

# Anti-Tamper Databases: Querying Encrypted Databases

Gultekin Ozsoyoglu<sup>#</sup>, David A. Singer<sup>+</sup>, Sun S. Chung<sup>#</sup>

<sup>#</sup>EECS Department, <sup>+</sup>Math Department  
Case Western Reserve University, Cleveland, OH 44106

(gxo3, das5, ssc7)*@po.cwru.edu*

## Abstract

With mobile computing and powerful laptops, databases with sensitive data can be physically retrieved by malicious users who can employ techniques that were not previously thought of, such as disk scans, compromising the data by bypassing the database management system software or database user authentication processes. Or, when databases are provided as a service, the service providers may not be trustworthy.

A way to prevent, delay, limit, or contain the compromise of the protected data in a database is to encrypt the data and the database schema, and yet allow queries and transactions over the encrypted data. Clearly, there is a compromise between the degree of security provided by encryption and the efficient querying of the database. In this paper, we investigate the capabilities and limitations of encrypting the database in relational databases, and yet allowing, to the extent possible, efficient SQL querying of the encrypted database.

We concentrate on integer-valued attributes, and investigate a family of open-form and closed-form homomorphism encryption/decryption functions, the associated query transformation problems, inference control issues, and how to handle overflow and precision errors.

## 1 Introduction

With the changes in present-day computing environments, databases with sensitive data can be compromised in new ways. Mobile computing and powerful laptops allow databases with sensitive data to travel everywhere. Laptops can be physically retrieved by malicious users who can employ techniques that were not previously thought of, such as disk scans, compromising the data by bypassing the database management system software or database user authentication processes. Or, when databases are provided as a service [HIM02, HILM02], the owner of the data and the database service provider who maintains the data may be different; the service provider may not be trustworthy, and the data needs to be protected from the database service provider. These examples illustrate the need to

- Encrypt data in a database since compromise will occur with the traditional ways of securing databases such as access controls [WJ01, RG00], multilevel secure database architectures [JAKMM01, QL97, MLS], database integrity lock architectures [MLS], or statistical database inference controls [AW89].
- Allow, as much as possible, standard DBMS functionality, such as querying and query optimization, over encrypted data.

In this paper, we consider a database environment where (a) the database contains sensitive data available only to its legitimate users, and (b) the database can be captured by the adversary physically, and a compromise threat exists (by physically accessing the data, and/or using a priori knowledge about the data in the database, and/or breaking user passwords, and accessing the database). We believe that a good way to prevent, delay, or contain the compromise of the protected data in a database is to encrypt the database, and yet allow queries over the encrypted data.

This paper considers attribute (field)-level encryption for relational databases. We give an example.

**Example 1.** Consider a relational employee database with the relation EMPLOYEE (Id, Salary) where Id and Salary are integer-valued employee id and employee salary attributes. Assume that  $f()$  and  $g()$ , functions from integers to integers with inverses  $f^{-1}()$  and  $g^{-1}()$  respectively, are used to encrypt employee salaries and employee ids, and the relation EMPLOYEE and the attribute names Id and Salary are encoded as the characters R, A, and B. Then the SQL query Q(DB) over the database DB

Q(DB):       SELECT EMPLOYEE.Id FROM EMPLOYEE WHERE Salary=a

returns the employees with salary a. Q(DB) is rewritten into the SQL query  $Q_E(DB_E)$  over the encrypted database  $DB_E$  as follows.

$Q_E(DB_E)$ : SELECT R.A FROM R WHERE B=f(a)

Then,  $Q_E(DB_E)$  is submitted to the DBMS, to be executed against the database  $DB_E$ . Assume that the output  $O(Q_E(DB_E))$  of the query  $Q_E(DB_E)$  is a single tuple with value  $y$ . Then the value  $y$  is decoded as  $g^{-1}(y)$ , and returned as the output  $O(DB)$  of the query  $Q(DB)$ .

We use the notion of key from cryptology for both functions  $f()$  and  $g()$  as follows: The function, say,  $f(x)$  and its inverse  $f^{-1}(x)$  are in the form  $E(K, x)$  and  $D(K, x)$  where  $K$  is the secret key, and  $D(K, E(K, x)) = x$ . That is,  $D$  decrypts  $x$  encrypted by  $E$  using the same key  $K$ .

We refer to the process of transforming the SQL query  $Q(DB)$  of the original database into the SQL query  $Q_E(DB_E)$  of the encrypted database as the *query rewriting process*.

Clearly, the requirement of high level of security for the encrypted data conflicts with the requirements of query processing and query optimization over encrypted data. We believe that, as long as the database performance degradation due to encryption is under control, users would choose to use encrypted databases over nonencrypted databases, even when the provided degree of security is not high. In this paper, we investigate techniques for encrypting a database, and the accompanying techniques to query the encrypted database, which we refer to as the *anti-tamper database*, and explore the tradeoffs among (a) the security of the database, and (b) the query expressive power and query processing. The site (computer) in which the anti-tamper database resides is called as the *anti-tamper site*.

Consider an integer attribute  $V$  with domain  $DOM(V)$ , and the encryption function  $f()$  for  $V$  values. We investigate closed-/open-form encryption that employs an *integer-to-integer* (e.g.,  $f: Int \rightarrow Int$ ) or *integer-to-vector of integers* (e.g.,  $f: Int \rightarrow Vector(Int)$ ) encryption  $f()$ . Clearly, query rewriting and query processing with an integer-to-integer transformation is simpler, and more efficient.

Consider a database  $DB$ , and the set  $S$  of SQL queries  $Q$  on  $DB$  that correspond to the set of safe relational calculus [Ullm89] queries<sup>1</sup>. Let  $DB_E$  be the encrypted database  $DB$  (i.e.,  $f(x)=y$ , for all  $x \in DB$ ,  $y \in DB_E$ ) with the transformed set  $S_E$  of SQL queries  $Q_E(DB_E)$  on the database  $DB_E$ . For encrypting the database  $DB$  using the encryption function  $f()$ , we choose an encryption function  $f()$  that is a *group homomorphism from  $DB$  onto  $DB_E$  with respect to  $S$* , i.e., for any query  $Q$  in  $S$ , we have  $Q(DB) = f^{-1}(Q_E(f(DB)))$  (with a slight abuse of notation). This means that (a) SQL queries in  $S$  can be completely evaluated solely using a single query  $Q_E$  on the anti-tamper database  $DB_E$ , and, (b) with the exception of decrypting the output of  $Q_E(DB_E)$  (possibly, at another “secure” site), there is no extra query processing burden on evaluating  $Q_E(DB_E)$ .

In this paper, for encrypting relational databases, we evaluate a family of homomorphism encryption functions  $f()$ , the associated query processing issues, and inference control under a priori knowledge.

We make the following assumptions about the anti-tamper database:

- The encrypting of the original database is transparent and unknown to the legitimate users of the database. That is, there is no extra query specification/transformation burden placed on the legitimate users of the anti-tamper database.
- In compliance with privacy laws, we assume that the general encryption mechanism is made available to the public, including the adversary. However, certain parameters of the encryption mechanism (e.g., private keys) are kept secret so as to make the protected information in the database secure.
- We assume that the commercial DB system that hosts the data does not know that the data in its database is encrypted. This assumption is for convenience in that commercial DBMSs are complex proprietary software systems, and any security technique that is deployable without modifying the DBMS is more desirable over ones that require changes to the DBMS.

---

<sup>1</sup> Thus, a query  $Q$  in  $S$  has a *Where* clause with conjunctions and/or disjunctions of predicates  $p$ , and existential and universal quantifiers. The predicate  $p$  either (a) specifies that a variable does/does not range over an attribute of a relation, or (b) specifies “ $x\theta z$ ” where  $x$  is a variable,  $z$  is a variable or a constant, and  $\theta \in \{<, >, =, \neq, \leq, \geq\}$ .  $Q$  has a finite output and finite evaluation time.

This paper is a first step towards investigating the feasibility of encrypting (object-)relational databases, and yet allowing SQL access over the encrypted data. The rest of the paper is organized as follows. In Section 2, we introduce the computing architecture used in the rest of the paper. Section 3 introduces order- and distance-preserving open-form encryption/decryption algorithms, a class of SQL queries for which a very simple transformation to the encrypted database exists, and inference control issues. In Section 4, we discuss order-preserving closed-form encryption and decryption, the associated overflow and precision errors, and an algorithm that generates an encryption/decryption function sequence. Section 5 concludes.

## 2 Computing Architecture

We employ the following *computing architecture* for our environment. Let the original database DB be encrypted into the *encrypted (anti-tamper) database*  $DB_E$ . Now, if  $DB_E$  directly becomes available to the adversary, its contents are devoid of semantics and, therefore not directly usable by the adversary. We employ an intermediary software agent, called the (Encryption/Decryption) *Agent*, which we assume to be secure. That is, the adversary cannot capture the Agent software code, and reverse-engineer its encryption/decryption algorithms. We consider two alternative architectures:

- The agent resides at a site different than the site of the anti-tamper database  $DB_E$ , and has significant computational power and storage space. We refer to this site as a *secure site*, and to the Agent as a *secure-site Agent*. There may also be a secure DBMS at the secure site (to be employed in our procedures). We refer to this DBMS as  $DBMS_{Agent}$ .
- The agent resides at the site of the anti-tamper database, and has little computational power. We refer to this Agent as the *anti-tamper-site Agent*.

For both architectures, user queries are processed as follows:

- a. The user forms a query  $Q(DB)$  against the original database DB, and submits it to the Agent.
- b. The Agent rewrites the original query  $Q(DB)$  into either
  - i. A single query  $Q_E(DB_E)$  against the encrypted database  $DB_E$  if possible, or
  - ii. A set  $\{Q_{E_i}(DB_E) | 1 \leq i \leq k\}$  of  $k$  different,  $k > 1$ , queries,
 and submits  $Q_E(DB_E)$  or the query set  $\{Q_{E_i}(DB_E)\}$ , respectively, to the DBMS of the anti-tamper database, referred to as  $DBMS_E$ .
- c. The  $DBMS_E$  processes the query  $Q_E(DB_E)$  or the query set  $\{Q_{E_i}(DB_E)\}$ , and returns the output  $O(Q_E(DB_E))$  or the output set  $\{O(Q_{E_i}(DB_E))\}$ , respectively, to the Agent.
- d. In the case of a single output  $O(Q_E(DB_E))$ , the Agent decrypts  $O(Q_E(DB_E))$  into  $O(Q(DB))$ , the legitimate output of the original query  $Q$  against the original database DB, and returns it to the user. In the case of the multiple output set  $\{O(Q_{E_i}(DB_E))\}$ , the Agent decrypts each output  $O(Q_{E_i}(DB_E))$ , performs, if necessary, additional computations with the decrypted output set to obtain and return the answer  $O(Q(DB))$  to the user.

Note that the *user site* in this architecture can be the secure site, the anti-tamper site, or yet a third (secure) site. If the user site is the third site, we assume that there is a secure communication channel between the Agent site and the user site. If the user and the Agent are both at the anti-tamper site then the decrypted output of a query returned to the user can be compromised (until it is destroyed by the user) if captured by the adversary.

The approach of Example 1 to query processing can be characterized as “*no decrypting at the anti-tamper site*”, which is clearly the desirable choice. For feasibility considerations, there is another alternative, referred to here as *anti-tamper-site decryption*, which uses key-based decryption and user-defined procedures [HS93, H94] at the anti-tamper site, illustrated with an example.

**Example 2.** Assume that the EMPLOYEE relation has the integer-valued attribute ProjId, which specifies the current project that the employee is working on. Attribute ProjId is encrypted using the encryption function  $E(K, x)$  where  $x$  takes its value from ProjId attribute values, and the key  $K$  at the Agent is such that  $D(K, E(K, x)) = x$  where  $D()$  is the decryption function. Assume that the user  $u$  poses the SQL query

Q(DB):       SELECT EMPLOYEE.Id FROM EMPLOYEE  
               WHERE Salary > 100,000 AND (EMPLOYEE.ProjId /2)\*2 = EMPLOYEE.ProjId

which returns the employees who make more than \$100K and work in projects with even-numbered project ids. Assume that this query *cannot* be rewritten into a *single* anti-tamper database query because the encryption function  $E(K, x)$  is such that there is no way of rewriting the predicate

$$(EMPLOYEE.ProjId /2)*2 = EMPLOYEE.ProjId$$

into a predicate for  $DB_E$ . However, at query evaluation time, ProjId can be decrypted using (a transformed version of) the key K. Assume that the relation EMPLOYEE and the attribute names Id, Salary, and ProjId are encoded as the characters R, A, B, and F; and  $f()$  is the encryption function for the Salary attribute. Then

(1) the Agent first splits the query Q(DB) into two queries:

Q<sub>1</sub>(DB):    SELECT EMPLOYEE.Id FROM EMPLOYEE WHERE Salary > 100,000  
               and

Q<sub>2</sub>(DB): SELECT EMPLOYEE.Id FROM EMPLOYEE WHERE (EMPLOYEE.ProjId /2)\*2 = EMPLOYEE.ProjId

(2) The agent rewrites Q<sub>1</sub>(DB) and Q<sub>2</sub>(DB) as

Q<sub>E1</sub>(DB<sub>E</sub>): SELECT R.A FROM R WHERE B > f(100,000)

Q<sub>E2</sub>(DB<sub>E</sub>): SELECT R.A FROM R WHERE (Decrypt(V(K), R.F)/2)\*2=Decrypt(V(K), R.F)

and sends both queries to the anti-tamper database  $DBMS_E$  for evaluation.

Here Decrypt is a user-defined procedure stored at  $DBMS_E$  which implements the decrypting function  $D()$ . The value of  $V(K)$ , sent by the agent, is sufficient<sup>2</sup> for the user-defined procedure Decrypt() to decrypt a given encrypted ProjId value.  $V(K)$  is retained at the anti-tamper site *during* the evaluation of  $Q_{E2}(DB_E)$  for Decrypt() to repetitively decrypt the encrypted ProjId values. That is, the evaluation of  $Q_{E2}(DB_E)$  takes place by first decrypting each R.F value as ProjId at the anti-tamper site, and then evaluating the SQL predicate  $(ProjId /2)*2 = ProjId$ .

(3) The agent then collects the outputs  $O_1(Q_{E1}(DB_E))$  and  $O_2(Q_{E2}(DB_E))$ , decrypts  $O_1(Q_{E1}(DB_E))$  into  $O_1(Q(DB))$ , and  $O_2(Q_{E2}(DB_E))$  into  $O_2(Q(DB))$ , takes the intersection of  $O_1(Q(DB))$  and  $O_2(Q(DB))$ , and sends the result to the user as the answer to the original query Q(DB).

The anti-tamper-site decrypting approach, while it works, has problems. First, if the adversary captures the anti-tamper database while the query  $Q_{E2}(DB_E)$  is in the process of being evaluated,  $V(K)$  will be compromised and all ProjId attribute values will be compromised. However, to the anti-tamper database software  $DBMS_E$ ,  $V(K)$  is a temporary value employed only during query processing, and, assuming that  $DBMS_E$  is instructed to destroy all temporary data related to a query after the query is processed<sup>3</sup>, the anti-tamper database is secure—except for the time window of processing the query  $Q_2(DB_E)$ . The second problem is, this approach is potentially much more costly in time as (a) extra decrypting takes place at the anti-tamper database, and (b) the Agent needs to have high computational power and, possibly, secondary storage capabilities since it needs to take a union or intersection operation of possibly two very large data sets. As a partial solution to (a), we propose “query splitting” which splits the original query so that only minimal decrypting takes place. And, as a partial solution to (b), when the agent is a secure-site agent with  $DBMS_{Agent}$ , we propose the use of  $DBMS_{Agent}$  for database query processing needs of the Agent.

Yet another approach is to defer some of the query processing that is not doable at the anti-tamper site to the agent site with the assumption that the Agent is located at the secure site, and  $DBMS_{Agent}$  is available at the secure site as well. The agent requests, using the query  $Q_{Agent}(DB_E)$ , the data to the secure site, decrypts and stores it into  $DBMS_{Agent}$ , and runs  $Q_{Agent}(DB)$  against  $DBMS_{Agent}$ . However, this approach, while secure, is even more time-costly than the anti-tamper-site decrypting approach as data is shipped to the secure site and then stored to another database.

### 3 Order- and Distance-Preserving, Open-Form Encryption

In this paper, we concentrate only on the primitive data type *integer*. Next, we illustrate with two examples the issues with the data type *integer*.

<sup>2</sup>  $V(K)$  is a transformed form of the key K so that K cannot be compromised during its transfer to and its use at the anti-tamper site.  $V(K)$  is a “one-way” function; i.e., it is infeasible to recover information about K from knowledge of  $V(K)$ .

<sup>3</sup> This is the case for commercial databases if the user requests that the cache is cleared after each query evaluation.

**Def'n (Order-Preserving Encryption).** Consider attribute  $V$  of relation  $R$ , and the encryption function  $f()$  for  $V$  values. The encryption function  $f()$  is order-preserving if when  $a > b$  for any two  $V$  values  $a$  and  $b$  in the original database  $DB$  then  $f(a) > f(b)$  in the anti-tamper database  $DB_E$ .

In other words, the encryption of attribute  $V$  retains the ordering in  $V$ . Arithmetic comparison operators  $<$ ,  $>$ ,  $\geq$ , and  $\leq$  (but, not  $=$  and  $\neq$ ) of primitive data types (e.g., integers) depend on the total ordering of attribute values. We give an example.

**Example 3.** Consider the Salary attribute of the EMPLOYEE relation. The SQL query

```
SELECT * FROM EMPLOYEE WHERE Salary > 100,000
```

returns those employees making more than \$100K annually. For the Agent to rewrite this query as

```
SELECT * FROM R WHERE B > f(100,000)
```

the Salary attribute should be encrypted using an encryption function that preserves the ordering of Salary values.

Integer attributes can also participate in arbitrary arithmetic expressions of SQL queries. Let us consider one of the simplest arithmetic expressions, namely, arithmetic difference.

**Def'n (Difference-Preserving Encryption).** Given an attribute  $V$  with nonnegative integer or real values, and an encryption function  $f()$ ,  $f()$  is *difference-preserving* if, for any two attribute  $V$  values  $a$  and  $b$  where  $(a - b) = k$ , we have  $f(a) - f(b) = r * k$ , where  $r$  is a constant.

**Example 4.** Assume that the relation EMPLOYEE has the attribute Age that lists the age of an employee, which is encrypted as attribute C using the encryption function  $f()$ . Consider a query where the user wants to know the names of pairs of employees who have an age difference of 10 or more years, which is expressed in SQL as

```
SELECT E.id, T.Id FROM EMPLOYEE E, EMPLOYEE T WHERE (E.Age - T.Age) > 10
```

When the encryption function  $f()$  is distance-preserving, this query can be rewritten as

```
SELECT E.A, T.A FROM R E, R T WHERE (E.C - T.C) > 10*r
```

where  $f(x) = r^{-1}x + c$ .

In general, an encryption function  $f(x)$  that is difference-preserving for integers (and reals) has to be affine, i.e.,  $f(x) = Ax + C$ , where  $C$  is a constant and  $A = r$  if  $x$  is a scalar.

In example 4, we have used  $f(x) = Ax + C$  which is a *closed-form encryption function* with a closed-form inverse, which we refer to as a *closed-form decryption function*. For a given closed-form encryption function  $f()$ , there may not exist a closed-form decryption, i.e., a closed-form function  $f^{-1}()$ .

When we employ an open-form encryption function, the system uses an *encryption algorithm*, which is more than computing a closed-form function, to encrypt original database  $DB$  into  $DB_E$ . Since  $DB_E$  creation is a one-time (database creation-time) task, employing a more time consuming encryption algorithm is acceptable. In addition, query rewriting may also require the execution of the encryption algorithm for constants appearing in the query.

For decrypting query outputs as well as for intermediate processing, the agent also needs to execute a *decryption algorithm*. If direct decrypting by the agent is expensive for large data (e.g., the algorithm in the next section) then decryption algorithms may be implemented by simply storing encrypted values into secondary storage-based index structures such as B+ trees [RG00], and employing searches on these structures where, given the encrypted value  $v$  in  $DB_E$ , the decryption algorithm searches the data structure and locates the original value  $f^{-1}(v)$  of the original database  $DB$ .

Consider an integer-valued attribute  $V$  with values  $X$ , to be encrypted using open form order-preserving encryption. Let us assume that the encryption function  $f(X)$  and its inverse are in the form of  $E(K, X)$  and  $D(K, Y)$ , respectively, where  $K$  is the secret key. We would like to define a family of functions  $Y = E(K, X)$ ,  $X = D(K, Y)$ , where  $X, Y$ , and  $K$  are nonnegative integers:

1.  $K$  is the secret key. Given  $K$ ,  $E$  and  $D$  should be efficiently computable.
2. For all  $X, Y, K$ , we have  $D(K, E(K, X)) = X$ . That is,  $D$  decrypts any number  $X$  encrypted by  $E$  using the same key. We will write  $E_K(X)$  for  $E(K, X)$  and  $D_K(Y)$  for  $D(K, Y)$ ; then we want  $D_K(E_K(X)) = X$ .

3. It should be hard (see inference control) to find X from Y or Y from X in the absence of knowledge of K, even assuming complete knowledge of the functions E and D.
4. If  $X < X'$  then  $E(K, X) < E(K, X')$  for any K. (Order-preservation)

Assume that the domain for our function  $E_K$ , is the integers from 1 to N, and the range is the integers from 1 to M. For fixed K, let  $y_n = E_K(n)$  for  $1 \leq n \leq N$ . Define a new sequence  $z_n$  by

$$z_1 = y_1 - 1, \quad z_{n+1} = y_{n+1} - y_n - 1.$$

In other words, the  $z_i$ 's are the differences (minus 1) between successive values of the encryption function  $E_K(n)$ . Then condition 4 above is equivalent to the statement

$$z_n \geq 0 \text{ for all } n$$

If we have any order-preserving function we can define the corresponding sequence  $z_n$ ; and the converse is also true: given a sequence  $z_1, z_2, \dots, z_n$  of nonnegative integers, there is a uniquely determined order-preserving function for which  $z_i$ 's are the differences. Furthermore, we have the algorithms in figure 1 for computing the encryption function  $E_K(n)$  and its inverse  $D_K(y_n)$ .

**Encryption:**

$E_K(1) := 1 + z_1;$   
 $E_K(n+1) := E_K(n) + 1 + z_{n+1};$

**Decryption:**

**Input:** Y

**Output:**  $D_K(Y)$

**begin**

$i := 0;$    $W := Y;$

**while**  $W > i$  **do begin**  $i := i + 1;$    $W := W - z_i$

**endwhile;**

**if**  $W = i$  **then** return  $D_K(Y) := i$  **else** output ``Failure";

**end.**

Figure 1. Encryption and decryption algorithms for order-preserving, integer-to-integer encryption

Our goal is to construct a family of functions indexed by a key K, which contains as little information as possible. One way to achieve this is to generate a pseudorandom sequence  $z_i$  using an initial seed K, and then use the algorithms in figure 1 to define the encryption and decryption functions. As for the time it takes to execute the algorithms, apart from the sequence generation, the only operation is addition, so the above decryption algorithm is feasible at least for reasonably small values of N. For our salary running example, where if the salary is in hundreds, we can assume  $N < 20,000$ , which would be simple to implement. For large N values such as  $N = 10^{12}$ , one can use secondary storage-based indexing structures such as B+ trees, through which the decryption at the agent can be done fast. Finally, there are a large number of well-known algorithms [MOV97] for generating pseudorandom integers efficiently which have known security properties.

It is clear that any order-preserving function leaks information leading to compromise; so the question is how much information is leaked. This will depend on the distribution of the  $z_i$ . We may use  $z_i$ 's that are identically distributed in some range, but this does not appear optimal. For example, it might be better if each  $z_i$  is uniformly chosen from an interval depending on the sum of the previous  $z_i$ . Figure 2 lists such an algorithm.

- 1) *Generate a sequence of random integers  $y_i$  in the range [1, M].*

Assume that we have a family of pseudorandom functions  $R[K, n] = x_n$ , where K is the secret key and  $n = 1, 2, 3, \dots$ . Typically this generates a stream of random bits, which can then be used to produce random integers [MOV97]. There are a large number of pseudorandom number generators with useful security properties. See, for example, M. Blum and S. Micali [BM84] for an early example. Here, the key serves as the initiator of the sequence.

- 2) *Define  $z_i$  by the rules:*

$S_1 := z_1 := y_1;$

**for**  $k \geq 1$  **do begin**  $A_k := M - S_k;$    $z_{k+1} := \text{Int} [y_{k+1} * A_k / M];$  **endfor**

$S_{k+1} := S_k + z_{k+1};$

*The encryption function is then defined by  $f(k) = S_k$ .*

Figure 2. Encryption function  $f()$  with uniformly distributed and expanding  $z_i$ 's.

For distance-preserving encryption,  $z_i$ 's need to be identically distributed in some range, but this leaks more information; see section 3.2 on inference control analysis. And, it is possible that the integer-valued attribute  $V$  instances  $X$  (being encrypted) may obey a known distribution  $D$  (such as the normal distribution), and this fact may be available to the adversary. In such a case, the task of choosing  $z_i$  values is augmented with the goal that the distribution of the encrypted values  $y_i$  is unknown to the adversary.

### 3.1 A Class of Queries with Simple Query Transformation and Same-Cost Query Processing

For the class of SQL queries equivalent to safe relational calculus queries, there is a very straightforward transformation from  $Q(DB)$  to  $Q_E(DB_E)$ :

**Query Transformation Rule:** Replace each constant  $c$  in  $Q(DB)$  with  $f(c)$  to obtain  $Q_E(DB_E)$ .

The query  $Q$  of example 1 illustrates the use of the Query Transformation Rule.

Theorem 1 below states that  $f()$  as defined in figure 2 is a group homomorphism from  $DB$  to  $DB_E$  with respect to the set of SQL queries equivalent to safe relational calculus queries. Moreover, the cost of evaluating a transformed query over the encrypted database is the same as the cost of evaluating the original query over the original database.

**Theorem 1.** Let  $DB$  be a relational database,  $f()$  be an encryption function as defined in Figure 2,  $DB_E$  be the encrypted database obtained from  $DB$  using  $f()$ , and  $O(Q(DB))$  be the output of query  $Q$  on database  $DB$ . Then

- (a) For any SQL query  $Q(DB)$  that is expressible in safe relational calculus, the corresponding single SQL query  $Q_E(DB_E)$  obtained using the Query Transformation Rule is such that applying  $f^{-1}(y)$  to each encrypted value  $y$  in  $O(Q_E(DB_E))$  produces  $O(Q(DB))$ .
- (b) Given any relational DBMS  $M$ , the query evaluation cost of  $Q(DB)$  on  $M$  is equal to the query evaluation cost of  $Q_E(DB_E)$  on  $M$ .

**Proof:** see [OSC03].

Theorem 1 is highly practical since (a) the class of safe relational calculus queries is a reasonably large class, (b) transformation from  $Q$  to  $Q_E$  is straightforward and not costly, and (c) all things being equal between  $DB$  and  $DB_E$  except encrypted values, processing  $Q$  and  $Q_E$  take the same amount of time. Nevertheless, Theorem 1 fails for SQL queries with arithmetic expressions and/or aggregate functions, as well as for SQL queries with object-relational features such as derived or complex types (as opposed to primitive types integers, reals, and strings). We give an example.

**Example 5.** Assume that the relation EMPLOYEE and the attribute name Salary are encrypted as characters  $R$  and  $B$ , and the attribute Salary is encrypted using an order-preserving encryption function  $f()$ . Then, the SQL query

```
SELECT AVG (EMPLOYEE.Salary) FROM EMPLOYEE WHERE Salary > 100,000
```

will need the evaluation of two queries:

```
QE1(DBE): SELECT SUM(R.B) FROM R WHERE B > f(100,000), and
```

```
QE2(DBE): SELECT COUNT(R.B) FROM R WHERE B > f(100,000)
```

And, the Agent will compute the answer as a function of the responses  $O_1$  and  $O_2$  to  $Q_{E1}$ ,  $Q_{E2}$  respectively, and  $f()$ . The answer is given by  $f^{-1}(O_1/O_2)$ . This formula works because the special form of  $f()$  preserves arithmetic means.

We make three observations. First, the query rewriting approach of Example 5 will fail if  $f()$  is not additive, i.e.,  $f(a + b) = f(a) + f(b)$ . That is, for any  $x$  and  $y$  values in the Salary attribute, the additivity property guarantees that  $f^{-1}(f(x) + f(y)) = f^{-1}(f(x + y)) = (x + y)$ . For example, the function  $f(x) = x^2$  is not additive, and cannot be used as the encryption function here. Second, open-form encryption functions by definition do not satisfy the additivity property. Third, when the aggregate operations are used in nested SQL queries (with correlated variables), the issue of rewriting the query becomes even more difficult.

**Example 6.** Consider the query in example 5. Now, assume that we have used an "instance-based" encryption; i.e., for each salary value  $a$ , there is a distinct  $f(a)$  value, not expressible in a closed form and maintained in a secondary data

structure at the Agent. In this case, the query rewriting scenario of example 5 fails, and we are forced to return *all* of the salary values in the desired range as

$Q_{E3}(DB_E): \quad \text{SELECT R.B FROM R WHERE B > f(100,000)}$

with  $O_3$  as the output. Then the Agent will need to (i) decrypt and sum up all of the values in the output  $O_3$  into  $X$ , and (ii) compute the output of the query as  $X/O_2$ . This scenario will incur heavy time delays since  $DB_E$  salary values have to be shipped to the Agent, and the Agent needs to perform (possibly, a disk-based) addition, independent of the database query optimization process.

Thus, one important goal in query rewriting is to have minimal performance degradation in transforming and evaluating the expected set of queries of the original database  $DB$ .

### 3.2 Inference Control

Let  $V$  be an integer-valued attribute and Domain of  $V$ ,  $\text{Dom}(V)$  be a set of integers in  $[1, N]$ . Let the encrypted domain  $E(\text{Dom}(V))$  be a set of integers between the range  $[1, M]$ , where  $M \gg N$ . Let the attribute  $V$  values be represented by  $v_i$ ,  $1 \leq i \leq n$ , which are encrypted to  $y_i$ 's with an encryption function  $f(v_i)$ .  $v_i, y_i$  are monotonically increasing, that is  $v_i < v_j$  when  $i < j$ , and  $y_i < y_j$ . We define the adversary's Knowledge Space (KS) as the set of two-tuples  $(v_i, y_i)$  i.e., for  $(v_i, y_i) \in \text{KS}$ , the adversary knows in advance that  $y_i = f(v_i)$ . And, the adversary may or may not know that  $v_i$ 's range is  $[1, N]$ , and  $y_i$ 's range is  $[1, M]$ .

We now briefly investigate the compromise risk; i.e., given KS, how much the adversary can infer about the values  $v_i$ . More precisely, when  $|\text{KS}| = k$ , we want to know the probability that an unknown  $(v_i, y_i)$  pair is revealed to the adversary. The system is said to be *compromised* when  $(v_i, y_i)$  is revealed to the adversary with a probability above the threshold  $\tau$ .

Let  $v_{\text{guess}}$  denote a guess by the adversary for  $v_r$ . Let  $P(v_{\text{guess}}=v_r | \text{KS})$ ,  $(v_r, y_r)$  is not in KS, denote the probability that the adversary learns  $f^{-1}(y_r)=v_r$  given KS. Then, we select the system-defined parameters so that  $P(v_{\text{guess}}=v_r) < \tau$  for all  $j$ .

When the adversary (a) does not know any  $v_i$ , that is,  $\text{KS} = \{ \}$ , and (b) knows that the domain range of  $V$  is  $[1, N]$ , and the encrypted domain range is  $[1, M]$  then the probability that  $n$  unknown  $v_r$  values are revealed from  $n$  distinct  $y_r$  values is equivalent to the probability of guessing  $n$  numbers over the range  $[1, N]$ . Choosing  $y_1$ , the adversary guesses  $v_1$  over  $[1, N]$  by  $v_{\text{guess}}$ . Then the possible range of  $v_1$  is  $[1, N-n+1]$  and  $P(v_{\text{guess}}=v_1) = 1/(N-n+1)$ . Similarly, with  $\text{KS} = \{(v_1, y_1)\}$ , the possible range for  $v_2$  is  $[v_1+1, N-n+2]$  and  $P(v_{\text{guess}}=v_2 | \text{KS}=\{(v_1, y_1)\}) = 1/(N-n-v_1+2)$ . Generalizing, we have  $P(v_{\text{guess}}=v_{k+1} | \text{KS}=\{(v_1, y_1), (v_2, y_2), \dots, (v_k, y_k)\}) = 1/(N-n-v_k+k+1)$ . Note that the uncertainty bound for each value depends on the previously compromised value and the next compromised value. Furthermore, if the adversary does not know the size of the encrypted domain  $M$ , (s)he can use the largest encrypted value  $y_n$  as the upper bound for  $M$ . Finally, the worst case is the case where  $v_i$  values in KS are evenly distributed over the domain range.

To distinguish the known values in KS from the original data instances  $v_i$  of  $V$ , we denote  $\text{KS} = \{(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)\}$ , where  $t_i (= y_i)$  is the encrypted value of  $s_i (= v_i)$ , and  $s_i < s_j$  when  $i < j$ . For any two consecutive elements  $s_j$  and  $s_{j+1}$  of KS, we denote by  $S$  a set of  $v_j$  values between  $s_j$  and  $s_{j+1}$ , and by  $T$  the set of encrypted  $y_i$  values between  $t_j$  and  $t_{j+1}$ . Assuming  $v_i$  values in  $S$  are equally likely (i.e., uniformly distributed) to be anywhere between  $s_j$  and  $s_{j+1}$ , we can compute the compromise probability as follows. Let  $v_{\text{guess}}$  denote the adversary's guess for  $v_r=s_{j+1}$  in  $S$ . Then  $P(v_{\text{guess}}=v_r) = 1/(s_{j+1} - s_j - |S|)$ . That is, once the values  $s_j$  and  $s_{j+1}$  are known by the adversary, the corresponding range between  $t_j$  and  $t_{j+1}$  is of no significance for the uncertainty range of  $v_r$ . Similarly, when  $v_r = s_1 - 1$  then  $P(v_{\text{guess}}=v_r) = 1/s_1 - (\text{no. of encrypted values less than } t_1)$ . And, when  $v_r = s_k + 1$  then  $P(v_{\text{guess}}=v_r) = 1/N - s_k - (\text{no. of encrypted values after } t_k)$ . Combining the three cases for all  $v_r$  values, compromise occurs when  $\max_{v_r} \{P(v_{\text{guess}}=v_r | \text{KS})\} > \tau$ .

It is possible to have domain values with different distributions. Let  $R$  be a range between any two known values  $s_j, s_{j+1}$  in KS, and having  $|T|$  unknown values distributed over  $R$  with a hypergeometric distribution. Assume that the adversary guesses  $m$  values from  $R$ . If we let random variable  $X$  denote the number of correctly guessed values then  $X=k$  means the adversary got exactly  $k$  values correct. So, the probability of  $X=k$  would be:

$$P(X = k) = \frac{\binom{|T|}{k} \binom{|R| - |T|}{m - k}}{\binom{|R|}{m}}$$

By the definition of hypergeometric distribution,  $E(X) = m|T|/R$  gives the expected number of compromised values.

#### 4 Order-Preserving Closed-Form Encryption and Decryption

Because database decryption is a one-time process, whether the encryption is closed or open makes little difference in the efficiency of the encryption. However, closed-form decryption has advantages (and disadvantages). On the plus side, closed-form decryption allows anti-tamper-site decryption. In the absence of a closed-form decryption at the anti-tamper site, we need a fast decryption algorithm. Such an algorithm may not exist and/or may result in significant query processing time delays, because decryption at the anti-tamper site based on a secondary-storage-based data structure (e.g.,  $B^+$  tree) is not an option due to its inefficiency and security problems. The second advantage of closed-form decryption is speed; it simply requires a closed-form function evaluation. In comparison, open-form decryption may be costly--even when the decryption requests are sorted and batched. On the minus side, inferring the closed-form of the decryption function for an attribute amounts to compromising all values of the attribute unless one-way encryption functions (like  $V(K)$  of Example 2) are used to protect the key  $K$ .

Closed-form encryption for integer-valued attributes has the disadvantage that, because a single closed-form function is employed for encryption, controlling the magnitude of the encrypted values becomes difficult, leading to overflow problems (for reals, this problem transforms into a problem of precision loss, which we do not directly discuss here to save space). For example, assume that the encryption function  $f(x)$  for the nonnegative integer-valued attribute  $V$  is  $f(x) = K \cdot x^2$  where  $K$ ,  $1 \leq K \leq 2^6$ , is the encryption key and we would like to use 1-word (i.e., 32-bit) representation for encrypted  $V$  values. Then, all  $V$  values should be less than  $2^{13}$ , otherwise, the encrypted values will have an integer overflow problem. One can employ a variety of techniques to control the sizes of encrypted values; here we discuss one approach: encrypting a single integer value into a set of integers. However, as pointed out before, translating one data type (integer) into another (sequence of integers) makes it difficult to transform integer operators in SQL queries into the operators of the type *sequence of integers*.

We would like to find *encryption functions* whose closed-form inverses exist, are cheap to compute, and “lossless”. Here are two examples.

- (a) The functions  $x^2$  and  $x^{1/2}$  with domains  $[0, \infty)$  are inverses to each other.
- (b) The inverse of the function  $ax^2 + b$  is  $[(x-b)/a]^{1/2}$ .

Consider the function  $f: S \rightarrow T$ . A function  $f^{-1}: T \rightarrow S$  is *the inverse of f* if  $f^{-1}(f(x)) = x$  for all  $x$  in  $S$  and  $f(f^{-1}(y)) = y$  for all  $y$  in  $T$ . A function  $f$  is *invertible* if it has an inverse. As an example,  $f(x) = ax^2 + b$  for the domain  $[0, \infty)$  is invertible since it has the inverse  $f^{-1}(y) = [(y - b) / a]^{1/2}$  in that  $f^{-1}(f(x)) = x$  for all  $x$  in  $[0, \infty)$ .

Not all functions have inverses. Consider  $f: S \rightarrow T$ . The function  $f$  is invertible if and only if  $f$  is one-to-one and maps  $S$  onto  $T$ . As an example,  $f(x) = x^2$  is *not* one-to-one for  $x$  in  $\mathcal{R}$  (reals). However, it is one-to-one (and onto) if we restrict  $x$  to domain  $[0, \infty)$ ; and, therefore its inverse exists and is  $x^{1/2}$ .

Note that query output decrypting can also be computationally expensive, when the output is large. It is always much faster to decrypt if the decrypting function involves only additions and deletions, as opposed to multiplications and divisions. And, it is even more costly if the function involves square root operations, as opposed to multiplications and divisions. Thus, query processing cost functions need to take into account the expected size of the query output as well as the type of the inverse function used for decrypting. In this

section, we investigate encryption functions that are polynomial functions applied to each value of an integer attribute with a domain range of  $[1, N]$ . We say that an encryption algorithm  $E$  is *lossless* for an attribute  $V$  when there exists a decryption algorithm  $D$  such that, for any value  $x$ ,  $1 \leq x \leq N$ ,  $D(E(x)) = x$ .

When the attribute value  $x$  is encrypted using a function, say,  $E(x) = C_1x + C_0$ , we refer to the constants  $C_0$  and  $C_1$  as the key  $K$ . We assume that users are informed of the encryption function, but not the key (i.e., values of the constants,  $C_0$  and  $C_1$ ). However, the range of the constants, say,  $[1, 2^5]$ , may or may not be made available to users. We call the number of such constants  $C_i$  as the *degree of security*. Our goal in this section is to come up with an encryption algorithm  $\text{Decide\_E}$  such that, given the desired degree of security  $n$ , the algorithm  $\text{Decide\_E}$  locates a “lossless” encryption function  $E()$  with the degree of security of  $n$ .

#### 4.1 Single Encryption Function and its Inverse for Decryption

One encryption approach is to use a single polynomial function  $f(x)$  as the encryption function  $E$ . Obviously, in this case, the degree of security is the number of constants employed by the function  $f(x)$ . Therefore, given  $n$  as the degree of security, the goal is to find an  $n$  degree polynomial that has an inverse function in closed form:

$$F(x) = C_n X^n + C_{n-1} X^{n-1} + \dots + C_1 X + C_0$$

**Remark 1.** If  $f(x)$  be a strictly increasing function, the inverse function  $f^{-1}(x)$  exists, and is defined on the set of values of  $f$ .

The positivity of the derivative gives a good test when a function is strictly increasing (decreasing): Whenever the function differentiates and its derivative is positive ( $\text{dev}(f(x)) \geq 0$ ) or negative ( $\text{dev}(f(x)) \leq 0$ ), there is an inverse.

**Remark 2.** Let  $f$  be a continuous function on the closed interval  $[x_1, x_2]$  and assume that  $f$  is strictly increasing. Let  $f(x_1) = y_1$  and  $f(x_2) = y_2$ . Then the inverse function is defined on the closed interval  $[y_1, y_2]$ .

Thus, to check whether the function has the inverse function or not, we can compute  $\text{dev}(f(x)) = 0$ . If there exists at most one solution for the above equation, then the inverse function of the polynomial exists for all  $x$ . If there exists more than one solution, we find the largest  $k$  such that  $\text{dev}(f(k)) = 0$ ; and the inverse function exists for  $x \geq k$  (by Remark 1). In general, the closed form of the inverse function of an arbitrary polynomial function may not exist even if the inverse itself exists. Therefore, next we introduce the approach of multiple encryption functions.

#### 4.2 Multiple Encryption Functions and an Inverse

Instead of finding the inverse of a single  $n$  degree polynomial for a requested  $n$  degree of security, we now apply  $n$  ‘simple’ functions iteratively where each function has its closed form inverse.

Some examples of simple polynomials with well-defined inverse functions are listed below.

$f_1(x) = C_0 x + C_1$	$f_1^{-1}(y) = 1/C_0 y - C_1/C_0$	For all $x$ .
$f_2(x) = C_2 x^2 + C_3$	$f_2^{-1}(y) = + (y/C_2 - C_3/C_2)^{1/2}$	For $x \geq 0$
	$f_2^{-1}(y) = - (y/C_2 - C_3/C_2)^{1/2}$	For $x \leq 0$
$f_3(x) = C_6 x^3 + C_7$	$f_3^{-1}(y) = (y/C_6 - C_7/C_6)^{1/3}$	For all $x$ .

Given  $n$  as the desired level of security, our goal is to find a sequence of functions  $f_i$ ,  $1 \leq i \leq k$  from the above list as a sequence of encryption functions with, altogether,  $n$  constants  $C_i$  (as the key  $K$ ) so that applying the inverses of each function in the sequence in reverse order constitutes the decryption. The encryption algorithm is applied as the sequence  $f_i$  of functions in such a way that the output of  $f_i(x)$  becomes the input of  $f_{i+1}(x)$  for  $1 \leq i \leq k - 1$ .

**Example 7.** Let  $f_1(x) = C_0 x + C_1$ ,  $f_2(x) = C_2 x^2 + C_3$ ,  $f_3(x) = C_6 x^3 + C_7$   
 $E(x) = f_3 (f_2 (f_1(x))) = C_6 (C_2 (C_0 x + C_1)^2 + C_3)^3 + C_7$

Note that each function  $f_i$  is *monotone*; that is, for  $x_1 > x_2$ ,  $f(x_1) > f(x_2)$ .

### 4.3 Nonlossy multiple encryption functions

There are two types of errors that may occur and may make the encryption lossy when applying a sequence  $f_i$ ,  $1 \leq i \leq k$  of functions as encryption/decryption functions, namely, *overflow errors* (integers) and *precision (round-off) errors* (reals). To prevent overflow errors, the maximum value of each attribute domain should be considered. And, to prevent round-off errors, arithmetic precision (the number of significant bits) should be controlled in order to avoid possible information loss [Mar85], and, thus, lossy encryption.

When multiple functions are applied iteratively to encrypt attribute values of integer domain, the magnitude of the encrypted value rapidly increases even when we restrict the degree of each polynomial to either 1 or 2. Since computer arithmetic uses limited (e.g. 32-bit or 64-bit) number of bits to store a number, an attempt to store an integer of large magnitude (e.g., greater than  $2^{31} - 1$  for 32-bit arithmetic and greater than  $2^{63} - 1$  for 64-bit arithmetic) will produce an integer overflow [Str88].

The ranges for the integer values of encryption functions to prevent overflow errors are obtained by the precision calculation as illustrated next:

**Example 8.** For a two degree polynomial with 3 degrees of security, the overflow range for  $x$  while using the encryption function sequence  $E(x) = f_2(f_1(x)) = C_2 (C_0 x + C_1)^2 + C_3$  where  $1 \leq C_i \leq 2^5$ ,  $0 \leq i \leq 3$ , is obtained by the following formula for the dominant term  $C_2 * (C_0 * x)^2$  of the polynomial:

- 1) 32 bit arithmetic:  $- 2^{31} \leq C_2 * (C_0 * x)^2 < 2^{31}$ ;  $-(2^{31} / 2^{15})^{1/2} \leq x < (2^{31} / 2^{15})^{1/2}$ ;  
 $- 2^8 \leq x < 2^8$ .
- 2) 64 bit arithmetic:  $- 2^{63} \leq C_2 * (C_0 * x)^2 < 2^{63}$ ;  $-(2^{63} / 2^{15})^{1/2} \leq x < (2^{63} / 2^{15})^{1/2}$ ;  
 $- 2^{24} \leq x < 2^{24}$ .

To be more precise, the maximum value  $x$  at which  $f_2(f_1(x))$  does not overflow is  $\max_x f_2 = 2^8$  for 32-bit arithmetic. Similarly,  $\max_x f_2 = 2^{24}$  for 64-bit arithmetic. Therefore, after a quadratic function is applied during the encryption, to reduce the magnitude of encrypted values, one may apply the following functions during encryption;

- The log function  $f_{\log}(x) = \log_2 x + C$  where  $C$  is a security constant, or
- The mod function  $f_{\text{mod } C} x$  where  $C$  is a security constant.

In this section, we investigate the use of the log function during encryption. Note that the log function's input and output need to be real values.

Since encryption with log functions involves real number arithmetic, let us review the floating point representation. Assume that the floating-point representation has a base  $B$  and a precision  $p$  (i.e., the number of bits to keep the fractional part of a real number). For example, assume  $B = 2$  and  $p = 24$ , then the decimal number 8 is represented  $1.0000000000000000000000 * 2^3$ . That is, the floating point number is represented as  $\pm b.bb...b * B^e$ , where  $b.bb...b$  is called the significant bits and has  $p$  digits, and  $e$  is the exponent. More precisely  $\pm b_0 . b_1 b_2 b_3 \dots b_{p-1} * B^e$  represents the number  $\pm (b_0 + b_1 B^{-1} + \dots + b_{p-1} B^{-(p-1)}) B^e$ , ( $0 \leq b_j < B$ ). For 32 bit arithmetic with  $B = 2$ , 24 bits are used for significant bits. And for 64-bit arithmetic, 53 bits are used for significant bits [Str88].

When the encryption computations convert the domain of an attribute from integer to real (e.g., in the case of applying  $f_{\log}(x) = \log_2 x$ ) or even when the domain of an attribute is real, information loss occurs if significant bits overflow (i.e., the precision error or round-off error occurs). When the number of significant bits of a real number is over the maximum number of significant bits of each domain, it is no longer possible to obtain the original values when decrypting due to the precision error.

When the log function  $f_{\log}()$  is applied in the encryption function sequence, the precision loss should be controlled. In order to avoid precision errors, we define the precision range for  $x$  with the log function  $f(x) =$

$\log_2 x$  by locating when  $f(x)$  loses its precision. More precisely, we find  $x$  and  $x+1$  where  $y_n = \log(x)$ ,  $y_{n+1} = \log(x+1)$  and  $y_n = y_{n+1}$ . That is,

$$y_n = \log(x) = b_0 . b_1 b_2 b_3 \dots b_{p-1} * B^e = (b_0 + b_1 * 1/2^1 + b_2 * 1/2^2 + \dots + b_{p-1} * 1/2^{(p-1)}) * B^e$$

$$y_{n+1} = \log(x+1) = b_0 . b_1 b_2 b_3 \dots b_{p-1} b_p * B^e = (b_0 + b_1 * 1/2^1 + b_2 * 1/2^2 + \dots + b_{p-1} * 1/2^{(p-1)} + b_p * 1/2^p) * B^e$$

where  $b_p$  is  $(p+1)^{th}$  bit of the floating-point representation for  $y_{n+1}$ , and cannot be represented with the available 32-bit (or 64-bit) floating point representation. That is,  $b_p$  is out of the precision range. Table 1 computes the decimal number (for 32-bit and 64-bit real number representations) at which the function  $f_{i,\log}(x) = \log_2 x$  loses its precision.

	32 Bit	64 Bit
x	757283 <sub>10</sub>	2.02967093951847 * 10 <sup>14</sup> <sub>10</sub>
x+1	757284 <sub>10</sub>	2.02967093951848 * 10 <sup>14</sup> <sub>10</sub>
Log(x) = Log(x+1)	19.530474 <sub>10</sub>	47.528239 <sub>10</sub>

Table 1. Precision Limits of  $\log_2(x)$  for 32-bit and 64-bit number representations

**Example 9.** Consider 32-bit arithmetic with the internal representation (Sign: Fraction: Exponent).

When the precision error occurs for 32-bit arithmetic, at  $x = 757283$  and  $x + 1 = 757284$ , the internal representations of  $\log_2 x$  and  $\log_2(x+1)$  are identical.

$$\begin{aligned} \log_2 x = 19.530474_{10} &= 10011.11010011011111001110_2 \\ &= 1.001111010011011111001110 * 2^4 \\ &= (1 * 2^0 + 0 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3} + 1 * 2^{-4} \dots + 1 * 2^{-23}) * 2^4 \end{aligned}$$

internal representation : 0 100111101001101111100111 0000100

$$\begin{aligned} \log_2(x+1) = 19.530474_{10} &= 10011.11010011011111001111_2 \\ &= 1.001111010011011111001111 * 2^4 \\ &= (1 * 2^0 + 0 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3} + 1 * 2^{-4} \dots + 1 * 2^{-23} + 1 * 2^{-24}) * 2^4 \end{aligned}$$

internal representation : 0 100111101001101111100111 0000100

To summarize, for the 32-bit representation, when the internal representation of  $\log_2(x)$  reaches to the 25<sup>th</sup> significant bit, the precision error occurs. From Table 1, we see that, after the application of  $\log_2(x)$  where  $x$  is a 32-bit or 64-bit number, the maximum value of  $\text{Int}(\log_2(x))$  (where  $\text{Int}$  denotes the function that returns the whole integer part of its input) is 19 or 47 for 32-bit or 64-bit arithmetic, respectively. On the other hand, the functions  $f_1()$  and  $f_2()$  do not produce precision errors since their input and output are integers, but may produce overflow errors. Furthermore, to avoid an early overflow error for attributes with large domain values, the function  $f_{i,\log}()$  is always applied first as the initial function before applying the regular encryption function sequence.

Therefore, assuming that the user requests a degree of security of  $2k$ , we propose to use the encryption function sequence  $E(x) = f_{k,\log}(f_{k-1,2}(f_{k-2,1}(\dots(f_{7,\log}(f_{6,2}(f_{5,1}(f_{4,\log}(f_{3,2}(f_{2,1}(f_{1,\log}(x))))))))))\dots))$ , to encrypt the values of a given attribute  $V$ , where the first subscript is used to number the functions in the sequence and the second subscript (1, 2, or log) refers to the function type (linear, quadratic or log).

Let's look at the problems encountered during the process of applying the encryption function sequence  $E(x)$  we propose above:

1. The translation from an integer to a real (needed as the input to  $f_{i,\log}()$  is a real) may cause a precision error. Since the input and the output of the function  $f_{i,\log}()$  are reals, the output of the function  $f_{i-1,2}(f_{i-2,1}(x)) = z$ , which is an integer less than  $2^{31} - 1$  for 32-bit representation, is

converted to a real to become an input to  $f_{i,\log}()$ . However, when  $z$  is greater than  $2^{24}$ , converting it into a real may (or may not) result in a precision loss.

2. The output of the function  $f_{i,\log}()$  may result in a precision error. For the 32-bit representation, when the internal representation of  $\log_2(x)$  reaches to the 25<sup>th</sup> significant bit (54<sup>th</sup> bit for 64-bit arithmetic), the precision error occurs. In other word, when the output value of the log function  $f(x) = \log_2 x$  is above the precision limit,  $f(x)$  loses its precision.

To control and avoid overflow and precision errors during the encryption process, we always deal with integer (encrypted) values, even after the application of the log function  $f_{i,\log}()$  as follows.

- After the application of each log function (i.e.,  $f_{i,\log}()$ ) in the encryption function sequence  $E$ , assume we have  $w_i$  and  $e_i$  as the whole part and the fractional part of the encrypted output. That is, let functions  $\text{Int}(r)$  and  $\text{Fract}(r)$  denote functions that, given a real value  $r$  as an input, return the integer (whole) part of  $r$ , and the fractional 24 bits of  $r$  as an integer, respectively. We maintain an “encrypted-value vector”  $V_E$  such that we

- (1) Store  $C * \text{Fract}(f_{i,\log}()) = e_i$ , where  $C$  is a constant (part of the key  $K$ ) in the range  $[1, 2^5]$ , into the encrypted-value vector  $V_E$ .
- (2) Provide  $w_i = (\text{Int}(f_{i,\log}()) + C')$ , where  $C'$  is a constant (part of the key  $K$ ) in the range  $[1, 2^5]$ , as an input to the function  $f_{i+1,1}()$ .
- (3) Repeat steps (1) and (2) above after the application of each  $f_{i,\log}()$  function.

The encrypted value of an attribute value  $x$  then becomes the vector  $V_E(x)$  that contains the fractional part of each  $f_{i,\log}()$  function application in the encryption function sequence  $E$ .

- To avoid precision errors described above, we extend our approach as follows. After applying  $f_{i-1}(f_{i-2}(x))$ , let  $z_i = f_{i-1,2}(f_{i-2,1}(x))$  for  $z_i < 2^{31}$  for 32-bit arithmetic. Before converting  $z_i$  into a real number in order to apply the log function  $f_{i,\log}$ , we split 32-bit integer  $z_i$  into two parts, say  $\text{Left}_i$  and  $\text{Right}_i$ .  $\text{Left}_i$  indicates the leftmost 19 bits of 32 bit integer  $z_i$  and  $\text{Right}_i$  indicates the rightmost 13 bits of  $z_i$ . Then a different function sequence is applied to each part separately. The function  $f_{i,\log}()$  is applied to  $\text{Real}(\text{Left}_i)$  as a function in the regular sequence, and a separate function  $f_1(x)$  is applied to  $\text{Right}_i$ .

Finally, in the encryption function sequence  $E(x) = f_{k,\log}(f_{k-1,2}(f_{k-2,1}(\dots(f_{7,\log}(f_{6,2}(f_{5,1}(f_{4,\log}(f_{3,2}(f_{2,1}(f_{1,\log}(x))))))))))\dots))$ , the first subscript of the outermost function  $f$  (in the above case, subscript  $k$  of the function  $f_{k,\log}()$ ) uniquely specifies the encryption function sequence. We refer to the subscript value  $k$  as the *index* of the encryption function sequence  $E$ . We give an example.

**Example 10.** Assume that the desired degree of security  $d = 14$ . Thus, the encryption function sequence  $E$  is determined by the index of the encryption function sequence  $k=6$ , given  $d$ , as follow;

$$E(x) = f_{6,2}(f_{5,1}(f_{4,\log}(f_{3,2}(f_{2,1}(f_{1,\log}(x))))))$$

$$\begin{aligned} \text{where } f_{1,\log}(x) &= \log_2 x \text{ with four constants } C_0, C_1, C_2, C_3 \\ f_{3,2}(f_{2,1}(x)) &\text{ with 3 constants, } C_4, C_5, C_6 \quad (\text{collectively for the two functions}) \\ f_{4,\log}(x) &= \log_2 x \text{ with four constants } C_7, C_8, C_9, C_{10} \\ f_{6,2}(f_{5,1}(x)) &\text{ with 3 constants } C_{11}, C_{12}, C_{13} \quad (\text{collectively for the two functions}) \end{aligned}$$

Consider the value  $a$  of attribute  $V$ , to be encrypted by  $E(x)$ . The encrypted-value vector  $V_E(a)$  for the value  $a$ :

$$V_E(a) = [(e_1, s_1), (e_2, s_2), \text{enc}]$$

Let  $l_i = \text{ExtractLeft}(z_i)$ . Note that  $l_i$  is an integer less than or equal to  $2^{19}$ . Also let  $r_i = \text{ExtractRight}(z_i)$  where  $r_i \leq 2^{13}$ .

$$z_1 = a$$

$$e_1 = C_0 * \text{Fract}(f_{1,\log}(\text{Real}(l_1))) \quad \text{Degree of security} = 1$$

$$w_1 = \text{Int}(f_{1,\log}(l_1)) + C_1 \quad \text{where } w_1 \leq (19 + C_1) \quad \text{Degree of security} = 2$$

$s_1 = C_2 (r_1) + C_3$	Degree of security = 4
$z_2 = f_{3,2} (f_{2,1} (w_1))$	Degree of security = 7
$e_2 = C_7 * \text{Fract} (f_{4,\log} (\text{Real} (l_2)))$	Degree of security = 8
$w_2 = \text{Int} (f_{4,\log} (l_2)) + C_8$ where $w_2 \leq (19 + C_8)$	Degree of security = 9
$s_2 = C_9(r_2) + C_{10}$	Degree of security = 11
$enc = f_{6,2} (f_{5,1} (w_2))$	Degree of security = 14

#### 4.4 Generating and Encryption Function Sequence with Desired Degree of Security

By applying the results of the previous section, the relationship between the encryption function sequence index  $k$  and the desired degree of security  $d$  can be defined as

$$d = a * ( (k/3) * 7) + b$$

where  $a$  is 0 if  $k < 3$  and  $a=1$  if  $k \geq 3$ ; and  $b$  is 0, 4, or 6 if  $(k \bmod 3)$  is 0, 1, or 2, respectively. Therefore, given the desired degree of security by the user, the system can generate the corresponding encryption function sequence  $E(x)$  in a straightforward manner.

Thus, we have proposed an encryption function sequence  $E(x)$  as an alternative closed form encryption function. The degree of security of  $E(x)$  is the number of constants  $C_i$  (coefficients) employed by the function sequence, and the set of  $C_i$  in  $E(x)$  constitute the secret key  $K$  in the encryption. However, since the input and output of  $E(x)$  are limited to integers, there is a diophantine equation problem in two variables. Algorithms to solve diophantine equations often employ the divisibility or the congruences of the parameters of the equations [Red96]. In fact there exists an algorithm to find solutions for the first order of diophantine equations [Red96], and for limited forms of the second order diophantine equations [Sch96]. Therefore, there may be possible adversary attacks employing diophantine equation algorithms. In such cases, the adversary doesn't need to know  $d$  pairs of  $(x_i, y_i)$  for  $E$  with  $d$  degree of security. However, such reasoning is limited only to first order equations, and some restricted form of second order equations (e.g., the family of  $x^2 + p = 2^n$ , where  $p$  is odd prime). There may not exist a solution for arbitrary  $n$  degree diophantine equations and there is no algorithm which can determine whether there exists a solution for an arbitrary  $n$ -degree diophantine equation whose degree  $\geq 3$  [Red96].

Since  $E(x)$  is composed of a (repeating) three function sequence block  $f_{3,2}(f_{2,1}(f_{1,\log}(x)))$ , and uses the whole part of  $\log$  function as the input to the next function, a sequence block  $f_{3,2}(f_{2,1}(f_{1,\log}(x)))$  is a second degree diophantine equation. A quick fix for the diophantine equation problem is that we don't allow the first degree function  $f_{k,1}(x)$  or the second degree function  $f_{k,2}$  in the form of  $y = x^2 + C$  as a final function of  $E(x)$ . Thus,  $E(x)$  includes the first degree function  $f_1(x)$  as the input to the next second degree function  $f_2(x)$ , but the final function of the sequence  $E(x)$  is either  $f_{k,\log}(x)$  or  $f_{k,2}(x)$ . Therefore, we always use the encryption function sequence index  $k$  where  $(k \bmod 3) = 1$ .

## 5 Conclusions

In this paper, we have proposed the first steps of an approach to securing databases: encrypt the database, and yet allow query processing over the encrypted database.

Much work remains to be done. Our approach needs to be extended to handle SQL queries with arithmetic expressions and aggregate functions as well as complex SQL queries with nested subqueries [SQL99]. We have discussed only the encryption of integer-valued attributes; encrypting and querying attributes of other primitive/complex data types is another research direction.

## 6 References

- [ALN87] N. Ahituv, Y. Lapid and S. Neumann, Processing encrypted data, CACM 20 (1987) 777-780.
- [AW89] Nabil R. Adam, John C. Wortmann: Security-Control Methods for Statistical Databases: A Comparative Study. ACM Computing Surveys 21(4): 515-556 (1989)

- [BM84] Blum, M., Micali, S., “How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits”, SIAM Journal on Computing, 13 (1984), 850-864.
- [BY87] E. Brickell and Y. Yacobi, On privacy homomorphisms, in: D. Chaum and W. L. Price, eds., Advances in Crypto-Eurocrypto’87 (Springer, Berlin, 1988) 117-125.
- [DP91] Duncan, G. and Pearson, R. (1991). Enhancing Access to Data While Protecting Confidentiality. Statistical Science, 6, 217-239.
- [DJW93] Duncan, G. T., Jabine, T. B., and de Wolf, V. A. (1993). Private Lives and Public Policies: Confidentiality and Accessibility of Government Statistics. National Academy Press, Washington, DC
- [HIM02] Hakan Hacigumus, Balakrishna R. Iyer, Sharad Mehrotra: Providing Database as a Service. IEEE ICDE 2002
- [HILM02] Hakan Hacigumus, Balakrishna R. Iyer, Chen Li, Sharad Mehrotra: Executing SQL over encrypted data in the database-service-provider model. ACM SIGMOD Conference 2002: 216-227
- [HS93] Joseph M. Hellerstein, Michael Stonebraker: Predicate Migration: Optimizing Queries with Expensive Predicates. ACM SIGMOD Conference 1993: 267-276
- [H94] Joseph M. Hellerstein: Practical Predicate Placement. ACM SIGMOD Conference 1994: 325-335
- [Jab93b] Jabine, T. B. (1993b). Procedures for Restricted Data Access. Journal of Official Statistics, 537-590.
- [JAKMM01] Sushil Jajodia, Vijayalakshmi Atluri, Thomas F. Keefe, Catherine D. McCollum, Ravi Mukkamala: Multilevel Security Transaction Processing. Journal of Computer Security 9(3): 165-195 (2001)
- [Mar85] Melvin J. Maron. Numerical Analysis – A Practical Approach, 1985 Macmillan Publishing Co., Inc.
- [MLS] Multi-level secure database management schemes: software review, CMU Software Engineering Institute, available at [http://www.sei.cmu.edu/str/descriptions/mlsdms\\_body.html](http://www.sei.cmu.edu/str/descriptions/mlsdms_body.html)
- [MOV97] Menezes, A.J, van Oorschot, P.C. and Vanstone, S.A., Handbook of Applied Cryptography, CRC Press, Boca Raton, 1997, page 239.
- [Neg] Goldberg, D., “What Every Computer Scientist Should Know About Floating Point Arithmetic”, ACM Computing Surveys, 1991, available at <http://citeseer.nj.nec.com/goldberg91what.html>
- [OSC03] Ozsoyoglu, G., Singer, D., Chung, S., “Querying Encrypted Databases”, Tech. Reprt, EECS, CWRU, available at <http://art.cwru.edu/secdb>
- [QL97] Xiaolei Qian, Teresa F. Lunt: A Semantic Framework of the Multilevel Secure Relational Model. IEEE TKDE 9(2): 292-301 (1997)
- [RAD78] R. L. Rivest, L. Adleman and M.L. Dertouzos, On data banks and privacy homomorphisms, in R. A. DeMillo et al., eds., Foundations of Secure Computation, Academic Press, New York, 1978, 169-179.
- [Red96] Redmond, Don. Number Theory, Marcel Dekker, Inc. New York, 1996.
- [RG00] Database Management Systems, R. Ramakrishnan, J. Gehrke, McGraw-Hill, 2000.
- [Sch96] Schumer, Peter D. Introduction to Number Theory, International Thomson, 1996.
- [SQL99] Eisengerg, A., Melton, J., “Sql: 1999, formerly known as sql 3”, ACM SIGMOD Record, 28(1), 131-138, 1999.
- [Str88] Richard Startz. 8087/80287/80387 – Applications and Programming with Intel’s Math Coprocessors. 1988 Brandy Books, a division of Simon & Schuster, Inc.
- [Ullm89] Ullman, J.D., “Principles of Database and Knowledge-Base Systems”, Vol 1., Computer Science Press, 1989.
- [WJ01] Duminda Wijesekera, Sushil Jajodia: Policy algebras for access control: the propositional case. ACM Conference on Computer and Communications Security 2001: 38-47