ANTI-TAMPER DATABASES:
Querying Encrypted Databases

Gultekin Ozsoyoglu[#], David A. Singer[+], Sun S. Chung[#]
[#]EECS Department, [+]Math Department, Case Western Reserve University, Cleveland, OH 44106, (gxo3, das5, ssc7)@po.cwru.edu

Abstract 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.

Key words: querying encrypted database, encryption algorithm, homomorphism encryption/decryption functions

1. INTRODUCTION

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 [3, 4], 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 [17, 13], multilevel secure database architectures [5, 11, 7], database integrity lock architectures [7], or statistical database inference controls [1].
- 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. 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 encrypted as the characters R, A, and B. Then the SQL query Q(DB) over the database DB (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 decrypted as $g^{-1}(y)$, and returned as the output O(DB) of the query Q(DB).

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.*

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 a database DB, and the set S of SQL queries Q on DB that correspond to the set of safe relational calculus [16] queries[1]. 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 [12], i.e., for any query Q in S, we have $Q(DB) = f^{-1}(Q_E(f(DB)))$. 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 $Q_E(DB_E)$.

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. (no extra query specification/transformation burden on the legitimate users)
- 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.
- The commercial DBMS that hosts the data does not know that the data in its database is encrypted. Commercial DBMSs are complex proprietary software systems, any security technique that is deployable without modifying the DBMS is more desirable over ones that require changes to the DBMS.

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

[1] 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.

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. 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 $Q_E(DB_E)$ over the encrypted DB or
 ii) A set $\{Q_{Ei}(DB_E)|1\leq i \leq k\}$ of k different, k> 1, queries,
 and submits $Q_E(DB_E)$ or the query set $\{Q_{Ei}(DB_E)\}$, 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 $\{Q_{Ei}(DB_E)\}$, and returns the output $O(Q_E(DB_E))$ or the output set $\{O(Q_{Ei}(DB_E))\}$, 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 $\{O(Q_{Ei}(DB_E))\}$, the Agent decrypts each output $O(Q_E(DB_E))$, 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.

3. ORDER- AND DISTANCE-PRESERVING, OPEN-FORM ENCRYPTION

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

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 <, >, ≥, and ≤ (but, not = and ≠) of primitive data types (e.g., integers) depend on the total ordering of attribute 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.

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.

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. For decrypting query outputs as well as for intermediate processing, the agent also needs to execute a *decryption algorithm.*

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.
3. It should be hard(see 3.2 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 $z_1 = y_1 - 1$, $z_{n+1} = y_{n+1} - y_n - 1$. 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 $z_n \geq 0$ 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 $z_1, z_2,\ldots,z_n$ 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. There are a large number of well-known algorithms [8] for generating pseudorandom integers efficiently which have known security properties.

To minimize information leakage of the proposed order-preserving encryption algorithm, one can optimize the encryption algorithm by altering the distribution of the z_i. 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]=xn, where K is the secret key and n=1,2,3,... Typically this generates a stream of random bits, which can then be used to produce random integers[9]. There are a large number of pseudorandom number generators with useful security properties. See, for example, M. Blum and S. Micali [2] for an early example. Here, the key serves as the initiator of the sequence.
2) *Define z_i by the rules*:
    (a) $S_1 := z_1 := y_1$;
    (b) for $k \geq 1$, define A_k, z_{k+1} as $A_k =: M - S_k$; $z_{k+1} =: \text{Int}\,[y_{k+1} * A_k/M]$;
    (c) Define S_{k+1} as $S_k + z_{k+1}$

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

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

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 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)).

Proof: See the Technical Report [10].

Remark 1. 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. Follows from the integer-to-integer, order-preserving transformations of each attribute.

Theorem 1 and Remark 1 are 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 2. 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:

$Q_{E1}(DB_E)$: **SELECT SUM(R.B) FROM R WHERE B > f(100,000),**

$Q_{E2}(DB_E)$: **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 2 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 $f^{-1}(f(x) + f(y)) = f^{-1}(f(x + y)) = (x + y)$. 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 3. Consider the query in example 2. 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 2 fails, and we are forced to return *all* of the salary values in the desired range as

$Q_{E3}(DB_E)$: 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, Dom(V) be a set of integers in [1, N]. Let the encrypted domain E(Dom(V)) be a set of integers between the range [1, M], where M >> N. Let the attribute V values be represented by v_i, $1 \le i \le n$, 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 $(v_i, y_i) \in KS$, 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 |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 τ.

Let v_{guess} denote a guess by the adversary for v_r. Let $P(v_{guess}=v_r| KS)$, (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 $P(v_{guess}=v_r) < \tau$ for all r.

When the adversary (a) does not know any v_i, that is, 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_{guess}. Then the possible range of v_1 is [1, N-n+1] and $P(v_{guess}=v_1) = 1/(N-n+1)$. Similarly, with $KS = \{(v_1, y_1)\}$, the possible range for v_2 is $[v_1+1, N-n+2]$ and $P(v_{guess}=v_2 \mid KS=\{(v_1, y_1)\}) = 1/(N-n-v_1+2)$. Generalizing, we have $P(v_{guess}=v_{k+1} \mid KS=\{(v_1, y_1), (v_2, y_2), \ldots, (v_k, y_k)\}) = 1/(N-n-v_k+k+1)$. 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 $KS = \{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$, 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_{guess} denote the adversary's guess for $v_r=s_j+1$ in S. Then $P(v_{quess}=v_r) = 1/ s_{j+1}- s_j - |S|$. 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 $P(v_{guess}=v_r) = 1/s_1 -$ (no. of encrypted values less than t_1). And, when $v_r = s_k +1$ then $P(v_{guess}=v_r)= 1/ N - s_k -$ (no. of encrypted values after t_k). Combining the three cases for all v_r values, compromise occurs when $\max_{vr} \{P(v_{guess}=v_r \mid KS)\} > \tau$.

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:

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

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

The advantage of closed-form decryption is speed, whereas 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 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. 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.

We would like to find *encryption functions* whose closed-form inverses exist, are cheap to compute, and “lossless”. 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 , $1 \le x \le N$, $D(E(x)) = x$.

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. 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 E such that, given the desired degree of security n, the algorithm 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:

$$F(x) = C_nX^n + C_{n-1}X^{n-1} + \ldots + C_1X + C_0$$

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 dev(f(x)) = 0. If there exists at most one solution for dev(f(x)) = 0 , 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 dev(f(k)) = 0; and the inverse function exists for $x \geq k$ (by Remark 2). The problem with this approach is that 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.

$f_1(x) = C_0 x + C_1$ $\quad f_2(x) = C_2 x^2 + C_3$ $\quad f_3(x) = C_6 x^3 + C_7$

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 4. 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)[6].

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 greater than $2^{31} - 1$ for 32-bit arithmetic and greater than $2^{63} - 1$ for 64-bit will produce an integer overflow [15].

Therefore, to reduce the magnitude of encrypted values, one may apply the log function $f_{log}(x) = \log_2 x + C$ 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.

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). $\pm b_0 . b_1 b_2 b_3 \ldots b_{p-1} * B^e$ represents the number $\pm (b_0 + b_1 B^{-1} + \ldots + b_{p-1} B^{-(p-1)})B^e$, $(0 \le b_j < B)$, where b_i is called the significant bits and has p digits, and e is the exponent [9]. With B = 2, 24 bits are used for significant bits for 32 bit arithmetic, 53 bits are used for 64-bit arithmetic [15].

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 log function $f_{log}()$ is applied in the encryption function sequence, in order to avoid precision errors, we define the precision range for x with $f(x) = \log_2 x$ by locating when f(x) loses its precision, i.e., we find x and x+1 where $y_n = \log(x)$, $y_{n+1} = \log(x+1)$ and $y_n = y_{n+1}$.

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

	32 Bit	64 Bit
X	757283_{10}	$2.02967093951847 * 10^{14}{}_{10}$
x+1	757284_{10}	$2.02967093951848 * 10^{14}{}_{10}$
Log(x) = Log(x+1)	19.530474_{10}	47.528239_{10}

From Table 1, after the application of $\log_2(x)$ where x is a 32-bit or 64-bit number, the maximum value of $Int(\log_2(x))$ avoiding precision errors is 19 or 47, respectively. To avoid an early overflow error for attributes with large domain values, the function f_{log} ( ) is always applied first.

We propose the following techniques to control and avoid overflow and precision errors during the encryption process. (See [11] for the detail.)

– After the application of each log function, We maintain an "encrypted-value vector" V_E such that we 1) store $C * Fract(f_{i,log}(\ )) = e_i$ into the encrypted-value vector V_E., (2) provide $w_i = (Int\ (f_{i,log}(\ )\ ) + C')$ as an input to the function $f_{i+1,1}(\ )$, where Int(r) and Fract(r) denote functions that return the integer part of r, and the fractional 24 bits of r as an integer, respectively, and C, C' are constants, and (3)repeat steps (1) and (2) above after application of each $f_{i,log}()$.

– 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 $Left_i$ which indicates the leftmost 19 bits of 32 bit integer z_i and $Right_i$ which indicates the rightmost 13 bits of z_i. Then a different function sequence is applied to each part separately.

Finally, we propose an encryption function sequence $E(x) = f_{k,log}(\ f_{k-1,2}(\ f_{k-2,1}(\ \ldots(\ f_{7,log}(\ f_{6,2}(\ f_{5,1}(\ f_{4,log}(\ f_{3,2}(\ f_{2,1}(\ f_{1,log}\ (x))))))))\ldots)))$, where subscripts are used to number the functions and the function type (linear, quadratic or log). The subscript k of the function $f_{k,log}$ () in E(x) uniquely specifies the encryption function sequence. We refer to the subscript value k as the *index* of the encryption function sequence E.

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 mod 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.

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[14]. 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

[1] Nabil R. Adam, John C. Wortmann: Security-Control Methods for Statistical Databases: A Comparative Study. ACM Computing Surveys 21(4): 515-556 (1989)

[2] Blum, M., Micali, S., “How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits”, SIAM Journal on Computing, 13 (1984), 850-864.

[3] Hakan Hacigumus, Balakrishna R. Iyer, Sharad Mehrotra: Providing Database as a Service. IEEE ICDE 2002

[4] 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

[5] 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)

[6] Melvin J. Maron. Numerical Analysis – A Practical Approach, 1985 Macmillan Publishing Co., Inc.

[7] Multi-level secure database management schemes: software review, CMU Software Engineering Institute, available at http://www.sei.cmu.edu/str/descriptions/mlsdms_body.html

[8] Menezes, A.J, van Oorschot, P.C. and Vanstone, S.A., Handbook of Applied Cryptography, CRC Press, Boca Raton, 1997, page 239.

[9] 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

[10] Ozsoyoglu, G., Singer, D., Chung, S., “Querying Encrypted Databases”, Tech. Report, EECS, CWRU.

[11] Xiaolei Qian, Teresa F. Lunt: A Semantic Framework of the Multilevel Secure Relational Model. IEEE TKDE 9(2): 292-301 (1997)

[12] 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.

[13] Database Management Systems, R. Ramakrishnan, J. Gehrke, McGraw-Hill, 2000.

[14] Eisengerg, A., Melton, J., “Sql: 1999, formerly known as sql 3”, ACM SIGMOD Record, 28(1), 131-138, 1999.

[15] Richard Startz. 8087/80287/80387 – Applications and Programming with Intel’s Math Coprocessors. 1988 Brandy Books, a division of Simon & Schuster, Inc.

[16] Ullman, J.D., “Principles of Database and Knowledge-Base Systems”, Vol 1., Computer Science Press, 1989.

[17] Duminda Wijesekera, Sushil Jajodia: Policy algebras for access control: the propositional case. ACM Conference on Computer and Communications Security 2001: 38-47