The other occurrence of the index value corresponding to `Alice` (i.e., α) is associated with a balance other than 200. Since there are only three other possible values, the probability of guessing it right is 1/3. In other terms, the probability of each association depends on the combination of occurrences of its values.

Intuitively, the basic protection from inference in the encrypted table is that values with the same number of occurrences are indistinguishable to the attacker. For instance, all customers but `Alice` are indistinguishable from one another, as well as all amounts but 200. By contrast, `Alice` and 200 stand out being, respectively, the only customer appearing twice and the only balance appearing three times.

The exposure of an encrypted relation to indexing inference can then be thought of in terms of an equivalence relation where indexes (and plaintext values) with the same number of occurrences belong to the same equivalence class. For instance, denoting each equivalence class with a dot notation showing the attribute name and its number of occurrences (e.g., class A.1 contains all the values of attribute A that occur once), we obtain:

A.1 = $\{\pi, \varpi, \xi, \varrho, \varsigma, \Gamma\}$ = {`Acc1`,`Acc2`,`Acc3`,`Acc4`,`Acc5`,`Acc6`}
C.1 = $\{\beta, \gamma, \delta, \epsilon\}$ = {`Bob`,`Chris`,`Donna`,`Elvis`}
C.2 = $\{\alpha\}$ = {`Alice`}
B.1 = $\{\mu, \eta, \theta\}$ = {100,300,400}
B.3 = $\{\kappa\}$={200}

The quotient of the encrypted table with respect to the equivalence relation defined above is the following.

QUOTIENT TABLE

$\mathbf{qt}_A$	$\mathbf{qt}_C$	$\mathbf{qt}_B$
A.1	C.2	B.1
A.1	C.2	B.3
A.1	C.1	B.1
A.1	C.1	B.3
A.1	C.1	B.1
A.1	C.1	B.3

IC TABLE

$\mathbf{ic}_A$	$\mathbf{ic}_C$	$\mathbf{ic}_B$
1/6	1	1/3
1/6	1	1
1/6	1/4	1/3
1/6	1/4	1
1/6	1/4	1/3
1/6	1/4	1

The *exposure* of the encrypted table to inference attacks can then be evaluated by looking at the distinguishable characteristics in the quotient table. In particular, the association ⟨Alice,200⟩ (and its correspondence $\langle\alpha, \kappa\rangle$) can be spotted with certainty being the encounter of two *singleton* equivalence classes (C.2 and B.3). For the other values, probabilistic considerations can be made by looking at the IC table, that is the table of the inverse of the cardinalities of the equivalence classes. In fact, the probability of disclosing a specific association is the product of the inverses of the cardinalities. The exposure of the whole relation (or projection of it) can then be estimated as the average exposure of each tuple in it. Formally we can write the *exposure coefficient* $\mathcal{E}$ associated with an encrypted relation with inverse cardinality table IC as:

$$\mathcal{E} = \frac{1}{n} \sum_{i=1}^{n} \prod_{j=1}^{k} \mathrm{IC}_{i,j} \qquad (1)$$

Here, i ranges over the tuples while j ranges over the columns.

With reference to our example we have a value of $\mathcal{E} = 1/18$ for the protection of the whole relation, and a value of 1/3 for the pair ⟨`Customer`, `Balance`⟩.

Note how a long tailed distribution of values (i.e., many values having low occurrence) can decrease the exposure to association attacks. This reflects the fact that while the attacker has information on many values, they all fall into

the same equivalence class resulting indistinguishable from one another.

Taking into account the fact that each index value corresponds to a single plaintext one, the exposure computed above may be regarded as a lower bound to vulnerability to association inference. Let us then now consider the effect on the exposure when indexes are obtained by hashing values rather than by direct encryption. In this case each hashed value can correspond to multiple attribute values (as it is the case for index value β in table Enc_Accounts2). Therefore, each equivalence class on hashed values can be composed of multiple subsets of plaintext values. For instance, any of the index values α, β, and δ in Enc_Accounts2 can correspond to any pair of customer values.

In a scenario where all plaintext values are distinct, their hashed values multiplicity is entirely due to collision. Then, each hash value in the equivalence class of multiplicity k can represent any k values extracted from the original set, that is, there are $\binom{n}{k}$ different possibilities. The identification of the correspondence between hashed and original values would require finding all possible partitions of the original values such that the sum of their occurrences is the cardinality of the hashed value. Computing the corresponding re-arrangement of equivalence classes would then equate to solving a *knapsack problem* [4]. In general this introduces a high degree of uniformity in the indexes and inference attacks become negligible.

3.2 DB+DB^k Exposure

We now consider a situation where the attacker knows both the encrypted and the plaintext database. A scenario for this attack occurs when the database owner switches from no encryption to the use of encryption as presented in this paper. A malicious user with access to the database server may then be very interested in reconstructing the correspondence between the plaintext and index values, in order to monitor the evolution of the database and keep access to most of its content, independently of the strength of the encryption function adopted.

The attacker knows precisely the distribution of every value and the relationships among the different values. The additional knowledge available to the attacker requires a more precise model than the one used in the previous section. We present a model for the attack that permits also to use robust and consolidated algorithmic techniques to derive a precise characterization of the solution space and of the exposure coefficient that needs to be redefined in this context.

3.2.1 The RCV-graph

Given a table $\mathcal{T}$ with attributes $A_1, A_2, \ldots A_n$ and tuples $t_1, t_2, \ldots t_m$, we build a 3-colored undirected graph $G = (V, E)$ called the RCV-graph (i.e., the row-column-value–graph) in the following way. The set of vertices V contains one vertex for every attribute (all of color "column"), one vertex for every tuple (all of color "row"), and one vertex for every distinct value in each of the attributes (all of color "value"); if the same value appears in different attributes, a distinct vertex is introduced for every attribute in which the value appears. The set of edges E is built first adding edges connecting the vertices representing columns with the vertices representing values appearing in the corresponding columns; then, edges are added connecting each tuple vertex with the vertices representing values appearing in the tuple. To illustrate, consider table Enc_Accounts1 in Figure 2, restricted to attributes `Customer` and `Balance`. We have two vertices labeled Col_1 and Col_2 for the attributes, six vertices labeled $t_1 \ldots t_6$ for the tuples, and nine vertices labeled $\alpha \ldots \theta$ for the distinct values appearing in the attributes. The addition of all the edges produces the RCV-graph depicted in Figure 3.

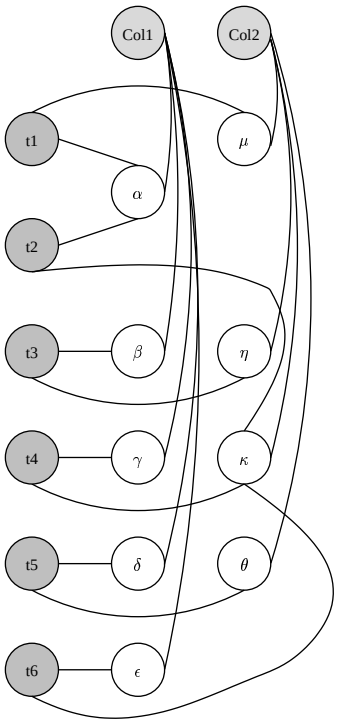


Figure 3: The RCV-graph from relation Enc_Accounts1 of Figure 2

An important property is that the RCV-graph built starting from the plaintext database is identical to the RCV-graph built starting from the encrypted database, since the cryptographic function only realizes a biunivocal mapping between plaintext and index values (in the relational model, the order of tuples and the order of attributes within a relation are irrelevant). The identification of the correspondence between plaintext and index values requires then to establish a correspondence between the vertex labels and the plaintext values discussed in the following section.

3.2.2 RCV-graph Automorphism

The identification of the correspondence between the labels on the graph $G = (V, E)$ and the plaintext values, when the plaintext database is known, can exploit information on the topological structure of the data that permits a more precise reconstruction than the one possible when the only information available is the distribution of values in each attribute. In the example, it is possible to correctly identify the correspondence between label Col_1 and attribute `Customer`, label Col_2 and attribute `Balance`, label α and value *Alice*, label μ and value 100, label κ and value 200. Also, label t_1 will be associated with the first tuple and label t_2 with the second. For the remaining vertices it is only

possible to obtain a probabilistic estimate of the correspondence.

The search for a correspondence that we above realized on the intuition, is strongly related to the presence of automorphisms in the RCV-graph. An automorphism of a graph is an isomorphism of the graph with itself. Formally, an automorphism of a graph is a permutation Γ of the graph labels such that $G(V,E) = G(V^{\Gamma},E)$ (i.e., $\forall e(v_i,v_j) \in E, e(v_i^{\Gamma},v_j{}^{\Gamma}) \in E$). If the graph is colored (as in our case), nodes with different color cannot be exchanged by the permutation. The identical permutation trivially satisfies the relationship; then, at least one automorphism exists for any graph. When the RCV-graph presents only the trivial automorphism, the correspondence between the vertex labels and the plaintext values can be fully determined and the knowledge of the plaintext database permits a full reconstruction of the correspondence between plaintext and index values. When there are several automorphisms in the RCV-graph, the identification of a vertex can be uncertain, as there are many alternative ways to reconstruct the correspondence between the vertices. In the example, the RCV-graph presents 4 automorphisms, that we represent here by the permutations of labels that characterize them. Each permutation is represented by a different order of the symbols in the following sequences.

$\mathbf{A}_1$ $\{Col_1, Col_2, t_1, t_2, t_3, t_4, t_5, t_6, \alpha, \beta, \gamma, \delta, \epsilon, \mu, \eta, \kappa, \theta\}$
$\mathbf{A}_2$ $\{Col_1, Col_2, t_1, t_2, t_5, t_4, t_3, t_6, \alpha, \delta, \gamma, \beta, \epsilon, \mu, \theta, \kappa, \eta\}$
$\mathbf{A}_3$ $\{Col_1, Col_2, t_1, t_2, t_3, t_6, t_5, t_4, \alpha, \beta, \epsilon, \delta, \gamma, \mu, \eta, \kappa, \theta\}$
$\mathbf{A}_4$ $\{Col_1, Col_2, t_1, t_2, t_5, t_6, t_3, t_4, \alpha, \delta, \epsilon, \beta, \gamma, \mu, \theta, \kappa, \eta\}$

The 4 automorphisms derive from the choice in the order of the two vertices sets (t_3,β,η)-(t_5,δ,θ) and (t_4,γ)-(t_6,ϵ). The number of automorphisms could appear as a measure of the protection against inference attacks, but we observe that it is not a good evaluator. In fact, in all the databases we studied the number of automorphisms increases exponentially with the size of the graph and may reach considerable (and inexpressive) values even for graphs of limited size; also, situations with evidently different protection may be characterized by the same number of automorphisms. We devised a more precise measure of protection, which considers the number of alternatives that are offered for the value of a label. The intuition is the following: for each value in a tuple in the database, we may have a given probability of guessing it based on the knowledge of the plaintext database: if all the RCV-graph automorphisms do not permute the corresponding vertex, we will have a probability ($p = 1$) of identifying its correct value. In general, if we have K automorphisms for the RCV-graph and in k of them the label assigned to vertex v_i is correct, we will have a probability $p_i = k/K$ of correctly identifying the vertex (i.e., we ignore row and column vertices). Since we are interested only in the identification of the correspondence for the vertices representing attribute values, we limit the consideration of the exposure coefficient to these nodes. Given the value p_i of each vertex v_i representing an attribute value, we estimate the probability of guessing right a generic value by computing the average on all the vertices of the probability p_i, obtaining the *attribute exposure coefficient* $AEC = \sum_{i=1}^{m} p_i/m$.

The automorphism problem has been extensively studied in the context of graph theory and many results can be directly applied to our context. First, the set of automorphisms of a graph constitute a group (called the *Automorphism Group* of the graph), which, for undirected graphs like ours, can be described by the coarsest *equitable partition* [12] of the vertices, where each element of the partition (each subset appearing in the partition) contains vertices that can be substituted one for the other in an automorphism. The *Nauty* algorithm that identifies the automorphism group of the graph [12], starts from a partition on the vertices that can be immediately derived grouping all the vertices with the same color and connected by the same number of edges. This partition is then iteratively refined, and a concise representation of all the automorphisms is produced. From the structure of the partition, it derives that all the vertices appearing in the generic partition element C_j are equivalently substitutable in all the automorphisms; from this observation, it derives that the probability p_i of a correct identification of a vertex $v_i \in C_j$ is equal to the inverse of the cardinality of C_j, $1/\mid C_j \mid$.

Then, for the identification of the AEC it is sufficient to identify the number of elements in the equitable partition and the total number of attribute vertices (i.e., it is not necessary to keep track of the number of vertices in each partition). In fact, with $\mid C_j \mid$ vertices in the partition element C_j, n elements in the equitable partition and a total number m of vertices, the exposure coefficient of the table is:

$$\sum_{i=1}^{m} p_i/m = \sum_{j=1}^{n}\sum_{v_i \in C_j} p_i/m = \sum_{j=1}^{n}\sum_{v_i \in C_j} 1/(\mid C_j \mid m) = \sum_{j=1}^{n} 1/m = n/m$$

In the example, the equitable partition for attribute vertices is $\{(\alpha)(\beta,\delta)(\gamma,\epsilon)(\mu)(\eta,\theta)(\kappa)\}$ and the $AEC = 6/9 = 2/3$. As a check, the reader can verify on the RCV-graph that the vertices appearing in singleton elements are associated with $p_i = 1$ and those in the remaining elements are associated with $p_i = 1/2$. The average of p_i on all the vertices returns $2/3$.

When the structure of the database is completely absent, as it occurs when all the attribute values appear once in the database, the AEC is minimal at $1/m$. The contribution of the knowledge of the plaintext database increases when the structure of the RCV-graph derived from it can impose restrictions that limit the number of options for a vertex, increasing the exposure coefficient.

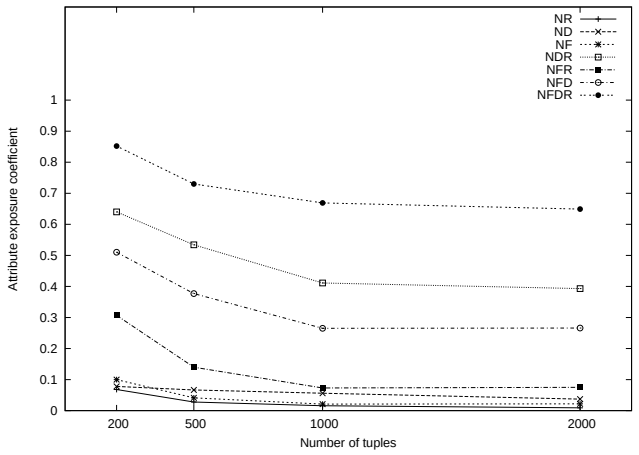
3.2.3 Experimental results

We implemented a tool that takes in input a relational database and builds the RCV-graph that models it, with the construction presented in the previous section. The program then invokes on the RCV-graph the Nauty algorithm [12], which is able to compute efficiently the automorphism group (around 15 minutes on a 700MHz Pentium III PC running Linux, for the greatest RCV-graph derived from a 2000 4-tuple table containing 2262 distinct values). The output of the program is then analyzed to reconstruct the equitable partition, that permits to determine the attribute exposure coefficient of the table. The tool was the basis for an analysis of the evolution of the exposure coefficient for tables of progressively increasing size and for an increasing number of indexes.

We retrieved from an Italian government site a table describing the professors of the universities in our region. The table listed 2340 professors, with 4 attributes: *Name* (first

Attributes	Number of tuples 200	500	1,000	2,000
{Name,Role}	14/206	14/507	16/1007	18/2006
{Name,Disc.}	21/269	39/585	61/1093	83/2234
{Name,Fac.}	21/209	21/509	21/1009	44/2019
{Name,Role,Disc.}	176/275	316/592	452/1100	881/2241
{Name,Fac.,Role}	66/215	72/516	74/1017	152/2027
{Name,Fac.,Disc.}	141/278	224/594	292/1103	599/2255
{Name,Fac.,Role,Disc.}	242/284	439/601	743/1110	1467/2262

(a)



(b)

Figure 4: Tabular representation of the experimental results (a) and their graphical representation (b) (curve labels refer to the initial of the attributes)

name and last name combined in a single attribute), *Faculty* (the name of the professor faculty), *Role* (the status of the professor: full, associate, assistant, etc., following the Italian structure) and *Discipline* (the field of the professor, using a classification by the Italian government). We applied the tool using a progressively greater number of tuples; we did not use the full table, but stopped at 2000 tuples. We considered all the combinations of attributes containing attribute *Name*; this choice was due to the fact that the analysis is meaningful only if at least two attributes are present in the table (otherwise, no correlation among attributes can be observed) and it was useful to keep a common attribute in all the experiments. The results appear in tabular form in Figure 4(a) and graphically in Figure 4(b).

The main result of the experiments is the finding that the number of attributes used for the index has a great impact on attribute exposure. With only two attributes, exposure coefficients tend to be quite low; when all the 4 attributes are used as index, the exposure is considerable.

Another question answered by the experiments is how the exposure evolves with an increase in the database size. What we observe is that the exposure slowly decreases with the size of the database. The explanation is that as the number of tuples increases, a greater number values become characterized by a distinct profile and are identifiable. At the same time, the new tuples introduce new values that are infrequent and indistinguishable, and this component wins over the former.

4. SUPPORTING INTERVAL-BASED QUERIES

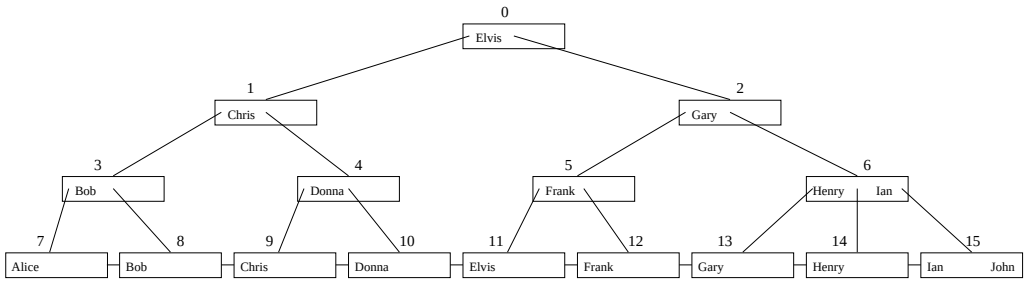
The solution presented in Section 2 does not support well interval-based queries, as a condition over an interval I would need to be mapped onto as many equality conditions as there are values in I. Of course, an efficient support for interval-based queries would be possible by using *order-preserving encryption* (imposing $E_k(t_i[A]) < E_k(t_j[A])$ whenever $t_i[A] < t_j[A]$). However, this solution is not viable as comparing the ordered sequences of index and plaintext values would lead an easy reconstruct the correspondence.

Interval-based queries are efficiently supported in traditional DBMSs with the use of *B+-trees* (whose construction and management is left to the DBMS). B+-trees are one of the most common solutions for the construction of indexes. A B+-tree (balanced tree) satisfies the constraint that the number of arcs that is necessary to traverse to go from the root to a leaf is the same for every leaf. A B+-tree with fan out F is a B-tree where there are no nodes with a number of outgoing arcs greater than F; additionally, all the non-leaf nodes, except the root, must have at least $\lceil F/2 \rceil$ outgoing arcs.

Given a relation $\mathcal{R}$ and a subset of its attributes K, a B+-tree built on the *key* K of $\mathcal{R}$ permits to access the tuples of $\mathcal{R}$ with a given value for the key. Each node presents $f-1$ key values, where f is the number of outgoing arcs; each key value k_i is associated with the outgoing arc a_i, except for the first arc a_0. To access a tuple characterized by key value $\overline{k}$, the nodes are considered starting from the root, and in each node if $\overline{k}$ is greater than or equal to k_i and smaller than k_{i+1}, the arc a_i is chosen; if $\overline{k}$ is smaller than k_1, arc a_0 is chosen; if $\overline{k}$ is greater than or equal to k_{f-1}, the arc a_{f-1} is chosen. In the leaves, the arcs are replaced by the IDs of the encrypted tuples. A B+-tree structure has in each leaf an outgoing arc that connects each leaf to the one following it in the order supported by the tree. This pointer supports the evaluation of range predicates (the inferior limit of the interval is first searched on the tree; then, the leaves are navigated, until the superior limit of the search interval is reached). Figure 5(a) illustrates an example of B+-tree built on attribute `Customer`. Here, each node can include two key values.

In our context, however, the untrusted DBMS only knows the encrypted data and any B+-tree defined on the indexes, not reflecting the order of the plaintext values, is practically useless for query execution. The only possible way to solve this problem is to leave the task of determining B+-tree information[2] to the trusted front end. The B+-tree can then be encrypted and stored at the untrusted DBMS (as the trusted front end has limited storage capacities). Obviously, protecting the B+-tree by encrypting each of its fields is unfeasible, as it would disclose to the server the ordering relationship between the index values. As an alternative, we propose to encrypt each B+-tree node as a whole. The original B+-tree is then represented at the untrusted DBMS as a table with two attributes: a node ID, automatically assigned by the system on insertion, and an encrypted value, representing the node content. Figure 5(b) shows the plaintext representation of the B+-tree table and its encrypted counterpart.

[2]We purposely avoid the use of the term index so not to create confusion with the indexes introduced in Section 2.



(a)

B+-tree Table

ID	Node
0	(1,Elvis,2,_,_)
1	(3,Chris,4,_,_)
2	(5,Gary,6,_,_)
3	(7,Bob,8,_,_)
4	(9,Donna,10,_,_)
5	(11,Frank,12,_,_)
6	(13,Henry,14,_,_)
7	(Alice,_,8)
8	(Bob,_,9)
9	(Chris,_,10)
10	(Donna,_,11)
11	(Elvis,_,12)
12	(Frank,_,13)
13	(Gary,_,14)
14	(Henry,_,15)
15	(Ian,John,_)

Encrypted B+-tree Table

ID	C
0	SeCS0U/7ZIY.A
1	/WKu5y8laqK82
2	jzKzVi0D1as8E
3	AXYaqohgyVObU
4	IUf7R.PK5h5fU
5	rzaslxohWS2l2
6	EXITGCUlYTVBc
7	uOtdm/HDXNSqU
8	GLDWRnBGIvYBA
9	a9yl36PA3LeLk
10	H6GwdJpXiU8MY
11	uOtdm/HDXNSqU
12	zj33kVaNvLFVk
13	V9rMw904cix3w
14	xTFcWtd6.IE.A
15	ji.gtDER6Hjis

(b)

Figure 5: An example of B+-tree on attribute `Customer` (a) and corresponding tabular and encrypted representation (b)

The advantage of this solution is that the content of the B+-tree nodes is not visible to the untrusted DBMS. The drawback, however, is that the B+-tree traversal can now only be performed by the trusted front end. Intuitively, to execute an interval query, the front end has to perform a sequence of queries that retrieve tree nodes at progressively deeper levels; when the leaf is reached, the node IDs in the leaf can be used to retrieve the tuples belonging to the interval. For instance, if it is necessary to retrieve all the customers whose name starts with a letter in the interval D-F using the B+-tree, the front end will produce a sequence of queries that will access in sequence nodes 0, 1, 4, and 10; then, other queries will be used to scan the leaves, accessing nodes 10, 11, and 12. In the Appendix we present a simple cost model that permits to identify the optimal size for the tree node.

5. RELATED WORK

Database encryption has been proposed since long as a fundamental tool for providing strong security for "data at rest". Thanks to recent advances in processors' capabilities and to the development of fast encryption techniques, the notion of encrypted database is nowadays well recognized, and several commercial products reached the market. However, developing a sound security strategy including database encryption still involves many open issues. Key management and security are of paramount importance in any encryption-based system and were therefore among the first issues to be investigated in the framework of database encryption [5]. Later, techniques have been developed aimed at efficiently querying encrypted databases [14], some of them related to parallel efforts by the text retrieval community [11] for executing *hidden queries*, that is, queries where only the ciphertext of the query arguments is made available to the DBMS. On the other hand, architectural research investigated optimal sharing of the encryption burden between secure storage, communication channels and the application where the data originates [10], looking for a convenient trade-off between data security and application performance. Recently, much interest was devoted to secure handling of database encryption in distributed, Web based execution scenarios, where data management is outsourced to external services [1]. The main purpose of this line of research is finding techniques for delegating data storage and the execution of queries to untrusted servers while preserving efficiency. The *index of range* technique proposed in [8] in the framework of a *database-service-provider* architecture relies on partitioning client tables' attributes' domains into sets of intervals. The value of each remote table attribute is stored as the index countersigning the interval to which the corresponding plain value belongs. Indexes may be ordered or not, and the intervals may be chosen so that they have all the same length, or are associated with the same number of tuples. This representation supports efficient evaluation on the remote server of both equality and range predicates; however, it makes it awkward to manage the correspondence between intervals and the actual values present in the database. A distinct, though related solution is proposed in [1], where smart cards are used for key management.

On a different line of related work, we note that the protection/exposure given by hashing can resemble the generalization approach for microdata protection; and correspondingly inference attacks exploiting it can resemble record linkage techniques examined in that context [13]. However, the two problems are quite different as while generalization maintains information on plaintext values (simply collapsing more values in a given interval), hashing is not associated with any semantics.

Also, it is important to note that the problem we consider differs from existing approaches protecting encrypted data, which investigated solutions as for the private information retrieval problem (protecting the query, that is, the information on what the user is looking for) or the problem of limiting the amount of data that users can acquire.

6. CONCLUSIONS

In this paper, we proposed a solution to the problem of database outsourcing on untrusted servers by providing a hash-based method for database encryption suitable for selection queries. Also, we gave a quantitative evaluation of our method's vulnerability, showing that even straightforward direct encryption can provide an adequate level of security against inference attacks, as long as a limited number of index attributes is used. In order to execute interval-based queries, we adapted to the database-service-provider model the B+-tree structures typically used inside DBMSs.

7. ACKNOWLEDGMENTS

The work reported in this paper was partially supported by the Italian MURST within the KIWI and MAPS projects.

8. REFERENCES

[1] L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *Proc. of the 28th International Conference on Very Large Data Bases*, pages 131–142, Hong Kong, China, August 2002.

[2] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Orlando, Florida, 1982.

[3] H. Chao, T.Y. Wu, and J. Chen. Security-enhanced packet video with dynamic multicast throughput adjustment. *International Journal of Network Management*, 11(3):147–159, 2001.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[5] G.I. Davida, D.L. Wells, and J.B. Kam. A database encryption system with subkeys. *ACM Transactions on Database Systems*, 6(2):312–328, June 1981.

[6] D.E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[7] S. Ghandeharizadeh and D. DeWitt. A multiuser performance analysis of alternative declustering strategies. In *Proc. of the 6th Int. Conf. on Data Engineering*, 1990.

[8] H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of the ACM SIGMOD'2002*, Madison, Wisconsin, USA, June 2002.

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

[10] C.D. Jensen. Cryptocache: a secure sharable file cache for roaming users. In *Proc. of the 9th Workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 73–78, Kolding, Denmark, September 2000.

[11] S.T. Klein, A. Bookstein, and S. Deerwester. Storing Text retrieval systems on CD-ROM: compression and encryption considerations. *ACM Transactions on Information Systems*, 7(3):230–245, July 1989.

[12] B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[13] P. Samarati. Protecting respondent's privacy in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1017, November/December 2001.

[14] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. of the 2000 IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, USA, May 2000.

[15] J.P. Walton. Developing an enterprise information security policy. In *Proc. of the 30th annual ACM SIGUCCS Conference on User Services*, Providence, Rhode Island, USA, 2002.

[16] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic storage area network fabric design. In *Proc. of the Conference on File and Storage Technologies (FAST 2002)*, Monterey, CA, January 2002.

[17] E.Y. Yang, J. Xu, and K.H. Bennett. Private information retrieval in the presence of malicious failures. In *Proc. of the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002.

APPENDIX

A. B+-TREE NODE SIZE CONFIGURATION

A critical aspect in the design of the B+-tree is the size of the node. When B+-trees are used to build indexes in a DBMS, the size of the I/O block typically defines the size of the node, as the goal is to minimize the number of I/O requests. For the B+-trees for this environment we have greater flexibility and indeed the size of the node becomes an important configuration parameter.

We build a simplified cost model that can be used to tune the node size for a specific environment. The parameters of the model are:

- Setup cost of a query that retrieves a single node given its ID, K_Q: there is a cost for every query execution, due to the fact that the query has to be transferred from the client to the server, the server has to understand it and execute it. Since we are interested in the modeling of accesses to the nodes of an encrypted B+-tree, we assume that the cost is the same for all queries.
- Size of the node, s_n: the size of the node depends on the number f of key values appearing in the node and on the size of keys s_k and on the size of pointers s_p.
- Cost C_t for the transmission and processing of a bit (we assume the cost to be constant; this is a good approximation for transmission costs and reasonable for the processing costs associated with the retrieval of data on the server and parsing on the client).
- The number N of tuples in $\mathcal{R}$.

The above terms can be used to build a formula that estimates the cost to access one node:

$$K_Q + C_t \times (f \times s_k + (f+1) \times s_p)$$

When the tree is used to search a value $\overline{k}$ among N, $log_f(N)$ tree nodes will be accessed; the following formula estimates the cost required to access a tree leaf:

$$C = log_f(N) \times (K_Q + C_t \times (f \times s_k + (f+1) \times s_p))$$

The optimal value for f will be one that minimizes the above cost function. The application of analytical methods to the function (variable f is replaced by the canonical analytical variable x) shows that the function can be also represented by the simpler structure:

$$C = \frac{\alpha + \beta x}{\log x}$$

Parameters α and β are both positive. The function is continuous and, considering for x the interval $1 - \infty$, it diverges at $x = 1$ and it also diverges as $x \to \infty$. The identification of the minimum can be based on an analytical study of the function. In particular, we compute the function derivative, obtaining:

$$C' = \frac{\beta x(\log x - 1) - \alpha}{x \log^2 x}$$

Function C' presents a single zero in the interval $(1 - \infty)$, that corresponds to the minimum in the cost function C. To identify the optimal x value in the domain of integers, it is sufficient to compare the two values for C that derive assigning to f the two integers obtained by rounding with *floor* and *ceiling* the minimum obtained by the numerical resolution. For instance, with 1,000,000 tuples, 0,01s query setup cost, $4 * 10^{-7}$ s/bit transmission cost, 30 bytes for a key and 2 bytes for a pointer, we obtain the optimum at $f = 38$.