Security Issues in Querying Encrypted Data*

Murat Kantarcıoğlu[1] and Chris Clifton[2]

[1] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083
http://www.murat.kantarcioglu.net
[2] Department of Computer Science, Purdue University, West Lafayette, IN 47907
{kanmurat, clifton}@cs.purdue.edu,
http://www.cs.purdue.edu/people/clifton

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 is considerable interest in the notion of a "secure database service": A Database Management System that could manage a database without knowing the contents[1]. While the business model is compelling, such a system must 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 security standards must be used carefully. For example, methods that quantize or "bin" values [1,2] reveal data distributions. Methods that hide distribution, but preserve relationships such as order [3,4], can also disclose information if used naïvely. While they may hide values in isolation, using such techniques on multiple attributes in a tuple can pose dangers.

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, ...), revealing *who* is being investigated is

* This material is based upon work supported by the National Science Foundation under Grant No. 0312357.

S. Jajodia and D. Wijesekera (Eds.): Data and Applications Security 2005, LNCS 3654, pp. 325–337, 2005.

sensitive: The appearance that they are accusing a customer of fraud could 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 customer/employee tuple whose ranks are similar, then the newspaper knows that with high probability, Chris is being investigated.

The key problem is that while encrypting a single value using order preserving encryption or a binning scheme may reveal no information, supporting multiple search keys for each tuple reveals a surprising amount. While methods meeting cryptographic security standards have emerged for some types of queries (e.g., keyword equality [5,6], additive update [7]) they do not meet the need for general database queries. To protect against 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. Section 2 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 3 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 – before enough queries have been seen to infer anything, the relationship between early and late queries will have been broken, preventing the adversary from inferring sensitive data.

In Section 3.2 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. Fortunately, there is an efficient and practical way

to achieve such a module: tamper-proof hardware [8]. We 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.

2 Implausibility of a Fully Secure Database Server

The cryptography community has developed solid and well-regarded definitions for securely encrypting a message. Encryption schemes are defined to be semantically secure if and only if the ciphertext reveals no information about the plaintext. We now use security definitions from cryptography to define what it means to "securely" encrypt and query a database.

2.1 Security Definitions for Encrypted Database Tables

Semantic security implies that, 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 [9].

Definition 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 over the internal coin tosses of* 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.

2.2 A Secure Method for Encryption of Database Tables

While one solution to the securely encrypting a database 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 (CTR) mode with a secure block cipher algorithm such as AES ([10], note that AES operates on 16 bytes blocks) to meet Definition 1 while still encrypting the individual fields in a tuple independently.

The idea in CTR mode is to choose a unique number (counter) for each block (a field of a tuple can be composed of multiple blocks but for simplicity

and without loss of generality, we assume that each field corresponds to a block) and encrypt that unique number. The resulting encryption of the counter is then xor-ed with the actual message block. The counter is stored in plaintext with the encrypted message. For example, let T_i be the i^{th} field of a tuple and C_i be the i^{th} encrypted field. We can encrypt the i^{th} field based on a counter value (ctr) that is incremented by n after encrypting a tuple with n fields. Encryption and decryption in CTR mode can be summarized as follows:

$$\text{CTR Encryption: } C_i = T_i \oplus E_K(ctr + i), i = 1..n$$
$$\text{CTR Decryption: } T_i = C_i \oplus E_K(ctr + i), i = 1..n$$

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.

2.3 Database Indistinguishability in the Presence of Queries

Database research has concentrated on efficient processing of queries. We would like to maintain this efficiency even if the data is encrypted. As examples in the introduction show, many prior proposals for querying encrypted data do not meet Definition 1 if an adversary is allowed to view data access patterns. 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 3.1 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.

Database Queries as a Special Case of Private Information Retrieval We still require that tuples be indistinguishable (Definition 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 database scan is required.

The definition comes from Private Information Retrieval (PIR) [11], which protects the query from disclosure. The server knows the data, but should learn nothing about the query. A PIR server must protect query privacy, and ensure the correctness of the query 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 the server will have the knowledge of those tuples' $a1$ fields. Note that we 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 privacy requirements.

Definition 2. ***(Correctness)***
Assume database D is stored securely on a server w.r.t Definition 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. Also privacy must hold even for a dishonest server:

Definition 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 previous 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. This could hold for even distinctly different queries (queries can inexpensively be padded to hide the differences), provided the same tables are access and the results are the same size. (The requirement that results be the same size is because padding results to the "maximum possible" result size would impose unacceptable inefficiences.)

We can define a secure query execution as one that runs on securely encrypted data and satisfies Definitions 3 and 2. We can show that even for queries that are running on a single table, above definitions imply that 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 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-size 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 $P = \mid Pr\{C_n(m_{Q_1^t}) = 1\} - Pr\{C_n(m_{Q_2^t}) = 1\} \mid$.

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

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 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. 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.

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

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 1. *A query execution that is secure in the sense of Definitions 3 and 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 (Lemma 2), if not then not all queries would satisfy the correctness Definition 2. We can now build a distinguisher for any query that requires less than full table access (Lemma 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. □

Database Queries as a Special Case of Software Protection. More generally, the cryptography community has produced the concept of *oblivious RAM*[12]: a method to obscure the program even to someone watching the memory access patterns during execution. 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.

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 $t > y$). 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 database 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.

3 Plausible Definition for a Secure Database Server

We have shown that any strict security and privacy requirement force us to scan entire databases. 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.

3.1 Definition

If the data and queries are encrypted, and the encryption satisfies multiple-message indistinguishability (e.g., Definition 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. ***(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.

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 5. ***(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 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 6. ***(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 the server has accessed t 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 7. *We consider a database to support secure query processing if it meets Definitions 1, 2, 5, and 6.*

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

3.2 Requirements for a Database Server

Methods that allow equality test of encrypted tuples, or field values in the tuples, violate Definition 7 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 6.

However, if we support a few simple operations that are "hidden" from the server, we can meet Definition 7. 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. To do this without sending tuples back to the querier we 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, and
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 [13], 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 7.

The idea is that the database is encrypted as in Section 2.2. 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[14]) – 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 6.) 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 needed 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. Finally, the remaining cached tuples are given to the server for delivery to the client.

Theorem 2. *Provided that tuples contributing to the result are (approximately) uniformly distributed across tuples read, this process meets Definition 7 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 5.

Definition 6 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 2 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[15]. 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 node 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 3. *The algorithm of [15] satisfies Definition 7.*

Proof sketch. Definition 5 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 6 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 6. □

This analysis is based on a single tuple result. 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.

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 2. 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 7. 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 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.

References

1. Hacigumus, H., Iyer, B.R., Li, C., Mehrotra, S.: 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 (2002) 216–227
2. Damiani, E., Vimercati, S.D.C., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational dbmss. In: Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA, ACM Press (2003) 93–102
3. Ozsoyoglu, G., Singer, D.A., Chung, S.S.: 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 (2003)
4. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order-preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France (2004)

5. Boneh, D., Boyen, X.: Efficient selective-id secure identity-based encryption without random oracles. In: EUROCRYPT. (2004) 223–238
6. Song, D., Wagner, D., Perrig, A.: Search on encrypted data. In: Procedings of IEEE SRSP, IEEE (2000)
7. Ahituv, N., Lapid, Y., Neumann, S.: Processing encrypted data. Communications of the ACM **20** (1987) 777–780
8. IBM: IBM PCI cryptographic coprocessor (2004) http://www.ibm.com/security/cryptocards/html/pcicc.shtml.
9. Goldreich, O.: Encryption Schemes. In: The Foundations of Cryptography. Volume 2. Cambridge University Press (2004)
10. NIST: Advanced encryption standard (aes). Technical Report NIST Special Publication FIPS-197, National Institute of Standards and Technology (2001) http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
11. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM **45** (1998) 965–981
12. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM **43** (1996) 431–473
13. Asonov, D., Freytag, J.C.: Almost optimal private information retrieval. In: Second International Workshop on Privacy Enhancing Technologies PET 2002, San Francisco, CA, USA, Springer-Verlag (2002) 209–223
14. Goldreich, O.: General Cryptographic Protocols. In: The Foundations of Cryptography. Volume 2. Cambridge University Press (2004)
15. Lin, P., Candan, K.S.: 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 (2003) 385