Privacy-Preserving Queries on Encrypted Data*

Zhiqiang Yang[1], Sheng Zhong[2], and Rebecca N. Wright[1]

[1] Computer Science Department, Stevens Institute of Technology, Hoboken, NJ 07030 USA
[2] Computer Science and Engineering Department, SUNY Buffalo, Amherst, NY 14260 USA

Abstract. Data confidentiality is a major concern in database systems. Encryption is a useful tool for protecting the confidentiality of sensitive data. However, when data is encrypted, performing queries becomes more challenging. In this paper, we study efficient and provably secure methods for queries on encrypted data stored in an outsourced database that may be susceptible to compromise. Specifically, we show that, in our system, even if an intruder breaks into the database and observes some interactions between the database and its users, he only learns very little about the data stored in the database and the queries performed on the data.

Our work consists of several components. First, we consider databases in which each attribute has a finite domain and give a basic solution for certain kinds of queries on such databases. Then, we present two enhanced solutions, one with a stronger security guarantee and the other with accelerated queries. In addition to providing proofs of our security guarantees, we provide empirical performance evaluations. Our experiments demonstrate that our solutions are fast on large-sized real data.

1 Introduction

As anyone who reads newspapers is aware, there have been a staggering number of data breaches reported in the last two years. Some of the largest of these revealed sensitive information of millions of individuals. For example, in June 2005, names and credit card numbers of more than 40 million MasterCard cardholders were exposed [12]. In May 2006, disks containing the names, social security numbers, and dates of birth of more than 26 million United States veterans were stolen from the home of an employee of the Department of Veterans Affairs [29]. In the wrong hands, this kind of sensitive information can be to carry out identity theft and other fraudulent activities that harm the individuals involved and have a large cost to society.

Techniques such as access control, intrusion detection, and policies about how data is to be used attempt to prevent such thefts and intrusion. However, existing techniques cannot ensure that a database is fully immune to intrusion and unauthorized access.

Encryption is a well-studied technique to protect sensitive data [13] so that even if a database is compromised by an intruder, data remains protected even in the event that a database is successfully attacked or stolen. Provided that the encryption is done properly and the decryption keys are not also accessible to the attacker, encryption can provide protection to the individuals whose sensitive data is stored in the databases,

* This work was supported by the National Science Foundation under Grant No. CCR-0331584.

reduce the legal liability of the data owners, and reduce the cost to society of fraud and identity theft.

While encrypting the data provides important protection, encrypted data is much less convenient to use encrypted data than to use cleartext data. Specifically, in a database that stores encrypted data, how can queries be processed? If a database user fully trusts the database server, she can simply send the encryption key to the database server together with her query. However, because the database may be compromised at some point during such an interaction, revealing the key to the database server is at the risk of leaking all sensitive data to an intruder. Theoretically, if efficiency was not a concern, a user could retrieve all encrypted tables from the database, decrypt the tables, and then perform queries on the cleartext tables. However, this is clearly impractical when we take efficiency into consideration.

In this paper, we investigate efficient methods for processing queries on encrypted data in such a way that the data remains secure even if the database server may be compromised at some point by an intruder. Such methods are very useful in strengthening the protection of sensitive data in databases.

1.1 Related Work

Various methods have been proposed recently for securing databases in various settings [7, 27, 30, 22, 25, 21]. In particular, encryption is an important technique to protect sensitive data [13]. An analysis of how to encrypt and securely store data in relational database management systems has been given in [24]. Recognizing the importance of encryption techniques, some database vendors have included encryption functionality in their products [1, 2]. By considering different privacy policies for different data records, Hippocratic databases, which combine privacy policies with sensitive data, are very useful in preventing unauthorized users from accessing sensitive data [3].

With data stored in an encrypted form, a crucial question is how to perform queries. Hacigumus et al. [19] studied querying encrypted data in the database-as-service (DAS) model where sensitive data is outsourced to an untrusted server [20]. Their solution divides attribute domains into partitions and maps partitions ids to random numbers to achieve privacy. This idea is simple, practical, and elegant. However, it relies on an implicit tradeoff between privacy and efficiency. Specifically, if the partitions are larger, then less information is leaked, but the database server needs to send more false positives (i.e., data that should not have been in the results of queries) to the user. If the partitions are smaller, then the database server needs to send fewer false positives, but more information is leaked. (This issue is further explored in [23].) Furthermore, no precise quantifications are given of either of the information leak relative to the size of partitions or of the amount of communication overhead. In comparison, the solutions we provide in this paper do not have such a tradeoff; our solutions enjoy strong privacy without wasting communication resources on false positives; our security guarantee is precisely quantified using a cryptographic measure. Another issue is that, although the partition ids in [19] can be used for indexing to speed up queries, such an index can incur inference and linking attacks as is pointed out in [11]. In comparison, our solution in Section 5 speeds up queries using metadata without introducing any additional information leakage.

Agrawal et al. [4] propose a solution for range queries on numerical data that allows convenient indexing. Their solution is built on an encoding that preserves the order of the numerical data in each column. Consequently, if a database intruder observes the encrypted data, he learns the order of all cells in every column, which is a significant amount of information. They give no rigorous analysis quantifying the information leak of their solution. In comparison, in this paper we show that our solutions reveal only a small amount (as quantified in later sections) of information to a potential intruder.

In the scenario considered in [4, 24], the adversary is modeled to have access to the data storage and has no access to the transmitted messages between the users and the database. In the setting of DAS model [19], since the database server is untrusted, the server itself is a potential adversary who tries to breach the data privacy. The server has access to all encrypted data and all the transmitted messages between the users and the server. In this sense, the server has the strongest power to breach data privacy. In comparison, we model that the adversary can have access to all encrypted data in the server, and he also can monitor some transmitted messages (up to t queries) between the server and the users. We give the details of the attack (or adversary) model in Section 2.1, and we also prove the security properties of our solutions under the adversary model.

The study of "search on encrypted data" is closely related to our work. Specifically, Song, Wagner, and Perrig [28] propose practical techniques for finding keywords in encrypted files, which allow a user, when given a trapdoor for a keyword, to check the existence of the key word in a file. But their solution needs to scan the entire file sequentially and no provably secure index technique is provided. A follow-up by Chang and Mitzenmacher [9] has interesting analysis but their solution is restricted to searching for a keyword chosen from a pre-determined set. Boneh et al. present a searchable public key scheme [6]; the scenario they considered is analogous to that of [28] but uses public-key encryption rather than symmetric-key encryption. In the same scenario, Goh demonstrates a method for secure indexes using Bloom filters [15]. These solutions are possibly useful in searching for keywords in a file; however, it is unclear how to apply them to the problem of efficiently querying encrypted relational databases. Yet another piece of related work is by Feigenbaum et al. [14], in which an encryption scheme was proposed to efficiently retrieve tuples from a look-up dictionary by using hash functions. The basic idea is that a tuple can only be retrieved if a valid key is provided.

In contrast to the goals of our work, private information retrieval [10, 8, 26] is designed to hide entirely from the database which queries a user is making. As we discuss later, we take a more pragmatic view that allows more efficient solutions.

1.2 Our Contributions

In this paper, we address the problem of performing queries on an encrypted database. We consider a pragmatic notion of privacy that trades off a small amount of privacy for a gain in efficiency. Specifically, in our solutions all data is stored and processed in its encrypted form. We note that given any solution that returns a response to a query to the user consisting of precisely the encryptions in the database of the items that match the query, this solution leaks the location of the returned cells to an attacker with access to the database. Similarly, even if a solution were to return different encryptions of the matching items, if the database is able to access only those cells, then the location

of those cells is revealed. In order to admit the most efficient solutions, we therefore consider this information to be the "minimum information revelation," as described in more detail in Section 2. We allow solutions that leak this minimum information revelation, while we seek to prevent leakage of any additional information.

The contributions of this paper can be summarized as follows.

- We present a basic solution for simple queries (Section 3). We give a rigorous security analysis to show that, beyond the minimum information revelation, our solution only reveals very little information, namely which attributes are tested in the "where" condition. Our security guarantee is quantitative and cryptographically strong.
- We present a solution with enhanced security (Section 4). We show that, for a broad class of tables, this solution reveals nothing beyond the minimum information revelation.
- We present a solution that adds metadata to further speed up queries (Section 5).
- Compared with previous solutions, an advantage of our schemes is that a database user does not need to maintain a large amount of confidential information (like the partitioning ids in [19] or the large keys in [4]). In our schemes, a user only needs to store several secret keys that amount to at most tens of bytes. Thus the storage overhead on the user side is negligible.

2 Technical Preliminaries

We consider a system as illustrated in Figure 1. In this system, data is encrypted and stored in tables. In the front end, when the user has a query, the query is translated to one or more messages that are sent to the database. Upon receiving the message(s), the database finds the appropriate encrypted cells and returns them to the front end. Finally, the front end decrypts the received cells. For ease of presentation, we do not distinguish the human user from the front end program; when we say "user," we mean the human user plus the front end program.

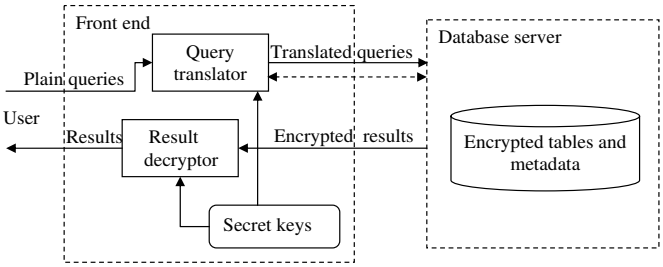

Fig. 1. Overall architecture

2.1 Trust and Attack Model

In this paper, we focus on the possibility that an intruder might successfully attack the database server. The goal of our work is that an intruder who has complete access to the database server for some time should learn very little about the data stored in the database and the queries performed on the data. Our trust and attack model is as follows:

1. We do not fully trust the database server because it may be vulnerable to intrusion. Furthermore, we assume that, once a database intruder breaks into the database, he can observe not only the encrypted data in the database, but can also control the whole database system for a time interval. During the time interval, a number of query messages sent by the user, as well as the database's processing of these queries, can be observed by the intruder. We note that assumption that an intruder can only control the whole database system for only a bounded time period is reasonable, for example, in the setting that a database administrator can physically reset the database server from time to time or when intrusions are detected.
2. We assume the communication channel between the user and the database is secure, as there exist standard protocols to secure it—e.g. SSL and IPsec. We also trust the user's front-end program; protecting the front-end program against intrusion is outside of the scope of this paper.
3. We require all data and metadata, including user logs and scheme metadata, to be stored encrypted. (Otherwise, these may open the door for intruders.)

2.2 Table and queries

We consider a database represented by a table T and we discuss queries performed on T. Suppose that T has n rows (i.e., n records) and m columns (i.e., m attributes). We denote by $T_{i,j}$ the cell at the intersection of the ith row and the jth column; we also refer to (i, j) as the *coordinates* of the cell. We denote the ith row by T_i. Each attribute of the table has a finite domain. For the jth attribute A_j, we denote the domain by D_j.

As we have mentioned, we store our tables in an encrypted form. More precisely, for a table T, we store an encrypted table T' in the database, where each $T'_{i,j}$ is an encryption of $T_{i,j}$. Without loss of generality, we assume that each cell $T_{i,j}$ of the plaintext table is a bitstring of exactly k_1 bits—that is, $\forall j \in [1, m]$, $D_j \subseteq \{0, 1\}^{k_1}$. (We can always encode any value of an attribute as a sufficiently long bitstring.) When we encrypt a cell, the encryption algorithm appends a random string of k_2 bits to the plaintext.[1] Hence, the input to the encryption algorithm is a k_0-bit string, where $k_0 = k_1 + k_2$. For simplicity (and following the practice of most symmetric encryption schemes), we assume the output of the encryption algorithm and the encryption key are k_0-bit strings as well. We therefore note that k_0 should be chosen to be long enough to resist brute-force key search attacks.

Suppose that a user intends to perform a query Q on the table T. As discussed earlier, in this paper, in order to allow solutions that are as efficient as possible, we consider query protocols that return to the user precisely the set of encrypted cells stored in the

[1] As explained in more detail in Section 3, the purpose of using a random string is that multiple occurrences of a plaintext should lead to different ciphertexts.

database that satisfy the condition of the query, with the same encryptions as in T'. We call such query protocols *precise query protocols*. Denote by $R(Q)$ the set of coordinates of the cells satisfying the condition of query Q— i.e., the cells satisfying the condition of query Q are $\{T'_{i,j} : (i,j) \in R(Q)\}$. Clearly, in *any* precise query protocol, if there is a database intrusion of the type discussed in Section 2.1, then $R(Q)$ is always revealed to the intruder. This is because the intruder can simply check T' to see which encrypted cells are in the returned result. Therefore, we say $R(Q)$ is the *minimum information revelation* of query Q. We allow solutions that reveal this minimum information revelation; we seek solutions that do not yield any additional information.

2.3 Privacy-Preserving Queries

We give a cryptographic definition of *privacy-preserving query protocols*. In particular, we consider that an intruder may observe up to t queries $Q_1, \ldots, Q_t$, where t is a polynomially bounded function of k_0. We quantify the information leaked by the protocol using a random variable α. Specifically, we say the protocol only reveals α beyond the minimum information revelation if, after these queries are processed, what the database intruder has observed can be simulated by a probabilistic polynomial-time algorithm using only α, $R(Q)$, and the encrypted table. For simplicity, we only provide here a definition of a privacy-preserving *one-round* query protocol. It is straightforward to extend this definition to multi-round query protocols.

Definition 1. *(Privacy-Preserving Query) A one-round query protocol* reveals only α beyond the minimum information revelation *if for any polynomial* $\mathsf{poly}()$ *and all sufficiently large* k_0*, there exists a probabilistic polynomial-time algorithm* $\mathcal{S}$ *(called a* simulator*) such that for any* $t < \mathsf{poly}(k_0)$*, any polynomial-size circuit family* $\{\mathcal{A}_{k_0}\}$*, any polynomial* $p()$*, and any* $Q_1, \ldots, Q_t$*,*

$$|\Pr[\mathcal{A}_{k_0}(Q_1, \ldots, Q_t, q_1, \ldots, q_t, T') = 1] - \Pr[\mathcal{A}_{k_0}(Q_1, \ldots, Q_t, \mathcal{S}(\alpha, R(Q_1), \ldots, R(Q_t), T')) = 1]| < 1/p(k_0).$$

A query protocol is ideally private *if it reveals nothing beyond the minimum information revelation.*

The above definition can be viewed as an adaptation of the definition of secure protocol in the semi-honest model (i.e., assuming the intruder does not modify the database software but attempts to violate data privacy by analyzing what he observes) [17]. However, note that a secure one-round query protocol as defined here *remains secure even in the case the intruder is fully malicious* (i.e., even when the intruder modifies the database software such that the database deviates from the protocol). The reason is that the the database's behavior does not affect the user's behavior in this case.

3 Basic Solution

In this section, we give a basic solution for queries of the format "*select . . . from* T *where* $A_j = v$," where $v \in D_j$ is a constant. We provide rigorous cryptographic specifications and proofs.

3.1 Solution Overview

Our basic idea is to encode each cell in a special redundant form. Specifically, for each cell $T_{i,j}$, the encrypted cell $T'_{i,j} = (T'_{i,j}\langle 1\rangle, T'_{i,j}\langle 2\rangle)$ has two parts. The first part $T'_{i,j}\langle 1\rangle$, is a simple encryption of $T_{i,j}$ using a block cipher $E()$; the second part, $T'_{i,j}\langle 2\rangle$, is a "checksum" that, together with the first part, enables the database to check whether this cell satisfies the condition of the query or not. $T'_{i,j}\langle 1\rangle$ and $T'_{i,j}\langle 2\rangle$ satisfy a secret equation determined by the value of $T_{i,j}$. When the database is given the equation corresponding to value v, it can easily check whether a cell satisfies the condition or not by substituting the two parts of the encrypted cell into the equation.

The remaining question is what equation to use as the secret equation. We use the following simple equation:

$$E_{f(T_{i,j})}(T'_{i,j}\langle 1\rangle) = T'_{i,j}\langle 2\rangle,$$

where f is a function. When the user has a query with condition $A_j = v$, she only needs to send $f(v)$ to the database so that the database can check, for each i, whether

$$E_{f(v)}(T'_{i,j}\langle 1\rangle) = T'_{i,j}\langle 2\rangle$$

holds. It should be infeasible to derive v from $f(v)$ because otherwise an intruder learns v when observing $f(v)$. To achieve this goal, we define $f(\cdot)$ to be an encryption of v using the block cipher $E(\cdot)$. Additional care needs to be taken when we use the block cipher E. As previously mentioned, we append a random string to $T_{i,j}$ before applying E to obtain $T'_{i,j}\langle 1\rangle$; this is done in order to prevent the database from being able to determine whether two cells have the same contents. Additionally, in order to avoid having the same $f(v)$ for different attributes, we append j to $f(v)$ before applying E.

3.2 Solution Details

Data Format. Let $E(\cdot)$ be a symmetric encryption algorithm whose key space, plaintext space, and ciphertext space are all $\{0,1\}^{k_0}$. We often use the notation $E_S(M_1, M_2)$ to denote a message (M_1, M_2) encrypted using secret key S, where M_1 (resp., M_2) is either a k_1-bit (resp., k_2-bit) string. We denote the corresponding decryption algorithm by D, and we assume that the key generation algorithm simply picks a uniformly random key from the key space $\{0,1\}^{k_0}$.

To create the table T in the database, the user first picks two secret keys s_1, s_2 from $\{0,1\}^{k_0}$ independently and uniformly. The user keeps s_1, s_2 secret. For each cell $T_{i,j}$, the user picks $r_{i,j}$ from $\{0,1\}^{k_2}$ uniformly at random and stores

$$\begin{aligned} T'(i,j) &\stackrel{\triangle}{=} (T'(i,j)\langle 1\rangle, T'(i,j)\langle 2\rangle) \\ &= (E_{s_1}(T_{i,j}, r_{i,j}), E_{E_{s_2}(T_{i,j},j)}(E_{s_1}(T_{i,j}, r_{i,j}))) \end{aligned}$$

Query Protocol. Denote by A_j the jth attribute of T. Suppose there is a query *select* $A_{j_1}, \ldots, A_{j_\ell}$ *from* T *where* $A_{j_0} = v$. To carry out this query, the user computes $q = E_{s_2}(v, j_0)$ and sends j_0, q, and $(j_1, \ldots, j_\ell)$ to the database.

For $i = 1, \ldots, n$, the database tests whether $T'_{i,j_0}\langle 2\rangle = E_q(T'_{i,j_0}\langle 1\rangle)$ holds. For any i such that the above equation holds, the database returns $T'_{i,j_1}\langle 1\rangle, \ldots, T'_{i,j_\ell}\langle 1\rangle$ to the user. The user decrypts each received cell using secret key s_1 and discards the k_2-bit tail of the cleartext.

In our scheme, note that each encrypted cell with the same plaintext value has a different encryption. Thus if an intruder breaks into the database and sees the encrypted table, he cannot tell whether two cells have the same plaintext value or not.

3.3 Security Analysis

We can prove the security of our scheme by using standard cryptographic techniques. Recall that for security we need to consider t queries. Suppose the uth query ($1 \leq u \leq t$) is of the format "select $A_{j_{u,1}}$,..., $A_{j_{u,\ell}}$ from T where $A_{j_{u,0}} = v_u$." We show that our basic solution only reveals $j_{1,0}, \ldots, j_{t,0}$ beyond the minimum information revelation. That is, the only extra information leakage by the basic solution is which attributes are tested in the "where" conditions.

The security of our scheme derives from the security of the block cipher we use. In cryptography, secure block ciphers are modeled as *pseudorandom permutations* [18]. Here, encryption key of the block cipher is the random seed for the pseudorandom permutation. For each value of the key, the mapping from the cleartext blocks to the ciphertext blocks is the permutation indexed by the value of the seed. In the following theorem, we assume the block cipher we use satisfies this security requirement.

Theorem 1. *If the block cipher E is a pseudorandom permutation (with the encryption key as the random seed), the basic protocol reveals only $j_{1,0}, \ldots, j_{t,0}$ beyond the minimum information revelation.*

Proof. We construct a simulator $\mathcal{S}$ as follows. First, let $R_1(Q_u) = \{i : (i,j) \in R(Q_u)\}$ and $R_2(Q_u) = \{j : (i,j) \in R(Q_u)\}$. Then, for any $u \in \{1, \cdots, t\}$, if there exists $u' < u$ such that $j_{u,0} = j_{u',0}$ and that $R_1(Q_u) = R_1(Q_{u'})$, $\mathcal{S}$ sets $\overline{q}_u = \overline{q}_{u'}$. otherwise, $\mathcal{S}$ chooses $\overline{q}_u$ from $\{0,1\}^{k_0} - \{\overline{q}_{u'} : u' < u \wedge j_{u,0} = j_{u',0}\}$ uniformly at random. Next, for $i = 1$ through n and $j = 1$ through m, $\mathcal{S}$ chooses $\overline{T'}_{i,j}\langle 1\rangle$ uniformly and independently from $\{0,1\}^{k_0}$. For $u = 1$ through t, for each $i \in R_1(Q_u)$, $\mathcal{S}$ computes

$$\overline{T'}_{i,j_{u,0}}\langle 2\rangle = E_{\overline{q}_u}(\overline{T'}_{i,j_{u,0}}\langle 1\rangle).$$

For any pair (i,j) for which $\overline{T'}_{i,j}\langle 2\rangle$ has not been defined, $\mathcal{S}$ chooses $\overline{T'}_{i,j}\langle 2\rangle$ from $\{0,1\}^{k_0}$ uniformly and independently. Finally, $\mathcal{S}$ outputs $\overline{q}_1, \ldots, \overline{q}_t$, $\overline{T'}$. The indistinguishability by polynomial-size algorithms follows from the pseudorandomness of E.

In this setting, even if the intruder has access to the whole database, the intruder can learn nothing about the encrypted data. By combining $j_{1,0}, \cdots, j_{t,o}$ with the minimum information revelation, an intruder can derive some statistical information about the underlying data or the queries (Theorem 1 does catch this case). In Section 4, we present a solution that leaks less information to make such attacks more difficult.

3.4 Performance Evaluations

To evaluate the efficiency of our basic solution in practice, we implemented the basic solution. Our experiments use the Nursery dataset from the UCI machine learning repository [5]. The Nursery dataset is a table with eight categorical attributes and one class attribute. There are 12,960 records in total. The total number of data cells is about $100,000$. The only change we made to the Nursery dataset is that we added an ID attribute to the Nursery dataset so that the table would have a primary key.

Because the time spent on communication is highly dependent on the network bandwidth, we focus on the computational overhead and ignore the communication overhead. The experimental environment is the NetBSD operating system running on an AMD Athlon 2GHz processors with 512M memory. For the block cipher, we use the Blowfish symmetric encryption algorithm with a 64-bit block size (i.e., $k_0 = 64$).

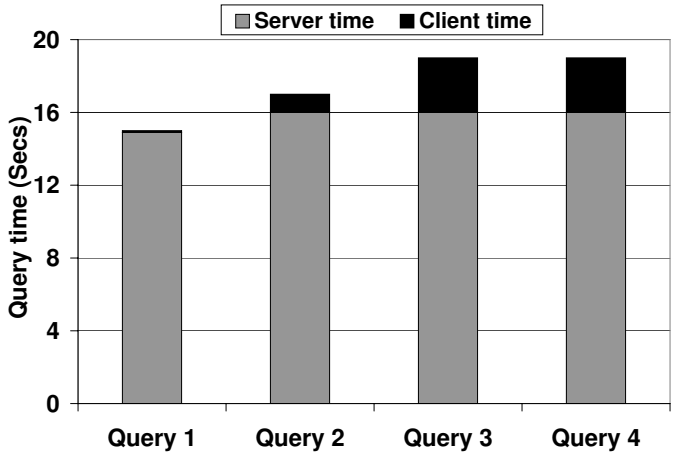

Fig. 2. Query time in the basic solution

Clearly, the overhead to encrypt a database is linear in the size of the database. Specifically, in our experiments it took only 25 seconds to encrypt the entire Nursery dataset. On average, encrypting a single record requires only 0.25 milliseconds. Figure 2 shows the time consumed by four SELECT queries. Those queries are SELECT $*$ FROM Nursery WHERE Parent=usual, WHERE Class=recommend, WHERE Class=very_recom and WHERE Class=priority.

For each query, the database server needs almost the same amount of time for computation (about 16 seconds). The user's computational time depends on the number of returned records from the database. In the first query, only 2 records are returned, and so the computational time by the user is extremely small. In contrast, the last two queries return 4320 and 3266 records, respectively. Therefore, the computational time of the user in each of these two queries is about 4 seconds.

4 Solution with Enhanced Security

In this section, we enhance the security of the basic solution so that the query protocol reveals less information. For a broad class of tables, we can show our solution with enhanced security is ideally private.

4.1 Solution Overview

Recall that the basic solution reveals which attributes are tested in the "where" conditions. Our goal is to hide this information (or at least part of this information). A straightforward way for doing this is to randomly permute the attributes in the encrypted table, in order to make it difficult for a database intruder to determine which attributes are tested.

There remains the question of which distribution to use for the random permutation. If the distribution has a large probability mass on some specific permutations, then the intruder can guess the permutation with a good probability. So the ideal distribution is the uniform distribution. However, if the permutation is chosen uniformly from all permutations of the attributes, the user needs to "memorize" where each attribute is after the permutation. When the number of attributes is large, this is a heavy burden for the user. To eliminate this problem, we use a pseudorandom permutation, which is by definition indistinguishable from a uniformly random permutation [16]. The advantage of this approach is that it requires the user to memorize the random seed.

In fact, we note that we do not need to permute all the attributes in the encrypted table. For each (i, j), we can keep $T'_{i,j}\langle 1\rangle$ as defined in the basic solution; we only need to permute the equations satisfied by $T'_{i,j}\langle 1\rangle$ and $T'_{i,j}\langle 2\rangle$ because only these equations are tested when there is a query. Specifically, the equation satisfied by $T'_{i,j}\langle 1\rangle$ and $T'_{i,j}\langle 2\rangle$ is no longer decided by the value $T_{i,j}$; instead, it is decided by $T_{i,\pi_S(j)}$, where $\pi_S()$ is a pseudorandom permutation. Consequently, when there is a query whose condition involves attribute A_j, the database actually tests an equation on attribute $A_{\pi_S^{-1}(j)}$.

4.2 Solution Details

Data Format. Let $\pi_S()$ be a pseudorandom permutation on $\{1, \cdots, m\}$ for a uniformly random seed $S \in \{0,1\}^{k_0}$. To store the table T in the database, the user first picks secret keys s_1, s_2, s'_2 from $\{0,1\}^{k_0}$ independently and uniformly. The user keeps s_1, s_2, and s'_2 secret. For each cell $T_{i,j}$, the user picks $r_{i,j}$ from $\{0,1\}^{k_2}$ uniformly at random, computes $\hat{j} = \pi_{s'_2}(j)$ and stores

$$\begin{aligned} T'(i,j) &\stackrel{\triangle}{=} (T'(i,j)\langle 1\rangle, T'(i,j)\langle 2\rangle) \\ &= (E_{s_1}(T_{i,j}, r_{i,j}), E_{E_{s_2}(T_{i,\hat{j}},j)}(E_{s_1}(T_{i,j}, r_{i,j}))), \end{aligned}$$

Query Protocol. Suppose there is a query *select* A_{j_1},...,A_{j_ℓ} *from* T *where* $A_{j_0} = v$. To carry out this query, the user computes $j'_0 = \pi^{-1}_{s'_2}(j_0)$ and $q = E_{s_2}(v, j'_0)$, then sends $j'_0, q, (j_1, \ldots, j_\ell)$ to the database.

For $i = 1, \ldots, n$, the database tests whether $T'_{i,j'_0}\langle 2\rangle = E_q(T'_{i,j'_0}\langle 1\rangle)$ holds. For any i such that the above equation holds, the database returns $T'_{i,j_1}\langle 1\rangle, \ldots, T'_{i,j_\ell}\langle 1\rangle$ to the user. The user decrypts each received cell using secret key s_1 and discards the k_2-bit tail of the cleartext.

4.3 Security Analysis

Again, recall that for security we need to consider t queries, where the uth query ($1 \leq u \leq t$) is of the format "select $A_{j_{u,1}}, \ldots, A_{j_{u,\ell}}$ from T where $A_{j_{u,0}} = v_u$." We introduce a new variable that represents whether two queries involve testing the same attribute: for $u, u' \in [1, t]$, we define

$$\epsilon_{u,u'} = \begin{cases} 1 & \text{if } j_{u,0} = j_{u',0} \\ 0 & \text{otherwise.} \end{cases}$$

Using this new variable, we can quantify the security guarantee of our solution with enhanced security. Furthermore, we are able to show that our solution with enhanced security becomes ideally private if the table belongs to a broad class, which we call the *non-coinciding* tables. Intuitively, a table is non-coinciding if any two queries that test different attributes do not have exactly the same result. More formally, we have the following.

Definition 2. *A table T is* non-coinciding *if for any $j \neq j'$, any $v \in A_j$, $v' \in A_{j'}$,*

$$\{i : T_{i,j} = v\} \neq \{i : T_{i,j'} = v'\}.$$

Theorem 2. *Suppose that the block cipher $E()$ is a pseudorandom permutation (with the encryption key as the random seed). Then the query protocol with enhanced security reveals only $\epsilon_{u,u'}$ for $u, u' \in [1, j]$. When the table T is non-coinciding, the query protocol with enhanced security is ideally private.*

Proof. We construct a simulator $\mathcal{S}$ as follows. First, recall that $R_1(Q_u) = \{i : (i, j) \in R(Q_u)\}$ and $R_2(Q_u) = \{j : (i, j) \in R(Q_u)\}$. For $u = 1$ through t, if there exists $u' < u$ such that $\epsilon_{u,u'} = 1$ and that $R_1(Q_u) = R_1(Q_{u'})$, $\mathcal{S}$ sets $\overline{q}_u = \overline{q}_{u'}$ and $j'_{u,0} = j'_{u',0}$; if there exists $u' < u$ such that $\epsilon_{u,u'} = 1$ and that $R_1(Q_u) \neq R_1(Q_{u'})$, $\mathcal{S}$ chooses $\overline{q}_u$ from $\{0, 1\}^{k_0} - \{\overline{q}_{u'} : u' < u \wedge \epsilon_{u,u'} = 1\}$ uniformly at random and sets $j'_{u,0} = j'_{u',0}$; otherwise, $\mathcal{S}$ chooses $\overline{q}_u$ from $\{0, 1\}^{k_0}$ uniformly at random, and $j'_{u,0}$ from $[1, m] - \{j'_{u',0} : u' < u\}$ uniformly at random. Next, for $i = 1$ through n and $j = 1$ through m, $\mathcal{S}$ chooses $\overline{T'}_{i,j}\langle 1\rangle$ uniformly and independently from $\{0, 1\}^{k_0}$. For $u = 1$ through t, for each $i \in R_1(Q_u)$, $\mathcal{S}$ computes

$$\overline{T'}_{i,j'_{u,0}}\langle 2\rangle = E_{\overline{q}_u}(\overline{T'}_{i,j'_{u,0}}\langle 1\rangle).$$

For each pair (i, j) such that $\overline{T'}_{i,j}\langle 2\rangle$ has not been defined, $\mathcal{S}$ chooses $\overline{T'}_{i,j}\langle 2\rangle$ from $\{0, 1\}^{k_0}$ uniformly and independently. Finally, $\mathcal{S}$ outputs $\overline{q}_1, \ldots, \overline{q}_t$, $\overline{T'}$. The indistinguishability by polynomial-size circuits follows from the pseudorandomness of $E()$.

When T is non-coinciding, in the above proof we can replace $\epsilon_{u,u'} = 1$ with $R_1(Q_u) = R_1(Q_{u'})$. Because these two conditions are equivalent, this replacement does not change the output of the simulator. On the other hand, because $\epsilon_{u,u'}$ is no longer needed by the simulator, we have shown the protocol is ideally private.

5 Query Speedup Using Metadata

In the two solutions we have presented, performing a query on the encrypted table requires testing each row of the table. Clearly, this is very inefficient in large-size databases. In this section, we consider a modification to the basic solution that drastically speeds up queries. It is easy to make a similar modification to the solution with enhanced security.

5.1 Solution Overview

We can significantly improve the efficiency if we are able to replace the sequential search in the basic solution with a binary search. However, our basic solution finds the appropriate rows by testing an equation, while a binary search cannot be used to find the items that satisfy an equation.

To sidestep this difficulty, we add some metadata to eliminate the need for testing an equation[2]. Specifically, for each cell in the column, we add a tag and a link. The tag is decided by the value of the cell; the link points to the cell. We sort the metadata according to the order of the tags. When there is a query on the attribute, the user sends the appropriate tag to the database so that the database can perform a binary search on the tags.

We illustrate the concept with a simple example, shown in Figure 3. Consider a column of four cells with values 977, 204, 403, 155. We compute four tag values based on the corresponding cell values. Suppose that the tags we get are 3, 7, 4, 8. We sort the tags to get a list: 3, 4, 7, 8. After we add links to the corresponding cells, we finally get: (3, link to cell "977"), (4, link to cell "403"), (7, link to cell "204"), (8, link to cell "155").

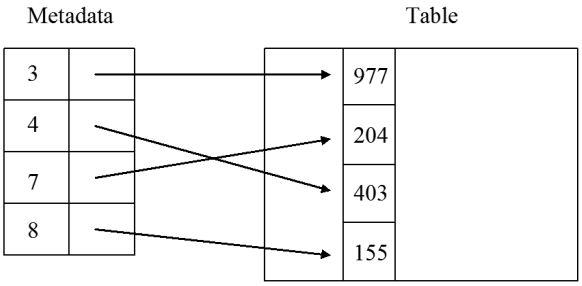

Fig. 3. Example of metadata

Nevertheless, there is a question of multiple occurrences of a single value: if multiple cells in the column have the same value, how do we choose the tags for these cells?

[2] The functionality of these metadata is analogous to indices in traditional database systems—to help speed up queries. However, since the structure and usage of these metadata are different from that of traditional indices (e.g., B+-trees), we do not call them indices. Note that indices like B+-trees cannot be used in our scenario.

Clearly, we cannot use the same tag for these cells; otherwise, when the database intruder looks at the tags, he can recognize cells with the same value. In fact, it should be hard for the intruder to find out which tags correspond to the same cell value. On the other hand, it should be easy for the user to derive the entire set of tags corresponding to a single value.

We resolve this dilemma using a two-step mapping. In the first step, we map a cell value $T_{i,j}$ to an intermediate value $H_{i,j}$ whose range is much larger. Then the $H_{i,j}$'s are sparse in their range, which means around each value of $H_{i,j}$ there is typically a large "blank" space. Consequently, to represent multiple occurrences of the same cell value $T_{i,j}$, we can use multiple points starting from $H_{i,j}$. In the second step, we map these intermediate points to tags such that the continuous intermediate points become random-looking tags. See Figure 4 for an illustration. In our design, the first step of

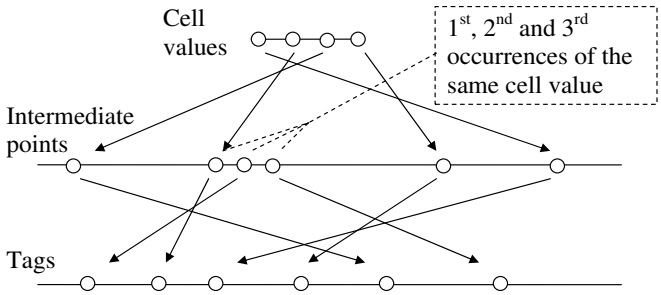

Fig. 4. Two-step mapping from cell values to tags

mapping (from the cell value to the intermediate value) is implemented using an encryption of the cell value (appended with a k_2-bit 0 so that the input to the cipher is k_0 bits), where the encryption key is kept by the user. The second step of mapping (from the intermediate value to the tag) is implemented using another encryption, where the key is again kept by the user. Since the database intruder does not know the two encryption keys, he cannot figure out which cell value corresponds to which tag, or which of the tags correspond to the same cell value. On the other hand, when there is a query, the user can simply send the database the tags for the cell value in the query; then the database can easily locate the rows satisfying the condition of this query.

Note that, for the convenience of queries, we should keep a counter of the occurrences of each cell value; otherwise, when the user has a query, he cannot know how many intermediate values (and thus how many tags) he should compute. Clearly such counters should be encrypted and stored in the database, where the encryption key is kept by the user. Each encrypted counter should be kept together with the corresponding intermediate value (of the first occurrence of the cell value), so that it can be identified by the user. When the database intruder observes encrypted metadata, he does not know which cell value corresponds to which intermediate value and therefore does not know which cell value corresponds to the encrypted counter.

5.2 Solution Details

Metadata Format. To speed up queries on attribute A_j, the user picks keys $s_3, s_4, s_5 \in \{0,1\}^{k_0}$ independently and uniformly. For $i = 1, \ldots, n$, the user computes

$$H_{i,j} = E_{s_3}(T_{i,j}, 0).$$

For each value of each attribute, the user keeps a counter of the number of occurrences. If this is the $c_{i,j}$th occurrence of the value $T_{i,j}$ in the attribute A_j, the user computes

$$I_{i,j} = E_{s_4}((H_{i,j} + c_{i,j}) \bmod 2^{k_0}).$$

When $I_{i,j}$'s have been computed for all i's, suppose the final value of the counter is $c_j(v)$ for each value v. Then the user encrypts $c_j(v)$ using secret key k_5:

$$C_j(v) = E_{k_5}(c_j(v), 0).$$

The user stores $L = \{(I_{i,j}, \text{link to row } T'_i)\}_{i \in [1,n]}$ and

$$\begin{aligned} B &\triangleq \{(B_x\langle 1\rangle, B_x\langle 2\rangle)\}_{x \in [1,\ |\{T_{i,j}: i \in [1,n]\}|]} \\ &= \{(E_{s_3}(v,0), C_j(v))\}_{v \in \{T_{i,j}: i \in [1,n]\}} \end{aligned}$$

in the database as metadata for queries on attribute A_j. Note that L should be sorted in an increasing order of $I_{i,j}$. The user keeps s_3, s_4, and s_5 secret.

Query Protocol. Now suppose there is a query *select* $A_{j_1}, \ldots, A_{j_\ell}$ *from* T *where* $A_j = v$. To carry out this query, the user first computes $h = E_{s_3}(v, 0)$ and sends h to the database. The database finds x such that $B_x\langle 1\rangle = h$ and sends the corresponding $C = B_x\langle 2\rangle$ back to the user. The user then decrypts C (and discards the k_2-bit tail) to get $c_j(v)$, the overall number of occurrences of v. For $c = 1, \ldots, c_j(v)$, the user computes

$$I_c = E_{s_4}((h + c) \bmod 2^{k_0}),$$

and sends I_c to the database. Since L is sorted in the increasing order of I_c, the database can easily locate I_c and find the link corresponding to I_c. For each row T'_i pointed by these links, the database sends the encrypted cells $T'_{i,j_1}\langle 1\rangle, \ldots, T'_{i,j_\ell}\langle 1\rangle$ to the user. Finally, the user decrypts each received cell using secret key s_1 and discards the k_2-bit tail of the cleartext.

5.3 Performance Evaluation

To evaluate the speedup of our solution, we measured the query time on the same dataset used for testing the basic solution. Figure 5 compares the metadata generation time for four different attributes: ClassLabel, Finance, Parents, and ID. The metadata generation time depends on not only the number of rows in the table, but also the domain size of the attribute (more precisely, the number of the different values that actually appear in the attribute). In the attributes we experimented with, ClassLabel, Finance, and Parents have small domain sizes; the metadata generation time for each of them is about 6

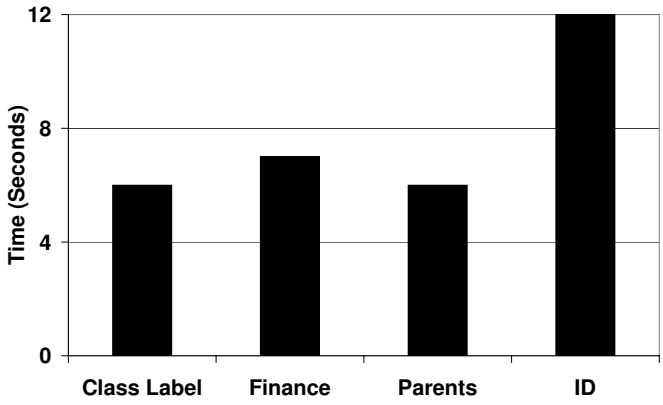

Fig. 5. Computational time to generate metadata

seconds. In contrast, generating metadata on ID attribute needs about twice as much time because is the ID attribute has a large domain.

Figure 6 compares the query time of the basic solution and that of the solution with metadata. with the following four queries that are to select all records where Class=recommend, where Class=very_recom, where Parent=usual, and where ID=1000. The results of the first and the fourth queries have only 2 and 1 record, respectively. For such queries, the solution with metadata is so fast that the query time can hardly be seen in the figure. The other two queries have more records in their results: the second query has 328 records in its result and our solution with metadata saves about 94% of the query time; the third query has 4320 records in its result and our solution with metadata saves about 79% of the query time. Clearly, the trend is that the solution with metadata gains more in efficiency if there are fewer records in the query result. However, even for a query with a large number of records in the result, the solution with metadata is much faster than the basic solution.

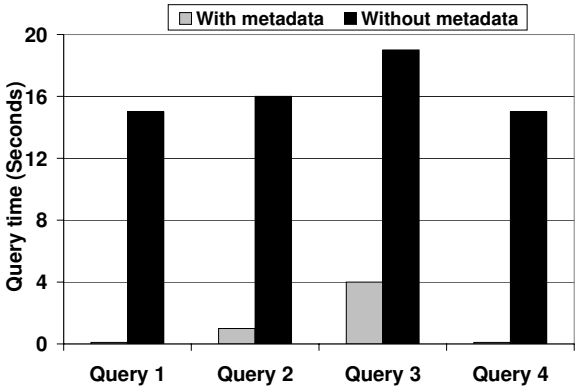

Fig. 6. Query time: solution with metadata vs. basic solution

6 Conclusions

In this paper, we have investigated privacy-preserving queries on encrypted data. In particular, we present privacy-preserving protocols for certain types of queries. Although the supported queries are limited, our main goal in this paper is to provide rigorous, quantitative (and cryptographically strong) security.

We note that, in general, it is difficult to evaluate the correctness and security of a new design if no quantitative analysis of information leakage is given. It is therefore beneficial to introduce quantitative measures of privacy such as those we have introduced. It is our hope that these measures may be useful elsewhere.

For many practical database applications, more complex queries than those considered in this paper must be supported. A future research topic is to extend the work in this paper to allow more complex queries. Ideally, the extension should maintain strong, quantifiable security while achieving efficiency for complex queries.

References

1. Oracle Corporation. Database Encryption in Oracle9i, 2001.
2. IBM Data Encryption for IMS and DB2 Databases, Version 1.1, 2003.
3. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, 2002.
4. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *SIGMOD*, 2004.
5. C. Blake and C. Merz. UCI repository, 1998.
6. D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *EUROCRYPT*, 2004.
7. L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *VLDB*, 2002.
8. C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, 1999.
9. Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive:2004/051, available at `http://eprint.iacr.org/2004/051.pdf`.
10. Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.
11. Ernesto Damiani, S. De Capitani Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS*, 2003.
12. Eric Dash. Lost credit data improperly kept, company admits. *New York Times*, June 20 2005.
13. G. I. Davida, D. L. Wells, and J. B. Kam. A database encryption system with subkeys. *ACM TODS*, 6(2):312–328, 1981.
14. J. Feigenbaum, M. Y. Liberman, and R. N. Wright. Cryptographic protection of databases and software. In *DIMACS Workshop on Distributed Computing and Cryptography*, 1990.
15. E. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216. `http://eprint.iacr.org/2003/216/`.
16. O. Goldreich. *Foundations of Cryptography*, volume 1. Cambridge University Press, 2001.
17. O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

18. S. Goldwasser and M. Bellare. Lecture notes on cryptography. Summer Course Lecture Notes at MIT, 1999.
19. Hakan Hacigumus, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.
20. Hakan Hacigumus, Balakrishna R. Iyer, and Sharad Mehrotra. Providing database as a service. In *ICDE*, 2002.
21. Hakan Hacigumus, Balakrishna R. Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*, 2004.
22. J. He and J. Wang. Cryptography and relational database management systems. In *Int. Database Engineering and Application Symposium*, 2001.
23. Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *VLDB*, 2004.
24. Bala Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, and Y. Wu. A framework for efficient storage security in RDBMS. In *EDBT*, 2004.
25. J. Karlsson. Using encryption for secure data storage in mobile database systems. Friedrich-Schiller-Universitat Jena, 2002.
26. E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database computationally-private information retrieval. In *FOCS*, 1997.
27. Gultekin Ozsoyoglu, David Singer, and Sun Chung. Anti-tamper databases: Querying encrypted databases. In *Proc. of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, 2003.
28. Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.
29. David Stout. Veterans chief voices anger on data theft. *New York Times*, May 25 2006.
30. R. Vingralek. A small-footprint, secure database system. In *VLDB*, 2002.