2004

Security Issues in Querying Encrypted Data

Murat Kantarioglu

Chris Clifton
Purdue University, clifton@cs.purdue.edu

Report Number:
04-013

SECURITY ISSUES IN QUERYING ENCRYPTED DATA

Murat Kantarciogulu
Christopher Clifton

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #04-013
March 2004

Security Issues in Querying Encrypted Data

Murat Kantarcıoğlu Chris Clifton
Purdue University
Department of Computer Sciences
250 N University St
West Lafayette, IN 47907-2066 USA
+1-765-494-6408, Fax: +1-765-494-0739
{kanmurat, clifton}@cs.purdue.edu

March 31, 2004

Abstract

There has been considerable interest in querying encrypted data, allowing a "secure database server" model where the server does not know data values. This paper shows how results from cryptography prove the impossibility of developing a server that meets cryptographic-style definitions of security and is still efficient enough to be practical. The weaker definitions of security supported by previous secure database server proposals have the potential to reveal significant information. We propose a definition of a secure database server that provides probabilistic security guarantees, and sketch how a practical system meeting the definition could be built and proven secure. The primary goal of this paper is to provide a vision of how research in this area should proceed: efficient encrypted database and query processing with *provable* security properties.

1 Introduction

There has been considerable interest in the notion of a "secure database service": A Database Management System that could that could manage a database without knowing the contents[11]. While the business model is compelling, it is important that such a system be *provably* secure. Existing proposals have problems in this respect; the security provided leaves room for information leaks.

Any method for database encryption that does not meet rigorous cryptography-based standards security must be used carefully. For example, methods that quantize or "bin" values [11] reveal data distributions. Methods that hide distribution, but preserve order [3], can also disclose information if used naïvely. While they may effectively hide values in isolation, using such techniques on multiple attributes in a tuple can pose dangers. We provide more discussion of the successes and potential pitfalls of related work in Section 6; we now give an example of how naïve use of prior proposals can disclose sensitive information.

Suppose a bank is trying to find who is responsible for missing money (e.g., fraud or embezzlement). They have gathered information on suspect employees and customers. Even though much of the information is publicly known (name, size of mortgage, age, postal code, ...), simply revealing *who* is being investigated is sensitive: The appearance that you are accusing a customer of fraud could well lead to a libel suit. Therefore they have encrypted each of the values using an order-preserving encryption scheme. Are they protected?

The answer is probably not. Assume a newspaper wants to know if an individual "Chris" is being investigated. They obtain the encrypted database. They know that the name "Chris" would rank at about 15% of all names – so if it appears in the encrypted database, it will be roughly in that position (the range for a given sample size and probability can be calculated using order statistics). The newspaper can do the same with size of mortgage, age, and other known data about Chris – and with the other employees/customers of the bank. If there is a tuple in the database whose rank on all attributes is close to the corresponding rank of Chris (in the overall dataset), and there is no other tuple among the customers/employees whose ranks are similar, then the newspaper knows that with high probability, Chris is under investigation.

The key problem is that while encrypting a single value using order preserving encryption or a binning scheme may reveal little information, supporting multiple index keys for each tuple reveals a surprising amount. To protect against such naïve misuse of order-preserving, homomorphic, or other such encryption techniques, we propose definitions for what it means for an encrypted database to be secure.

This paper presents a vision for how research enabling a secure database service should proceed: Establish solid definitions of "secure", develop encryption and query processing techniques that meet those definitions, demonstrate that such techniques have practical promise. We start with definitions of security from the Cryptography community that have withstood the test of time. To what extent can we apply these definitions to a "secure DBMS", enabling a proof of security? Section 3 gives security definitions for database and query indistinguishability based on the cryptographic concept of *message indistinguishability*. This leads to a troubling result: prior work in cryptography shows that a secure DBMS server meeting these definitions requires that the cost of every query be linear in the size of the database, making a secure DBMS impractical for real-world use.

Section 4 begins the real contribution of this paper: a slightly relaxed definition that gives probabilistic guarantees of security. For the data itself, security is equivalent to strong cryptographic definitions. An adversary tracing query *execution* could conceivably infer information over many queries, but the quality of the information decays exponentially – by the time enough queries have been seen to infer anything sensitive, the relationship between the early and later queries will have been broken, so the adversary will be unable to infer sensitive data.

In Section 4.2 we show that for this definition, a secure DBMS server with reasonable performance could be constructed. The one caveat is that it requires the existence of a secure execution module: a way of running programs on the server that are hidden from the server. We show how basic query processing operations (select, join, indexed search) can be implemented with a simple secure execution module supporting encryption, decryption, pseudo-random number generation, and comparison; and give sketches of how the operations could be proven to meet our definition of security. This paper addresses read-only queries (select-project-join); extension to insert/update is reasonable, but beyond the scope of the current work.

How do we get such a secure execution model? *Program obfuscation*, providing a program whose execution reveals no information to the server, has been proven impossible for several classes of program[6]. We provide evidence that if possible at all it is unlikely to be efficient for a secure DBMS. Fortunately, there is another solution, implementing the secure execution module in tamper-proof hardware. Such hardware exists; one example meeting our requirements is IBM's [12]. Section 5 shows how to use such a module to implement a system safe from software-driven attacks; the specifications and evaluation of the hardware tell us the protection provided against electronic/physical attacks.

2 Background and Definitions from Cryptography

The cryptography community has developed solid and well-regarded definitions for securely encrypting a message. The result of this research is that secure encryption should hide any partial information about the data: one should not be able to distinguish between encryptions of two different messages of the same length. Consider the following scenario where a stock trader sends buy or sell messages for a particular stock. An adversary knows messages are either buy or sell, but indistinguishability guarantees the adversary will have no clue which one was sent.

A formal definition of indistinguishable private key encryption, from [8], is:

Definition 2.1 Indistinguishability of encryptions[8]
An encryption scheme with efficient key generation algorithm G, efficient encryption algorithm E and decryption algorithm D where $D_k(E_k(x)) = x$ for secret key k has indistinguishable private key encryption if for every polynomial-size circuit family $\{C_n\}$, every polynomial p, all sufficiently large n and every $a, b \in 0, 1^{poly(n)}$ with equal length.

$$|Pr\{C_n(E_{G(1^n)}(a)) = 1\} - Pr\{C_n(E_{G(1^n)}(b)) = 1\}| < \frac{1}{p(n)}$$

The probability in the above terms is taken over the internal coin tosses of algorithms G and E.

Intuitively the definition states that any adversary with computing power comparable to the polynomial-size circuit family will not be able to predict whether a given ciphertext is the encrypted form of a or b significantly better than a random guess.

The same key is often used to encrypt many messages. The preceding definition ensures that no two messages can be distinguished, but it does allow the *distribution* of messages to be learned (e.g., an adversary could learn that 75% of orders were of one type and 25% were the other, even if it didn't know which was buy and which was sell.) For multiple message, the cryptography community has a definition that ensures that even the distributions are not revealed. The formal definition from [8] is:

Definition 2.2 Indistinguishability of encryptions: multiple-message case[8] *An encryption scheme with efficient key generation algorithm G, efficient encryption algorithm E and decryption algorithm D where $D_k(E_k(x)) = x$ for secret key k has multi-message indistinguishable private key encryption if for every polynomial-size circuit family $\{C_n\}$, every polynomial p, all sufficiently large n and every $\bar{a} = (a_1, a_2, \ldots, a_{t(n)})$, $\bar{b} = (b_1, b_2, \ldots, b_{t(n)})$ where $\bar{a}, \bar{b} \in 0, 1^{poly(n)}$, following inequality holds*

$$|Pr\{C_n(E_{G(1^n)}(\bar{a}) = 1\} - Pr\{C_n(E_{G(1^n)}(\bar{b})) = 1\}| < \frac{1}{p(n)}$$

where $E_{G(1^n)}(\bar{a})$ is defined as
$(E_{G(1^n)}(a_1), E_{G(1^n)}(a_2), \ldots, E_{G(1^n)}(a_n))$

Any encryption scheme that satisfies the above definition will hide the distribution of the transmitted encrypted messages. Our definition of securely encrypting a database is based on these definitions. The first captures general database encryption, the second is applicable to encryption of individual tuples or field values.

Fortunately, the cryptography community has a method for extending many ciphers providing message indistinguishability to provide multi-message indistinguishability. The counter-based CTR encryption mode given in Algorithm 1 enables a block cipher meeting Definition 2.1 to meet 2.2. (It also supports encryption where the block size exceeds that of the block cipher.) The idea is to choose a unique number (counter) for each block, and encrypt that unique number. The resulting encryption is then xor-ed with the actual message block. The encrypted message consists of the counter (in plain text), and the message. Assuming the underlying block cipher is a pseudo-random permutation (DES is presumed to be such a permutation), this method satisfies Definition 2.2. The key idea is that two identical messages (or identical blocks in a message) will be xor-ed with different values, since the masking encryption will be from a different counter.

Algorithm 1 Counter-based CTR mode Encryption

```
Encryption:
Require: plain text $x$ and initial counter value $C$.
  Divide $x$ into $n$ $k$-bit blocks
  for i=1 to n do
    $y_i = E_k(C+i) \oplus x_i$
  end for
  return $(C, y_1 y_2 \dots y_n)$
Decryption:
Require: ciphertext $y = (C, y_1 y_2 \dots y_n)$
  for i=1 to n do
    $x_i = E_k(C+i) \oplus y_i$
  end for
  return$(x_1 x_2 \dots x_n)$
```

3 Implausibility of a Fully Secure Database Server

Encryption schemes are defined to be secure if and only if the ciphertext reveals no information about the plaintext. We now use the security definitions from cryptography to define what it means to "securely" encrypt a database table and securely query the encrypted data.

3.1 Security Definitions for Encrypted Database Tables

As mentioned above, given any two pairs of ciphertexts and plaintexts of the same length, it must be infeasible to figure out which ciphertext goes with which plaintext. This means that any two database tables with the same schema and the same number of tuples must have indistinguishable encryptions. To be more precise, we now give a database-specific adaptation of the definitions stated in Section 2.

Definition 3.1 *An encryption scheme (G, E, D) for database tables; consisting of key generation scheme G, encryption function E, and decryption function D; has indistinguishable encryptions if for every polynomial-size* *circuit family $\{C_n\}$, every polynomial p, and all sufficiently large n, every database R_1 and $R_2 \in \{0,1\}^{poly(n)}$ with the same schema and the same number of tuples (i.e., $|R_1| = |R_2|$):*

$$|Pr\{C_n(E_{G(1^n)}(R_1)) = 1\} - \\ Pr\{C_n(E_{G(1^n)}(R_2)) = 1\}| < \frac{1}{p(n)}$$

The probability in the above terms is taken over the internal coin tosses of algorithms G and E.

This definition says that if we try to construct a polynomial circuit for distinguishing any given encrypted database table R_1 (i.e., the circuit will output one if the encrypted form belongs to R_1, else it will output zero), the circuit will have a success probability that is at most slightly better than a random guess.

To clarify the meaning and impact of this definition, we give an example of a plausible but insecure encryption of a database table.

Example 3.1 *Define an encryption scheme for a database as follows: G randomly outputs a key for a particular block cipher, E encrypts every field of each tuple with the same key using a block cipher algorithm (e.g., DES), and D decrypts the every field using the same cipher and key. Even though the given block cipher algorithm is secure, we can distinguish encryptions of the following two database tables with probability one. Assume that R_1 and R_2 have a schema (a, b) where both a and b are k bit numbers, and both have one tuple. If R_1 has a tuple (x, x) and R_2 has a tuple (y, z) $(y \neq z)$, a simple (polynomial) circuit that compares the encrypted values of the first and second fields $(E(a) = E(b))$ will return true for the encrypted R_1 and false for the encrypted R_2, distinguishing the two.*

3.2 A Secure Method for Encryption of Database Tables

While one solution to the problem of Example 3.1 is to simply encrypt the entire database as a single message this would prevent any meaningful query processing (the entire encrypted database would have to be returned to the querier to enable decryption). Fortunately, we can use Counter-based CTR

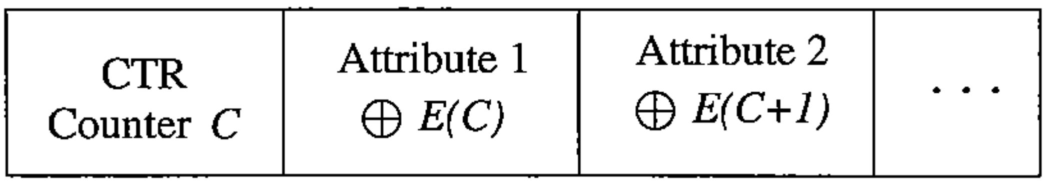

Figure 1: Encrypted tuple structure

mode to meet Definition 3.1 while still encrypting at the individual fields in a tuple independently. The idea is that each tuple consists of a counter and encrypted fields, as described in Section 2. Figure 1 shows an example tuple encrypted in this manner. Since identical field values will now be xor-ed with different values, the fact that they are identical (or any other relationship between them) will be hidden, alleviating the problem of Example 3.1 and in fact meeting Definition 3.1.

3.3 Database Indistinguishability in the Presence of Queries

Much of database research has concentrated on efficient processing of queries. We would like to maintain this efficiency even if the data is encrypted. Prior proposals for querying encrypted data do not meet Definition 3.1 if an adversary is allowed to view data access patterns; this will be discussed in Section 6. This is not just a problem of poor use of encryption. What we really need to ensure is that not only is the encrypted database itself secure, but that the act of processing queries against the database does not reveal information. Unfortunately, achieving such security is at odds with efficient query processing. We now give a definition of secure database querying based on a model from the cryptography community, and show that the only way to meet this strict definitions is to access the entire database for each query. In Section 4.1 we will build on these definitions to give a slightly weaker (but still semantically meaningful) definition supporting more efficient queries. In our current discussion, we assume that data resides on single server and do not consider potential gains due to the replicated data.

3.4 Database Queries as a Special Case of Private Information Retrieval

We now give an alternative definition based on comparison of queries. We still require that tuples be indistinguishable (Definition 3.1), and also require that two *queries* be indistinguishable (e.g., the queries are encrypted). The idea is that if we can't tell tuples or queries apart, we don't really gain information from processing the queries. Unfortunately, this leads us to a result where full table scan is required.

The definition comes from Private Information Retrieval (PIR), which protect the query from disclosure. The server knows the data, but should learn nothing about the query.[7] A PIR server must maintain query privacy, and ensure that the query issuer gets the correct result.

Why do we want the privacy of the user query be protected? The problem is that if the server knows the query, knowing just the size of the result reveals information about the database. For example, if server knows that $\sigma_{R.a1=300}(R)$ returns three tuples, then server will have the knowledge of those tuples' $a1$ fields. One important thing to note is we should only require query indistinguishability for queries that have the same result size. Otherwise we would need to set an upper bound on query result size (the entire database if we want to support full SQL), and transmit that much data for every query – the actual result size would distinguish queries.

We now formally define the correctness and the privacy requirements described above.

Definition 3.2 *(Correctness)*
Assume database D is stored securely on a server w.r.t Definition 3.1. Let $E(D)$ be the securely encrypted database and let Q be a query issued on the database. A query execution is said to be correct if given $(Q, E(D))$, an honest server provides a result enabling the query issuer to learn $Q(D)$.

The correctness definition implies that if the server follows the protocol, the query issuer will get the correct result.

Privacy must hold even for a dishonest server:

Definition 3.3 *(Privacy)*
For every query pair Q_i, Q_j that run on the same set of tables over D and have the same size results, the messages m_{Q_i}, m_{Q_j} sent for executing the queries are

computationally indistinguishable if for every polynomial-size circuit family $\{C_n\}$, every polynomial p, all sufficiently large n, m_{Q_i} and $m_{Q_j} \in \{0,1\}^{poly(n)}$,

$$|Pr\{C_n(m_{Q_i}) = 1\} - Pr\{C_n(m_{Q_j}) = 1\}| < \frac{1}{p(n)}$$

The probability in the above terms is taken over the internal coin tosses of the query issuer and the server.

This privacy definition implies that whatever the server tries to do, it will not be able to distinguish between two different queries run on the same set of tables and returning the same size results. For example, if $Q1 = \sigma_{a1=300}(R)$ returns 100 tuples and $Q2 = \sigma_{a1=100}(R)$ returns 100 tuples, there is no way for the server to predict which of the two is executed more effectively than a random guess.

We can define a secure query execution as one that runs on securely encrypted data and satisfies Definitions 3.3 and 3.2.

We will show that even for queries that are running on a single table, we need to scan the entire table.

We first prove that given a set of queries on a particular table with t, if there exists a query that must access at least v tuples, then we can distinguish it from a query that occasionally accesses fewer than v tuples. Second, we show that for any admissible query result size t, there exists a query which requires the scan of the entire database.

Lemma 3.1 *Let S_t be queries that run on table R with result size t, and let us assume that there exists a query Q_1^t that needs to access at least v tuples for correct evaluation. Let Q_2^t be an element of S_t that needs to access at most $v-1$ tuples with probability greater than $\frac{1}{p(n)}$. Then there exists a polynomial-circuit family C_n that can distinguish them with non-negligible probability.*

Proof. We define C_n as follows. Given the messages exchanged during the execution of the query, the circuit will count the number of the tuples accessed. If it is $\geq v$, C_n will output 1; otherwise it will output zero. Note that C_n only does a simple counting, therefore is polynomial in terms of the input size. Now let us calculate the probability $P = | Pr\{C_n(m_{Q_1^t}) = 1\} - Pr\{C_n(m_{Q_2^t}) = 1\} |$.

$$\begin{aligned}
P &= \mid Pr\{C_n(m_{Q_1^t}) = 1\} - Pr\{C_n(m_{Q_2^t}) = 1\} \mid \\
&= \mid 1 - Pr\{C_n(m_{Q_2^t}) = 1\} \mid \\
&= \mid 1 - Pr\{\text{more than } v-1 \text{ tuples accessed}\} \mid \\
&> \mid 1 - (1 - \frac{1}{p(n)}) \mid \\
&> \frac{1}{p(n)}
\end{aligned}$$

Again, note that the probability is taken over the internal coin tosses of the query issuer and the server; it does not depend on the database values.

Since P is bigger then $\frac{1}{p(n)}$ we can conclude than C_n distinguishes the above queries with non-negligible probability. □

We now show that the queries needed by the above definition exist.

Lemma 3.2 *For any given result size t, there exists a query that needs to access the entire table.*

PROOF. Since the result must be encrypted to preserve security (otherwise all queries would have to return the same result to avoid being distinguished), the resulting set size must be a multiple of the cipher block size k of size, up to the size of the table. Let R have n tuples with a attributes blocked into u blocks of size k as defined in Section 5.1.1. Here without loss of generality, we assume that each attribute is k bits long, therefore u is equal to a.

Let assume that *id* field added to the database is also k bit long. So for each admissible size t where t is the multiple of k and less than $k * n * a$, we can define a query that needs to access the entire database as follows.

$$\begin{aligned}
Q_1^t &= \bigcup_{i=1}^{\lfloor \frac{t}{kn} \rfloor} \pi_{a_i}(R) \\
&\quad \cup \pi_{a_1}(\sigma_{id<(t \bmod kn-1)*a}(R) \\
&\quad \cup avg(\pi_{a_1}(R))
\end{aligned}$$

The above query simply gets the average of a single attribute to make sure that query needs to access the entire table, and pads the result set to make sure that result size is t. (Since we have not specified a value for k, this generalizes to any block size, including 1.) □

Using the above lemmas, we can now prove the following:

Theorem 3.1 *A query execution that is secure in the sense of Definitions 3.3 and 3.2, even for queries known to access a particular database table, must scan the entire database table non-negligibly often.*

PROOF. For the set of queries returning a result of size t, at least one must require full table access (a construction is given in Lemma 3.2), if not then not all queries would satisfy the correctness Definition 3.2. We can now build a distinguisher for any query that requires less than full table access (formal proof in Lemma 3.1). Since at least one query in t requires full table access, if any requires less than full access a non-negligible portion of the time, the distinguisher will be able to tell the two apart. Such a distinguisher contradicts Definition 3.3. □

3.5 Database Queries as a Special Case of Software Protection

More generally, the cryptography community has produced the concept of *oblivious RAM*[10]: a method to obscure the program even to someone watching the memory access patterns during execution.

In their main result, they show that if a program and its input with total size y uses memory size m and has a running time t, then it can be simulated by using $m \cdot (\log_2 m)^2$ memory in running time $O(t(\log_2 t)^3)$ without revealing the memory access patterns of the original program (assuming $t > y$). In other words, they provide a solution such that the distribution of memory accesses does not depend on input. This implies that execution of queries can be made indistinguishable if they access the same number of tuples and have the same result size.

Unfortunately, even under this relaxation, we will not achieve much improvement in terms of efficiency. They show that the lower bound on the oblivious simulation cost is $max\{y, \Omega(t \log t)\}$. In their model, the input y includes everything to be protected, including the program and data. The database would be modeled as part of the program, so the size of the database and the program will be a lower bound for number of memory access. This still implies a full table scan. At this point, we would like to stress that we are considering running a query in isolation – batching queries could improve throughput (a full scan for each batch), but would prevent effective ad-hoc or interactive querying.

4 Plausible Definition for a Secure Database Server

In the previous section, we showed that any strict security and privacy requirement force us to scan entire database tables. The previous definitions' main problems are that they try to preserve indistinguishability even if a server can look at tuple access patterns. What we need is a definition that allows revealing the access patterns for a tuple, enabling more efficient query processing.

4.1 Definition

If the data and queries are encrypted, and the encryption satisfies multiple-message indistinguishability (e.g., Definition 3.1), then the ability to distinguish between queries or tuples carries little information, especially if the ability to trace tuple access between queries is limited. Using this observation, we give a new definition that guarantees some level of privacy while allowing a higher degree of efficiency than the previous examples.

First, we define a minimum set of support tuples for each query: the tuples that must be accessed to compute the query results. We then only apply query indistinguishability to queries that have the same support tuple set.

Definition 4.1 ***(Min support set)***
Let query Q be defined on tables $R_1, R_2, \ldots, R_n$. Let S be the set of elements in $R_1 \times R_2 \times \ldots \times R_n$. A set $S \subset (R_1 \times R_2 \times \ldots \times R_n)$ is a min support set for Q if $Q(S) = Q(R_1 \times R_2 \times \ldots \times R_n)$, and S is the smallest such set for which this is true.

Example 4.1 *Assume we have two tables: $R_1(a1, a2) = \{(1, a), (2, b), (3, c)\}$ and $R_2(a1, a2) = \{(1, 2), (2, 3), (3, 4)\}$. Let $Q = \Pi_{R_1.a_1}(\sigma_{R_1.a_1 = R_2.a_2}(T))$. $Q(R_1 \times R_2)$ returns the same result as $Q(\{(1, a, 1, 2), (2, b, 2, 3), (3, c, 3, 4)\})$, and these three tuples are the smallest such set.*

Using this, we can now give a definition that ensures nothing is disclosed by watching query processing except the size of the result and what tuples were processed in arriving at the result.

Definition 4.2 *(Query Indistinguishability)*
For every query pair Q_i, Q_j on the same set of tables, with the same result size and min support set, the messages m_{Q_i}, m_{Q_j} sent for executing the queries are computationally indistinguishable if for every polynomial-size circuit family $\{C_n\}$, every polynomial p, all sufficiently large n, and m_{Q_i} and $m_{Q_j} \in \{0,1\}^{poly(n)}$,

$$\mid Pr\{C_n(m_{Q_i}) = 1\} - Pr\{C_n(m_{Q_j}) = 1\} \mid < \frac{1}{p(n)}$$

This, combined with Definition 3.1, guarantees that all an adversary can do is to trace the tuples accessed during query execution, and possibly relate that to result size. As this could disclose information over the course of many queries, we also give the following definition, requiring that the confidence in tracing tuples drops over time:

Definition 4.3 *(Three Card Monte Secure)*
A database is c-secure if given a query Q with min support set T, the probability that a server trying to track $t \in T$ can do so correctly is $< \frac{1}{c(k+1)} + \frac{|T|}{|DB|}$, where k is the number of times t has been accessed since completion of Q.

The key to this definition is that an adversary's confidence that they know which tuples Q accessed will decrease over time. (Formal proof of the efficacy of this definition of security is beyond the scope of this paper.) With high probability any useful information inferred from tracking tuple access will be incorrect.

Definition 4.4 *We consider a database to support secure query processing if it meets Definitions 3.1, 3.2, 4.2, and 4.3.*

We now describe how to construct a database server meeting these definitions.

4.2 Requirements for a Database Server

Methods that allow equality test of encrypted tuples, or field values in the tuples, violate Definition 4.4 because tuples can be distinguished. The problem is that if the tuples are truly indistinguishable, the server will be unable to do any query processing beyond "send the entire table to the client" – any meaningful query processing requires distinguishing between tuples. If the tuples can be distinguished, then they can be tracked over multiple queries, disclosing information in violation of Definition 4.3.

However, if we support a few simple operations that are "hidden" from the server, we can meet Definition 4.4. The key idea is that operations that must distinguish between tuples (e.g., comparing a tuple with a selection criteria) occur by decrypting and evaluating a tuple in a manner invisible to the server. The tuples accessed are then re-encrypted and written back to the database, but not necessarily in the same order. This prevents the server from reliably tracking the tuples accessed across multiple queries. We don't want to send tuples back to the querier to do this. However, assume the existence of a module capable of the following:

1. decrypt tuples,
2. perform functions on two tuples,
3. maintain simple (constant-size) history for performing aggregate functions,
4. generate a new tuple as a function of the inputs, and
5. maintain a constant-size store of tuples,
6. perform a counter-based CTR mode encryption of the new tuple.

The module may return an (encrypted) tuple to write back into the location most recently read from – but this is not necessarily the most recently read tuple (making tracking difficult). (Such swapping was proposed for PIR in [5], here we amortize the cost as opposed to periodically shuffling off-line.) It also optionally returns a tuple that becomes part of the result. The module also returns the address of the next tuple to be retrieved. Assuming such a module can perform these operations while obscuring its actions and intermediate results from the server, we can construct a machine meeting Definition 4.4.

The idea is that the database is encrypted as in Section 3.2. An encrypted catalog (in a known location) contains pointers to the first tuple in each table or index. The secure module decrypts the query, reads the catalog to get the location of the first tuple of the relevant tables/indexes, then begins processing. We first show how individual relational operations can be securely performed using the above module. We give a sketch of the proof of security of each using a simulation argument (as used in Secure Multiparty Computation[9]) – the idea is that given the results (min support set and result size), the server is able to simulate the actions of the secure module. If it is able to do so, then all queries on that set and result size must be indistinguishable from the simulator, and thus indistinguishable from each other. (These are sketches; full details require probabilistic simulation proofs to meet Definition 4.3.) We will then discuss composing operations to perform complex queries.

Selection makes use of the fact that we have some memory hidden from the server (adversary). The secure module keeps the results until the local memory is partially filled. At this point, after each new tuple is read, one of the cached result tuples *may* be output to the server. This decision is a random choice, with the probability based on the estimated size of the results relative to the estimated number of tuples read.

Formally, assume that the estimated number of tuples that need to be read to execute the query is t, the estimated result size is r, and the local memory size is m. The secure module reads the first $(t/r) \cdot (m/2)$ tuples, caching the results in local memory. At this point, for every tuple read, with probability r/t one of the cached result tuples is given to the server. When the query is complete, the remaining cached tuples are given to the server for delivery to the client.

Theorem 4.1 *Provided that queries contributing to the result are (approximately) uniformly distributed across all tuples read, the above process meets Definition 4.4. for full table scan selections.*

Proof Sketch. Using a simulation argument, we assume the simulator for the server is given t and r (since these will be known at the end of the query.) m is public knowledge. The simulator can thus compute $(t/r) \cdot (m/2)$. After this many tuples have been read, the simulator begins creating result tuples. Since the tuples are encrypted using pseudo-random encryption, the simulator just uses a counter and an appropriate length random string of bits to simulate a tuple. By arguments on the strength of encryption the simulated output tuples and re-encrypted tuples are computationally indistinguishable from the real execution. After each tuple is read, a simulated result tuple is created with probability t/r. When all tuples have been read, the simulator creates the remaining result tuples (so the total is r.)

Since the result tuples can be simulated using this approach, and the simulator decides when to create the result tuple in exactly the same fashion as the real algorithm decides when to output a result tuple, the simulator is (computationally) indistinguishable from the actual selection. This shows that it meets Definition 4.2.

Definition 4.3 is more difficult. This relies on the assumption of approximately uniform distribution. Because of this, the a-priori probability that a given tuple is in the first t/r tuples is high, so little information is revealed by disclosing that the first result occurs in the first $(t/r) \cdot (m/2)$ tuples. □

This approach does fail when the distribution of which tuples contribute to the result to all query tuples is skewed. For example, if none of the first $(t/r) \cdot (m/2)$ tuples cause a result tuple to be generated, the algorithm will be unable to begin outputting result tuples "on schedule". Thus the server can make an improved estimate of the probability that a tuple contributes to the result. In the worst case (e.g., only the last r tuples contribute to the result), this probability approaches 1.

Queries that generate most results based only on the first tuples read are unlikely. Queries that generate results only after reading most or all of the tuples are more common: aggregation, indexed search. However, these queries will generally return a small number of results. If $r \leq m/2$, the secure coprocessor will not be expected to produce results until all tuples are read, so Theorem 4.1 holds. Queries where the results are highly skewed should be processed using an indexed selection anyway (to efficiently access only the desired tuples.)

Indexed Selection can be done using a method developed for oblivious access to XML trees[14]. Nodes are swapped, re-encrypted, and written back to the tree. The key idea is that each time a node is read, $c-1$ additional nodes are read – one of which is known to be empty. All the nodes are re-encrypted and written, with the target written into the empty node. When the nodes are written, the original is written into the empty. This proceeds in levels: The first two levels are read, the location of the second level empty is determined, and the parent is updated to point to the previous empty node, and the first level written. The third level is read, second level parent updated, etc.

Theorem 4.2 *The algorithm of [14] satisfies Definition 4.4.*

PROOF SKETCH. Definition 4.2 is satisfied because queries with the same min support set will follow the same path to the same leaf. The random choice of $c-1$ additional nodes comes from the same distribution, and are thus indistinguishable. Likewise, encryption and rewriting is indistinguishable by arguments based on strength of encryption.

Definition 4.3 is satisfied because of the swapping. Each time a node is accessed, it is placed in a new location. However, since c locations have been read and written, and are indistinguishable to the server, the probability that the server can pick which of the c locations the node is in is $1/c$.

The next time the node is read, it is again placed in one of c locations, with which one unknown to the server. The best the server can now do is guess that it is in one of the $2c$ locations. (Access to other of the original $2c$ locations may confuse the server, causing it to guess more than $2c$ locations, but we are guaranteed at least $2c$.) This continues, with each access to the tuple causing an additional c decrease in the server's best guess, giving our $1/c(k+1)$ target.

The only problem is that the randomly chosen set of "masking" locations may include locations previously used. This is inherent in a finite database - the best we can do is $1/|DB|$. This is the reasoning behind the $|T|/|DB|$ "floor" factor in Definition 4.3.

$\square$

This analysis is based on a query returning a single tuple. Extension to range queries is straightforward.

Projection is straightforward. The comparison function simply returns $E(\Pi\ tuple)$ rather than $E(tuple)$. Knowing the length of a projection from the encrypted result, the simulator can randomly generate an equivalent-length string that is computationally indistinguishable from the real encrypted result. In particular, note that a "null" projection (e.g., "select * from table") is indistinguishable from any other projection producing tuples of the same size – the only way to distinguish selection from projection is the fact that some tuples are "removed" by a selection.

Join can be either repeated full-table scan selection (nested loop join) or indexed selection (index join). To perform a join, the module first requests a tuple from one table, then from the second table. Both are decrypted, the join criteria is checked, and if met the joined tuple is stored for output. Assuming a reasonably uniform distribution of tuples meeting the join criteria, or a small number of tuples meeting the join criteria, the proof follows that of Theorem 4.1. A similar argument holds for an index join. Again, we need a reasonably uniform distribution of tuples meeting the join criteria. The swapping in the index search prevents too much tracking between tuples, and caching the results allows the resulting tuples to be output at a constant rate.

Set operations are straightforward, except for duplicate elimination. Union is simply two selections. Intersection is a join. Set difference is again similar to a join, but output only occurs if after completion of a loop (or index search), a joining tuple is not found.

Duplicate elimination could reveal equality of two tuples. This is more than simply "does it contribute to the result", and thus violates Definition 4.4. One solution is to replace duplicates with an encrypted dummy tuple. The client thus gets a correct result by ignoring the dummy tuples, at the cost of increased size of the result.

4.3 Discussion

Real query processing requires combining these methods to form a query tree/plan. A simple approach would reveal the query tree and plan to the server. At first glance, this seems excessive. However, simply given the access patterns it is often possible to make a good guess as to the query plan: If two tables are accessed (e.g., tuples of different sizes), it is probably a join; a table being accessed and returning fewer tuples is a selection; logarithmic access frequencies represent a tree-based index. Rather than trying to pretend such information is hidden, we suggest explicitly revealing it. This allows the server to perform rule-based query optimization, prefetching, and likely other types of performance enhancements.

We believe that meaningful improvements in security either require a query processor that can hide substantial (and non-constant) intermediate state from the server, or run afoul of the problems of Section 3. Any method with constant-space "hidden storage" will either require full table scan for all queries, or will reveal information equivalent to the above *for some sequences of queries.*

5 Architecture for a Secure Database Server

The implementation just described depends on the ability to execute a simple function on the server, without the server seeing the execution. This problem is known as *program obfuscation*, and has been the object of some study. The results are not encouraging. General program obfuscation has been shown to be impossible for large classes of functionalities[6]. While we have not proven that such an obfuscated program is impossible for our "decrypt, compare, and re-encrypt" function, if possible at all it is likely to pose a high computational cost.

Fortunately, there is an alternative: tamper-resistant hardware. This enables execution of a function while hiding the execution from the server, as needed to meet our definition. This has been proposed as a solution to Private Information Retrieval[5]; we show how it also applies to general database query processing. In addition, such hardware already exists. One example is the IBM 4758 Cryptographic Coprocessor.[12] This is a single-board computer consisting of a CPU, memory and special-purpose cryptographic hardware contained in a tamper-resistant shell; certified to level 4 under FIPS PUB 140-1. When installed in the server, it is capable of performing local computations that are completely hidden from the server; tampering is detected and clears internal memory.

How does this solve our problem? Using public key cryptography, the client can verify that the server contains an approved tamper-proof coprocessor, and provide the coprocessor with the key for decrypting the tuples in the database. The client has a key for the coprocessor, allowing it to clear the coprocessor and load the key and program to be executed. The coprocessor can now perform the "decrypt, compare, and re-encrypt" function. Any attempt by the server to take control of (or tamper with) the coprocessor, either by software or physically, will clear the coprocessor, thus eliminating any decrypted view of the tuples. (The same holds true of another client executing a query; taking control of the coprocessor clears old information.)

Further details on using a tamper-proof coprocessor to securely meet our requirements are given below.

5.1 Considerations for Practical Implementation

Care must be taken to ensure that a tamper-proof coprocessor is in fact used in a secure way. The basic process of setup for running a query is as follows.

1. The client creates a query execution program that includes the database key, and encrypts it with the public key of the coprocessor. This query execution program is fixed, and may be stored at the server (use of a checksum along with the fact that is encrypted presents the server from

tampering with the program.)

2. The client creates a query (including a checksum to prevent tampering), encrypts with the database key stored in the (encrypted) query execution program, and sends it to the server.

3. The server delivers the query execution program to the secure coprocessor, and instructs it to reset, then decrypts the program with its private key and executes it. This particular private key can only be used as part of such a "reset, decrypt, and load" command, ensuring that the database key cannot be decrypted unless the accompanying program for query execution is run.

4. The server provides the query to the coprocessor, which decrypts it and verifies the checksum.

5. The coprocessor now begins requesting tuples from the server. As each is received, it is decrypted, compared with appropriate query terms, and any result is re-encrypted before being returned to the server (as described in Section 4.2).

6. On query completion, the co-processor sends a "done" message to the server and resets (clearing its memory.)

7. The server returns the results to the client, which decrypts them with its database key.

The key to the security of this protocol is that the server never sees the key used to encrypt the database / queries, or any data that is not encrypted with that key. The database key is stored in two places: At the client[1]; and at the server, but encrypted with the coprocessor's public key. The only way the server-stored key can be decrypted is after resetting the coprocessor, and while executing the (client-provided and verified) program that executes queries. Thus the database key is only accessible to code provided by the client, running in an environment that is not visible to the server.

While the secure coprocessor is somewhat at the mercy of the server, the only way the server can modify the code executed by the coprocessor is by resetting and reloading the code, which clears the coprocessor (including the

[1]The client is presumed to be trusted. In practice, key management could be handled with a secure subsystem such as the Trusted Computing Group chip[17].

a1	, a2 ,	a3
b1	, b2 ,	b3
c1	, c2 ,	c3

Table 1: Original Table

database key.) Any attempt to modify the encrypted code will prevent validation of the code, and the coprocessor will refuse to run the code. Thus the coprocessor can be guaranteed to securely provide the functionality required in Section 4.2.

Note that there is nothing to prevent multiple clients from sharing the same server; the only limitation is that only one may use the secure coprocessor at any given time. This poses some interesting challenges for concurrency control, but these are beyond the scope of this paper.

The tamper-proof hardware solution has other advantages. Since secure coprocessors are typically designed for use in setting where data is encrypted, they will typically contain special-purpose encryption/decryption hardware, providing improved performance. For example, the IBM 4758 includes hardware support for DES, modular math to support public-key encryption, and random number generation. Raw throughput can achieve 23.5MB/sec for DES.

5.1.1 Encryption Optimizations

In practical terms, each encrypted value needs to be of the given block cipher size. We can easily combine, split, or pad attributes to achieve this, depending on our requirements. For example, if the each attribute is 32 bits long and cipher operates on 64 bits blocks, we can encrypt them pair by pair and pad the last block with zeros. Let u be the minimum number of blocks needed to encrypt each tuple. We encrypt the i^{th} tuple's j^{th} block B_{ij} as $E_K(i * u + j) \oplus B_{ij}$, where K is the encryption key.

Since tuples will be processed alone, we will add a new field to store $i * u$ in every tuple. For example, given the original database Table 1 where each attribute is 32 bits, the encrypted table using a 64-bit block cipher is shown in Table 2. $\circ$ denotes the concatenation operation and 0^{32} denotes a 32 bit string of all zeros.

It is clear from the example that given the key, we can decrypt any part of

0,	$E_K(0) \oplus (b1 \circ b2)$,	$E_K(1) \oplus (0^{32} \circ b3)$
2,	$E_K(2) \oplus (c1 \circ c2)$,	$E_K(3) \oplus (0^{32} \circ c3)$
4,	$E_K(4) \oplus (a1 \circ a2)$,	$E_K(5) \oplus (0^{32} \circ a3)$

Table 2: Encrypted Table

the table independently. Another advantage is that we can decrypt counters in advance: assigning the counters in order is not a problem (they just need to be unique), so it may be possible to guess (and decrypt) counters in parallel with a block of tuples being retrieved from disk.

It has been suggested that decrypting data as part of a query would pose an unreasonable cost[15]. Based on benchmarks published by IBM[13], simple decryption of a single tuple would take several milliseconds. In practice, a system would probably encrypt/decrypt at the page rather than tuple level (assuming sufficient internal memory). The re-encryption/swapping would then also occur at the page level. In addition, the decryption of CTR counters can be done even before tuples are read, leaving only a simple xor operation for each tuple. Given this, we believe it is feasible to implement decryption/encryption of the CTR counters at speeds that approaches the peak 23.5MB/sec DES throughput achieved by the IBM 4758, and the xor operation would not introduce a significant delay. This approaches disk speeds, allowing a secure database server without a significant performance penalty.

6 Prior work in Secure DBMS Servers

There have been several efforts to develop systems for managing encrypted data. In [4], Ahituv, Lapid, and Neumann addressed the problem of managing *updates* (but not queries) on encrypted data. They require that which tuple is to be updated be known, but the value in that tuple is secret. They developed a method for additive updates in two cases: One where the value to be added is known to the server, and one where both are unknown. Their approach is based on homomorphic encryption: $E(a+b) = E(a)+E(b)$. The idea is that using homomorphic encryption, we don't need to decrypt values to add them.

The issue of indexing encrypted data is addressed in [11]. The key idea is that values are partitioned into buckets. A query is translated into operations on these buckets; the server returns all results from the appropriate buckets. One problem with this technique is that buckets must be of a fixed size to avoid violating Definition 3.1 even in the absence of queries. While feasible for an initial database, maintaining this after insertions is not feasible. Thus the relative size of buckets reveals information about the distribution of the data. A second problem is that relationships between fields in a tuple are revealed, as in Example 3.1. Only *probabilistic* relationships are learned, but as the bucket size decreases the probability of learning such a relationship increases. Large buckets are not a feasible solution, as an equ-join becomes a cross-product of buckets, with the result size (and client effort) growing rapidly with larger buckets (as shown in [11]).

Ozsoyoglu et al. [15] also addressed running queries on encrypted databases. They suggest heuristic encryption methods that preserve some relationships among the data, such as order or the difference between attributes. Such order preserving encryption functions cannot satisfy database indistinguishability (Definition 3.1). Consider databases with two attributes and n tuples. The first database contains tuples $< 1, 1 >, < 2, 2 >, ...$, the second $< 1, n >, < 2, n-1 >, ...$ If the encryption preserves order, we can always distinguish between the two sequences with probability 1, as the tuples sort the same on both attributes in database 1, and sort in opposite order on the two attributes in database 2. Similar arguments can be used for encryption that preserves difference between the messages. While data values may be protected from direct inspection, a determined server with some additional knowledge (e.g., a history of queries) may be able to significantly compromise security.

Careful definitions of the system environment *can* be secure using these methods. A method for order-preserving encryption that hides distributions, and an architecture for its use, is presented in [3]. The key is in their assumptions: The database software is trusted (the adversary only has access to the encrypted database, not the running system), and only one attribute in each relation uses order-preserving encryption. While secure within the assumptions, it does not meet our goal of a "secure database service".

A key distinction between our work and that described above is that we address what the server learns from processing queries. Statistical database work has shown that a sequence of queries can reveal information beyond that revealed by any single query[1]; this is just as true for queries on encrypted data. Private Information Retrieval has addressed this issue [7], but under the assumption that the data is known to the server and the query must be kept private. The results are discouraging; the data access lower bound for single server is the entire database. However, by encrypting both data and queries, we can obtain reasonable levels of security and avoid the impossibility results of Private Information Retrieval.

7 Conclusions

The idea of a database server operating on encrypted data is a nice one: It opens up new business models, protects against unauthorized access, allows remote database services, etc. Achieving this vision requires compromises between security and efficiency. We have shown that a server that would be considered secure by the cryptography community would be hopelessly inefficient by standards of the database community. Efficient methods (e.g., operations on encrypted data) can not meet cryptographic standards of security.

We have given a definition of security that is the best that can be achieved while maintaining reasonable levels of performance. We have shown that this definition can be realized using commercially available special-purpose hardware.

This definition and approach raises many questions. The first is real-world performance: What happens if we implement such a system? We plan to pursue such an implementation. This will lead to many challenges: More efficient join and indexing strategies that meet security requirements, concurrency control that does not violate security, query optimization approaches, etc. A second issue is when the security offered by Definition 4.4 is inadequate, and only (inefficient) approaches meeting Definitions 3.2-3.3 are adequate. While we have shown that our approach is the best we can (efficiently) do, the disclosures may be too much for some applications. Progress in this area will require better definitions of security; e.g., privacy definitions as rigorous as the security definitions that enabled progress in multilevel secure databases[16].

In spite of these questions, a database server managing encrypted data is feasible. Such a server will provide substantial benefit, such as easing enforcement of several of the ten principles proposed for a Hippocratic Database system[2]. With careful and rigorous work on ensuring that security is achieved, we can expect to see significant progress in this area. Key areas for future work are:

- Formal proof that Definition 4.3 adequately protects against inference from tracking query accesses,
- Formal proof that the Definitions given are complete and consistent,
- Methods for proving that query processing algorithms meet security definitions,
- Query optimization methods that provide provably secure means of combining the various component algorithms,
- Practical efficiency issues: query and data pipelining in conjunction with encryption (this will demand implementation on real hardware), and
- System issues: How does this play out in real-world applications?

We believe a new and important research direction is taking off; this paper shows how with careful definition of what it means to be secure, such research can ensure the promised security while achieving practical efficiency.

References

[1] N. R. Adam and J. C. Wortmann, "Security-control methods for statistical databases: A comparative study," *ACM Computing Surveys*, vol. 21, no. 4, pp. 515–556, Dec. 1989. [Online]. Available: http://doi.acm.org/10.1145/76894.76895

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proceedings of the 28th International Conference on Very Large Databases*, Hong Kong, Aug. 20-23 2002, pp. 143–154. [Online]. Available: http://www.vldb.org/conf/2002/S05P02.pdf

[3] ——, "Order-preserving encryption for numeric data," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Paris, France, June 13-18 2004.

[4] N. Ahituv, Y. Lapid, and S. Neumann, "Processing encrypted data," *Communications of the ACM*, vol. 20, no. 9, pp. 777–780, Sept. 1987. [Online]. Available: http://doi.acm.org/10.1145/30401.30404

[5] D. Asonov and J.-C. Freytag, "Almost optimal private information retrieval," in *Second International Workshop on Privacy Enhancing Technologies PET 2002.* San Francisco, CA, USA: Springer-Verlag, Apr. 14-15 2002, pp. 209–223. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&vo%lume=2482&spage=209

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '01)*, J. Kilian, Ed. Santa Barbara, California: Springer-Verlag, Aug. 19-23 2001, pp. 1–18. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&vo%lume=2139&spage=1

[7] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM*, vol. 45, no. 6, pp. 965–981, 1998. [Online]. Available: http://doi.acm.org/10.1145/293347.293350

[8] O. Goldreich, *The Foundations of Cryptography.* Cambridge University Press, 2004, vol. 2, ch. Encryption Schemes. [Online]. Available: http://www.wisdom.weizmann.ac.il/~oded/PSBookFrag/enc.ps

[9] ——, *The Foundations of Cryptography.* Cambridge University Press, 2004, vol. 2, ch. General Cryptographic Protocols. [Online]. Available: http://www.wisdom.weizmann.ac.il/~oded/PSBookFrag/prot.ps

[10] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, May 1996. [Online]. Available: http://doi.acm.org/10.1145/233551.233553

[11] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June 4-6 2002, pp. 216–227. [Online]. Available: http://doi.acm.org/10.1145/564691.564717

[12] "IBM PCI cryptographic coprocessor." [Online]. Available: http://www.ibm.com/security/cryptocards/html/pcicc.shtml

[13] "CCA API performance - IBM PCI cryptographic coprocessor." [Online]. Available: http://www.ibm.com/security/cryptocards/html/perfcca.shtml

[14] P. Lin and K. S. Candan, "Hiding traversal of tree structured data from untrusted data stores," in *Proceedings of Intelligence and Security Informatics: First NSF/NIJ Symposium ISI 2003*, Tucson, AZ, USA, June 2-3 2003, p. 385. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&vo%lume=2665&spage=385

[15] G. Ozsoyoglu, D. A. Singer, and S. S. Chung, "Anti-tamper databases: Querying encrypted databases," in *Proceedings of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, Estes Park, Colorado, Aug. 4-6 2003. [Online]. Available: http://art.cwru.edu/TOpapers/IFIP2003.Security.pdf

[16] B. Thuraisingham and W. Ford, "Security constraint processing in a multilevel secure distributed database management system," *IEEE Trans. Knowledge Data Eng.*, vol. 7, no. 2, Apr. 1995.

[17] "TCG TPM specification version 1.2," Nov. 5 2003. [Online]. Available: https://www.trustedcomputinggroup.org