Fast, Secure Encryption for Indexing in a Column-Oriented DBMS

Tingjian Ge, Stan Zdonik
Brown University
{tige, sbz}@cs.brown.edu

Abstract

Networked information systems require strong security guarantees because of the new threats that they face. Various forms of encryption have been proposed to deal with this problem. In a database system, there are often two contradictory goals: security of the encryption and fast performance of queries. There have been a number of proposals of database encryption schemes to facilitate queries on encrypted columns. Order-preserving encryption techniques are well-suited for databases since they support a simple, and efficient way to build indices. However, as we will show, they are insecure under straightforward attack scenarios.

We propose a new light-weight database encryption scheme (called FCE) for column stores in data warehouses with trusted servers. The low decryption overhead of FCE makes comparisons of ciphertexts and hence indexing operations very fast. Since it is hard to use classical security definitions in cryptography to prove the security of any existing symmetric encryption scheme, we propose a relaxed measure of security, called INFO-CPA-DB. INFO-CPA-DB is based on a well-established security definition in cryptography and relaxes it using information theoretic concepts. Using INFO-CPA-DB, we give strong evidence that FCE is as secure as any underlying block cipher (yet more efficient than using the block cipher itself). Using the same security measure we also show the inherent insecurity of any order preserving encryption scheme under straightforward attack scenarios. We discuss indexing techniques based on FCE as well.

1. Introduction

Typically a DBMS provides two ways to achieve security: access control and data encryption. Of the two, access control is a relatively older way to protect sensitive data. However, access control by itself is not sufficient. An adversary who gains access to the database files can access sensitive data, thus, bypassing the access control mechanism. As a result, it is necessary to encrypt data in the DBMS.

Encryption is well studied in cryptography. However, when used in a DBMS, the traditional security definitions and properties of classical encryption schemes have a considerable performance impact on queries on encrypted data. First, standard definitions of security in cryptography [7,8,3] do not allow ciphertext values to reveal any information about the plaintext values, including the relative order information between their corresponding plaintexts. This implies that even comparisons have to go through decryption first. Second, encryption and decryption of existing cryptographic schemes have high CPU cost. Analogous to disk I/O, even though the speed of modern symmetric encryption schemes is improving, it remains costly for the database CPU. Therefore, data encryption significantly slows down query processing. For example, evaluating predicates that reference an encrypted column would generally require an expensive decryption step.

DBMS-specific encryption schemes that perform well for queries, but that preserve security are, thus, very desirable. One state-of-the-art technique is order-preserving encryption (e.g., OPES [1]). Such a scheme supports direct comparison of ciphertexts, which allows us to build indices in support of range queries. However, as we will show, such schemes are inherently not secure under straightforward attack scenarios.

In this paper, we propose an efficient light-weight database encryption scheme (called FCE), in which comparisons can be done with *partial* decryption (Early Stopping). FCE uses any block cipher to encrypt only a few bytes of random seeds in each page of the database, and uses lighter-weight computation to encrypt the actual data in a page. The low overhead of FCE enables efficient comparison and, therefore, efficient indexing on the ciphertext. We present evidence regarding the security of this scheme.

There have been a few proposals of "homegrown" encryption schemes for database systems for the purpose of fast search [1,2,9,15]. But how secure are these schemes? We stress that the importance of security cannot be under-estimated, because after all, "hiding" the information of data is the goal of using any encryption to begin with (otherwise the usage of it would not exist).

Let us look at a specific example. Suppose a mortgage company uses a customer table with schema *Customer* (*name*, *age*, *address*, *loan type*, *net assets*). Assume that only *net assets* is sensitive. So it is more efficient to encrypt only sensitive columns while leaving other columns in the clear. This way, fetching results of a non-sensitive column, say *age*, does not require decryption. For example, Oracle [14] provides column-level encryption. In this paper we assume this usage scenario. Consider using any order preserving encryption scheme (e.g., [1]) to encrypt *net assets* while leaving other columns in the clear. Then, an adversary who has access to the database files can discover the relative order of the *net assets* values between two customers identified by other attributes. This phenomenon of "insecurity" of newly proposed database encryption schemes is fairly common. The "bucketing" approach of [9] reveals value range correlation between columns. For example, an adversary could discover that the records that have values in the same bucket on one column are very likely to have values in the same bucket on another column. Column value distributions can also be revealed as pointed out by [1]. Similarly, column value distributions can be revealed by other schemes such as summation of random numbers [2], or by polynomial functions [15].

The security of a database encryption scheme must be examined more carefully. Unfortunately it is hard to prove the security of any symmetric encryption scheme used in databases today using the established security definitions in cryptography [7,8,3] (public key encryption schemes, which are provably secure, are slower and generally not used in databases). In light of this fact, we propose a relaxed measure of security based on the Real-Or-Random definition [3] in cryptography. The relaxation uses the concept of "entropy", and gives strong evidence that FCE is as secure as any block cipher that it uses for encrypting the random seeds (a few bytes) in each page header in the database. We also show the insecurity of any order preserving encryption scheme using this same measure. As any other new scheme, besides the theoretical analysis, the security of FCE needs to stand the scrutiny of cryptanalysts as well as the test of time. But those are beyond the scope of this paper.

FCE is specifically tailored to database systems in the following ways:

Comparison is fast, which facilitates the search of indices.

We show its security using INFO-CPA-DB, a security measure defined in the database context.

Secret random functions are stored at the *database page* level, which corresponds to the unit of I/O.

The rest of the paper is organized as follows. We first discuss the threat model in Section 2. Section 3 presents our new database encryption scheme: FCE. We also show a variant of it, r-FCE, which will be used for analysis. Section 4 discusses our security measure. In Section 5, we provide a detailed analysis of the security of any order-preserving scheme, and of FCE. Then, in Section 6, we discuss how indexing works with FCE. Section 7 gives our experimental performance results. We present related work in Section 8 and summarize the paper in Section 9.

2. Threat Model

We separate the issue of the security of the communication channel between client and DBMS server, and the issue of the security of the on-disk data [1], i.e., data security in the storage system of the server ("divide and conquer" approach). We assume the database software is trusted, in particular, the adversary does not have access to the values in the memory of the database software, and the database software is trusted to decrypt column values to evaluate predicates on encrypted columns (otherwise efficient processing by the server on encrypted data is not possible anyway). See Figure 1 for the system model.

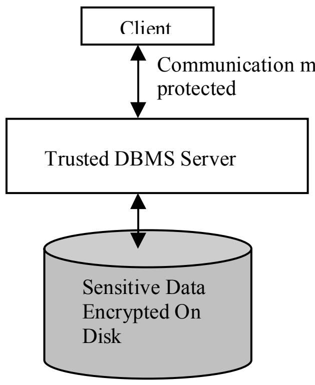

Client
Communication may be protected
Trusted DBMS Server
Sensitive Data Encrypted On Disk

Figure 1: Divide and conquer approach for DBMS security.

In this work, we specifically aim at encryption to ensure the security of on-disk data. We leave open the security of the communication between the client and server as a separate orthogonal issue, which can be protected by a traditional symmetric key encryption scheme if needed. Our threat model and protection goal are also considered by [1], which is not a surprise, as both consider efficient processing of encrypted data by the server.

3. The New Encryption Scheme

3.1 C-Store: A Column Oriented DBMS

We will analyze the usage of the newly proposed encryption scheme in the context of an open-source column-oriented DBMS called C-Store [19]. C-Store is a *read-optimized* relational DBMS. The most salient differences between C-Store and a traditional "row-store" system are that C-Store organizes data by column rather than by row, and that it makes heavy use of sorting and compression [18].

3.2 Fast Comparison Encryption (FCE)

3.2.1 r-FCE Algorithms

We first consider a version of FCE based on random permutations, hence the name r-FCE. C-Store stores the values of a column together in a set of pages. Suppose we somehow associate with each (encrypted) data page a truly random permutation whose size is the same as the page size (in bytes). Let the page size be P bytes (for C-Store, $P=64KB$). Thus each page (of the encrypted column) is associated with one of the $P!$ random permutations. To represent the random permutation, at least $log(P!)$ random bits are needed. When $P=64K$, using Stirling's formula [13], we can compute that each page needs $\log_2(64K!) \approx 954037$ random bits to represent the permutation. This is unrealistic. Hence, r-FCE is an idealized version. But we use r-FCE just as an intermediate scheme, purely for the purpose of information theoretic analysis. In section 3.2.2, we'll give an actual FCE scheme that uses k-wise independent functions, and we'll argue that it is *computationally indistinguishable* [7] from r-FCE.

r-FCE is a symmetric key encryption scheme for a DBMS. As in C-Store, we assume we are encrypting a whole page of data values of some column. Let's denote the key by *K*, and its bit-length by |K|. A typical value is 32Kb ("K" here following a number denotes a unit of "1024", not to be confused with the secret key "K". It should be clear from the context). The key generation algorithm is simply to generate a random |K| bit number. We next describe the encryption algorithm for a page of plaintext values.

Encryption Algorithm

Input: encryption key *K* (one key for the entire database), a page of plaintext values (*P* bytes), and a random permutation (function) associated with the page (one for each page) $perm: \{1,..,P\} \to \{1,..,P\}$.
Output: a page of ciphertext values.

We consider and encrypt *each byte* of the page separately:

For $i = 1$ *to* P,

(1) Let $d_i = perm(i) \bmod |K|$ which is clearly in the range of [0, |*K*|-1].

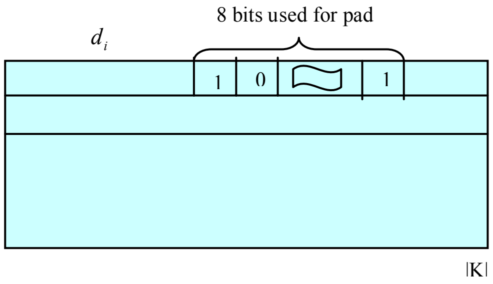


Figure 2: Using d_i to find one byte in K as a pad

(2) Then the ciphertext byte c_i of the plaintext byte b_i of the page is simply the bitwise XOR of byte b_i and the byte starting from the d_i'th bit of *K* (See Figure 2). If d_i falls on the last 7 bits of K, wrap around and use both the ending bits and the starting bits of *K* to form a pad byte. For example, if $d_i = |K| - 2$, then the pad byte is the last two bits of *K* concatenated with the first 6 bits of *K*.

Example: Let's say we are encrypting the 30th byte of a page, and the plaintext byte value is 00101110 in binary. As in C-Store, let |*K*|=32768, and P=65536. In step (1) of the encryption algorithm, the 30th byte gets a random permutation value $d_i = perm(30) \bmod 32768$, where *perm* is the random permutation associated with the page. Suppose $perm(30) = 33466$, so $d_i = 33466 \bmod 32768 = 698$. Next, in step (2) of the algorithm, we find the byte value starting from the 698th bit of the key *K*. Assume the byte (in *K*) is 10011010. Then the ciphertext byte is simply the XOR of this key byte and the plaintext byte: $10011010 \oplus 00101110 = 10110100$.

The decryption algorithm is the reverse of the encryption. The details are omitted, as it is fairly easy to derive.

3.2.2 The FCE Algorithms for C-Store

The only difference between FCE and r-FCE is that in FCE, we replace the random permutation $perm$ associated with each page by a "secret" k-wise independent function [12] (informally, it means any k points of the function are completely independent). Specifically, we can use a 4-wise independent function family (i.e., k=4; we'll explain the reason in Section 5) in step (1) of the encryption algorithm. We use a rather natural and efficient construction of a 4-wise independent function family, namely, random polynomials of degree k-1 (where k=4) as described in [12]:

$$p(x) = ax^3 + bx^2 + cx + d$$

where $a, b, c, d, x \in [0, P-1]$.

We now describe the implementation in C-Store where the page size is 64KB (hence the domain $[0, 2^{16}-1]$), and $|K| = 2^{15}$ (hence $\bmod 2^{15}$ in FCE algorithms). We need one such (secret) random polynomial per page, which means we need four (secret) random values a, b, c, d per page, totaling 64 bits. Therefore, we can store a random 64-bit seed at the page header (for a 64KB page, a 64-bit seed is certainly acceptable[1]), and use a block cipher (say DES) to get a 64-bit actual (secret) seed from the original seed, and split it to get a, b, c, d. We will use the same block cipher key (e.g., a 64-bit DES key) for every page of the database, and a different seed for each page of the database. The key of the block cipher and the encryption key *K* described above together form the secret key of FCE.

Observe that in step (1) of the encryption algorithm, the function $p(x)$ is applied on each byte position of the page, which means the set of input values ($[0, 2^{16}-1]$) are the same across pages. We therefore can pre-compute x^3 and x^2 values for each byte position, and use them universally for any page. Thus an evaluation of the random polynomial function simply involves 3 multiplications and additions and is very efficient. For example, this is fewer than 10 CPU cycles per encrypted byte on TMS320C6211 [4, 20], considering both computation and possible cache miss cost. The detailed analysis is in the full version of the paper, due to space constraints. In contrast, a DES implementation on the same processor needs 30 to 50 cycles per byte [16].

[1] Note that we can increase the number of random bits for more security.

The comparison of two ciphertext values starts from the most significant byte (assuming this can be known from the value type) and proceeds byte by byte from left to right. It is essentially an **Early Stopping** (ES) partial decryption of the two ciphertext values. The procedure stops as soon as a byte difference is found. This is feasible with FCE because encryption is done byte by byte, whereas in other block ciphers (e.g., DES) it is done in a unit of 8 bytes or more.

FCE uses a block cipher as a subroutine to encrypt only a small number of bytes per page. The encryption of the remainder of the data on the page is very light-weight. In FCE, comparing two ciphertext values and comparing a ciphertext with a plaintext value are very similar, as both work in the same manner starting from the most significant byte. As a result, joining two FCE encrypted columns and joining an encrypted with a non-encrypted column will work similarly. We'll discuss in Section 5.1 that this is not the case with OPES.

C-Store is read-optimized and targets data warehousing applications [18]. In such a system, an UPDATE is rare and is of less concern. Updates are applied in batch, rather than incrementally. During batch updates, a fresh random seed is generated for a page that uses FCE, to ensure security.

4. The Security Measure

It would be best to prove the security of our scheme according to an established definition. Unfortunately, the fact is that no symmetric encryption scheme is provably secure in that regard. We therefore propose a relaxation of an existing security definition in cryptography, the so-called *Real-Or-Random* definition [3]. The relaxation gives a formal security measure of an encryption scheme. We use entropy, which is a basic concept in information theory [5] that gives a universal measure of randomness. The *entropy* in bits of a discrete random variable X is given by

$$H(X) = -\sum_{x} \Pr(X = x) \log_2 \Pr(X = x)$$

where the summation is over all values x in the range of X. We assume the threat model defined in Section 2 and that encryption is specified per attribute. Observe that a column-wise storage is most friendly to such selective encryption. Nonetheless, this definition can be easily extended for cases that must encrypt every column of the table.

We first describe the intuition behind the security measure. An encryption scheme is secure if the adversary cannot distinguish the ciphertexts of any two (equal length) messages (i.e., plaintext values). In turn, the scheme is secure if the adversary cannot tell apart the ciphertext of any *Real* message and that of an equal-length *Random* message (By transitivity, one cannot tell apart the ciphertext of any two real messages). Namely, this is exactly what the *Real-Or-Random* definition requires. To capture the notion of "any" message, we simply let the adversary (to her advantage) *arbitrarily* choose any message. The more power we give to the adversary (and if we can still demonstrate certain security conditions are met), the more secure the system is. Such a notion in a security measure is termed "Chosen Plaintext Attack" (CPA) [7,3]. In our security measure (INFO-CPA-DB), we add a player (Guard of the cryptosystem) into the game. The Guard has to come up with *random* messages, which under a legally generated key, encrypt to the exact same ciphertext of the messages that the adversary has chosen. Consequently, not knowing which key is actually used, but just seeing the ciphertext, no one can tell whether it was from a real message or from random garbage. The security measure leaves space for certain "imperfections" of the cryptosystem by measuring the entropy of the (supposedly) random string that the Guard comes up with. The closer it is to the maximum entropy, the more random it is, and hence, the more secure the scheme is. In other words, we use "entropy" as a metric that measures how far the system is from being "perfectly secure". See Figure 3.

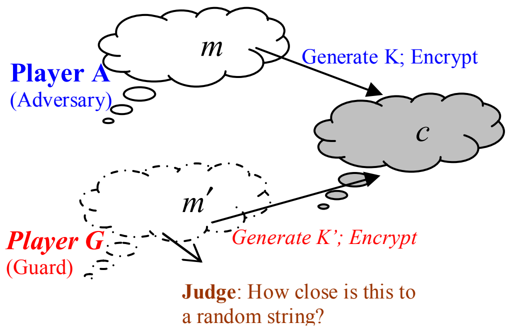


Figure 3: A mental game between A & G in INFO-CPA-DB definition.

Definition 1 [INFO-CPA-DB] Let $SE = (\kappa, \varepsilon, D)$ be a symmetric encryption scheme used for a DBMS (the three parameters are key generation, encryption, and decryption algorithms respectively). A relational table T that has sensitive information includes two columns: *ID*, which is the primary key, and *MSG*, which is encrypted. Consider a game between **player A** (Adversary) and **player G** (Guard of the cryptosystem).

(1) First, the key generation algorithm is run for **A** and **A** gets key K. To her own advantage, **A** arbitrarily generates q records for the table T (i.e., **A** chooses q *ID* values and *MSG* plaintext values $(id_i, m_i\ (1 \le i \le q))$). **A** encrypts the *MSG* values using her key K, and the table data is stored on disk. Let $\sum_{i=1}^{q} |m_i| = n$ and $c_i = enc(m_i, K)$. The database files (but not K) are passed to **G**.

(2) **G**, without knowing K, tries to come up with a "simulation script" that would create the exact same database files: **G** needs to run the key generation algorithm to get K', and come up with a sequence of "random" messages $m_i'(1 \le i \le q)$ such that $|m_i'| = |m_i|$

and $c_i = enc(m'_i, K')$ for all $1 \le i \le q$. In other words, using K' to encrypt the random messages, together with the same *ID* values used by **A** must produce the *exact same* files that **A** passes to **G**. Let $m' = m'_1 \circ m'_2 \circ ... \circ m'_q$ (where $\circ$ is bit-string concatenation).

(3) The success of **G** (and the security of *SE*) is measured by how close m' is to a uniformly random *n* bit string, specifically, how close $H(m')$ is to *n*. Then we say that *SE* is $(H(m'), n)$ secure. Clearly, the most secure scheme would be *(n, n)* secure.

Using entropy, INFO-CPA-DB relaxes the *Real-Or-Random* definition, and is a continuous measure of security. We note a caveat here that the exact relationship between the amount of entropy and the amount of resources necessary to break the scheme is unknown, and is left as future work.

5. Analysis of Order Preserving Encryption Schemes and of FCE

5.1 Analysis of Any Order Preserving Scheme

Under an order-preserving encryption scheme, ciphertext values preserve the order relationship of the corresponding plaintext values. It is ideal for query performance since comparisons can operate directly on ciphertext, saving the cost of expensive decryptions. A state-of-the-art order-preserving encryption scheme is OPES [1]. As expected, B-tree indices can be built and used on encrypted columns as well. However, as we will show, any order preserving encryption scheme is inherently not secure under the common usage scenario that only a subset of the columns is encrypted, which is our assumed usage model.

Intuitively, any order-preserving scheme reveals the order of column values between records. This information may be quite significant. We have seen an example in Section 1. Further, if the adversary somehow knows one or more plaintext values of the column, he or she can narrow down the possible range of other values. Thus, order-preserving schemes are prone to inference attacks. To be concrete, in our example if we know Alice and Charles bracket Betty, and we have side information about the assets of Alice and Charles ($1M and $1.1M), then we have a good estimate for Betty's assets. [1] uses "percentile exposure" as the security measure. However, that only tells if the scheme hides the column value distribution (which an adversary might already know to begin with). Security via encryption must hide much more than that.

Now we formally analyze the security of order-preserving encryption schemes using our information theoretic security measure. Recall that the idea of our definition is to give the adversary (player A) advantage and freedom to choose an arbitrary set of plaintext, and the player G (Guard of the cryptosystem) needs to respond with a set of "*random*" message that (under some key) encrypts to the same ciphertext. Intuitively, this definition rules out the security of any order-preserving scheme, because if the adversary chooses two values $m_1 < m_2$, then their ciphertext must satisfy $c_1 < c_2$. And whatever "random" messages (r_1, r_2) the Guard comes up with must satisfy $r_1 < r_2$, which makes $r_1 \circ r_2$ as a whole not random. Theorem 1 that follows is based on this observation and says that there is a set of messages (which the adversary may choose) that leaves the Guard nothing but *one choice* of plaintext that is the same as the adversary's, and hence is not at all random. So the entropy is 0, and the order preserving encryption is insecure.

Theorem 1: *Consider the INFO-CPA-DB definition of security for an order preserving encryption scheme. In the game, there exists a strategy for player A, such that whatever player G's strategy is, it holds that* $H(m') = 0$.

Proof: Let us give such a strategy for player A. We simply let plaintext be fixed-length bit strings, of length l bits. We can thus represent the plaintext domain as [0, $2^l - 1$], in that order, i.e., $0 < 1 < 2 < ... < 2^l - 1$. Note that A has the freedom to choose l such that 2^l is a reasonable number and A can fill in the table with this many records. A's strategy is to fill the table with 2^l records, where the MSG column values are distinct and in increasing order (as the ID column).

Now, because the encryption is order preserving, the ciphertext values must also be distinct and in increasing order. Player G, given the ciphertext, has to create a simulation script and come up with "random" MSG values $m'_i (1 \le i \le 2^l)$ with the same length (l bit), which can encrypt to the same set of ciphertexts with G's key K'. Again due to the order-preserving property, it must be that $m'_1 < m'_2 < m'_3 < ... < m'_{2^l}$. Clearly, due to the plaintext domain, it must be that $m'_i = i - 1 \, (1 \le i \le 2^l)$. In other words, there's only one possibility for $m'_i (1 \le i \le 2^l)$. Therefore, $H(m') = 0$.

Theorem 1 verifies our observation that any order preserving scheme is inherently insecure. Put another way, the INFO-CPA-DB definition protects us from attacks based on the order-preserving property of the encryption.

OPES assumes that the adversary does not have prior information about the value distribution. But in reality, many applications may have a column of sensitive data whose distribution is well known by the adversary, or whose distribution can be easily guessed (e.g., if there are only a limited number of probable distributions). Therefore, OPES cannot be used in these cases. On the other hand, it should be noted that once plaintext values are encrypted, unlike FCE, OPES can be used in an untrusted server environment, where decryption is not an option.

Further, OPES requires that the distribution is well known to the database (e.g., when a large amount of data already exists), before the key can be generated and encryption can happen. If data updates change the distribution, this process has to be repeated. For applications, a lot of times the column data's distribution is unpredictable before encryption is required, and may change over time. A complete recoding costs too much.

Also, consider the JOIN operation on two OPES encrypted columns of two tables. Most likely, the two columns do not have the same distribution, which means they are not directly comparable. Conversion from one side to the other must be carried out which involves expensive decryptions and/or encryptions.

Overall, the most severe problem with any order preserving encryption scheme is not its usage limitations, but the inherent security problem.

5.2 Security Analysis of r-FCE

We now show that r-FCE is indeed secure. In the game defined by INFO-CPA-DB, player A is the adversary, and we play the part of the player G (Guard). A picks a sequence of plaintext messages (and ID's) totaling ***n* bits**, then encrypts them using her key K under r-FCE. The resulting files on disk are handed to G. G now needs to create the simulation script. Our strategy for G is to simply call the key generation algorithm to generate a key K', decrypt the ciphertext values on disk using K' (note that the ciphertext values were encrypted using A's key K). During the decryption, a fresh random permutation for each page (and hence "d" values for each ciphertext byte) is obtained as in the encryption in r-FCE. Let the resulting plaintext values be m'. Now we try to obtain a lower bound of $H(m')$, the entropy of m'.

To compute the entropy of m', we first need to understand the random factors that determine the different outcomes of m'. In the game of the INFO-CPA-DB definition, A passes to G a sequence of ciphertext values, which we denote as c. G applies a randomized decryption algorithm to c. So, c is fixed. There are *two probabilistic factors* that determine the value of m':

The *random permutations* for each page, which derive the bit offsets into the key for decryption of each byte.

The *key K'*, which determines what bits are actually XOR'ed with c to get m'.

We first consider the effect of the *random permutations*. Suppose a random K' (the second random factor) is fixed. The process of decrypting each byte of c uses some random *d* value (determined by the permutation) to get 8 bits from K' and XOR's them with the byte of c. Two *d* values may result in the same sequence of 8 bits for the XOR. We put the 32K *d* values (1 to 32K) into $2^8 = 256$ groups, such that each group corresponds to a distinct sequence of 8 bits (let's call it a "pad byte" from now on). We consider random variables $X_i (0 \le i \le 255)$ that are the cardinalities of each group. We want to compute the number of unique assignments of pad bytes for a 64KB page. This is the same as the number of ways to write 256 numbers $0 \le i \le 255$, $2X_i$ times respectively, on a board that can hold 64K numbers. So the number of unique assignments in a page (resulting in unique m' values) is $\frac{(64K)!}{\prod_{i=0}^{255}[(2X_i)!]}$. There are $\frac{n}{8 \times 64K}$ pages, totaling $\left(\frac{(64K)!}{\prod_{i=0}^{255}[(2X_i)!]}\right)^{\frac{n}{8\times 64K}}$ unique assignments, each with *equal* probability (note that the *equal* probability property will greatly simplify the computation of entropy), resulting in *unique* m' values. Now we compute a lower bound of $H(m')$. Resorting to conditional entropy [5], we have:

$$H(m') > H(m' \mid X_0, X_1, \ldots, X_{255})$$

$$= \sum_{x_0, x_1, \ldots, x_{255}} \Pr(X_0 = x_0, X_1 = x_1, \ldots, X_{255} = x_{255}) \left(\frac{(64K)!}{\prod_{i=0}^{255}[(2x_i)!]}\right)^{\frac{n}{8\times 64K}} \cdot \frac{1}{\left(\frac{(64K)!}{\prod_{i=0}^{255}[(2x_i)!]}\right)^{\frac{n}{8\times 64K}}}$$

$$\cdot \left(-\log \frac{1}{\left(\frac{(64K)!}{\prod_{i=0}^{255}[(2x_i)!]}\right)^{\frac{n}{8\times 64K}}}\right)$$

$$= \sum_{x_0, x_1, \ldots, x_{255}} \Pr(X_0 = x_0, \ldots, X_{255} = x_{255}) \log\left(\frac{(64K)!}{\prod_{i=0}^{255}[(2x_i)!]}\right)^{\frac{n}{8\times 64K}}$$

$$= \frac{n}{8\times 64K} E\left(\log \frac{(64K)!}{\prod_{i=0}^{255}[(2X_i)!]}\right) \qquad (1)$$

We can approximate this bound by

$$H(m') > \frac{n}{8\times 64K} \log \frac{(64K)!}{\prod_{i=0}^{255}[(2\cdot E(X_i))!]} \qquad (2)$$

For now we use this approximation, and later we use Chernoff bounds [13] to show that with high probability the actual bound will be very close to this one, so this is indeed a good approximation. What we have done is to simplify the problem of computing $H(m')$ to giving a lower bound using the conditional entropy, conditioning on the second (harder) probabilistic factor.

All that remains is to compute $E(X_i) (0 \le i \le 255)$. Let X_i be the cardinality of the group with pad byte value *i*. Consider 32K random variables $Y_i (0 \le i \le 32K-1)$ satisfying

$$Y_i = \begin{cases} 1 & \text{if } d = i \text{ gives a pad byte value} = 0 \\ 0 & \text{if } d = i \text{ gives a pad byte value} \ne 0 \end{cases}.$$

Therefore,

$$\Pr(Y_i = 1) = \frac{1}{2^8} (0 \le i \le 32K-1)$$

due to the randomness of K'. Then,

$E(Y_i) = \frac{1}{2^8}\ (0 \le i \le 32K-1)\cdot$ We also have

$$X_0 = \sum_{i=0}^{32K-1} Y_i \qquad (3)$$

and from the linearity of expectation, we have

$$E(X_0) = \sum_{i=0}^{32K-1} E(Y_i) = \frac{32K}{2^8} = 128$$

Because key K' is uniformly random, then from symmetry, we have

$$E(X_i) = 128\ (0 \le i \le 255)\cdot$$

From all the above, we can compute the lower bound of $H(m')$:

$$H(m') > \frac{n}{8\times 64K}\log\frac{(64K)!}{\prod_{i=0}^{255}(2\cdot E(X_i))!} = \frac{n}{8\times 64K}\cdot\log\frac{(64K)!}{(256\,!)^{256}}$$

We can use Stirling's Formula [13] to compute $\log(64K)!$ and $\log 256!$. Finally we get the lower bound value: $H(m') > 0.9974n\cdot$

Recall that from (1) to (2) we used an approximation. We can use Chernoff bounds, union bound [13] (which basically says $\Pr(A\ or\ B) \le \Pr(A) + \Pr(B)$), and the constraint $\sum_{i=0}^{255} X_i = 32K$ to show that with high probability (1) indeed is very close to (2). We omit the details due to space constraints. They are in the full version of the paper.

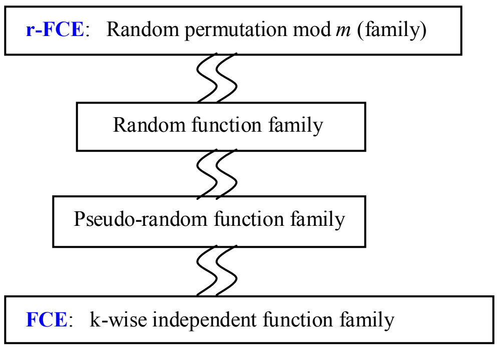
The lower bound result (0.9974n) indicates that $H(m')$ is very close to n, or in other words, m' is very close to a uniformly random bit string. This gives us great confidence in the security of r-FCE according to the INFO-CPA-DB and Real-Or-Random definitions. However, the missing entropy might be a concern for an application that requires strict security. We leave the problem of analyzing the effect of the leak as future work.

We can also obtain a general lower bound of entropy as a function of page size *P* and key size |*K*|, which indicates that a bigger page size or key size implies more security, but higher overhead. The details are in the full version.

From the security analysis of an order preserving encryption scheme and FCE, we can see that to show something is *secure*, we give a "simulation script" or strategy for *player G*. To *disprove* the security of some scheme, we give a strategy for *player A*.

5.3 Connection Between FCE and r-FCE

As we mentioned earlier, r-FCE uses ideal random permutations. So we have to use cryptographic techniques to realize it. We have introduced the FCE scheme in Section 3.2.2. Most cryptographic techniques are based on the "computational indistinguishability" [7] framework. Informally, it means that given "reasonable" resources (e.g., probabilistic polynomial time), one cannot distinguish between two distributions. We use "$A \approx B$" to denote that distribution A is "computationally indistinguishable" from distribution B.



r-FCE: Random permutation mod m (family)
Random function family
Pseudo-random function family
FCE: k-wise independent function family

Figure 4: From "ideal" to "realization", a road connected by "computational indistinguishability".

As we show in Figure 4 (omitting the proofs here), r-FCE uses a random permutation return value mod |*K*| (as the *d* values to probe into the key). This is computationally indistinguishable from a random function family whose function has the same domain as the permutation, but has the range of {0,…,|*K*|-1} (provided that the permutation size is a multiple of |*K*|). In turn, this random function family is computationally indistinguishable from a pseudorandom function family (with the same domain and range). In the final step of Figure 4, the FCE scheme uses a k-wise independent function family.

Hoory et al. in [10] discuss the motivation for understanding the relationship between k-wise independence and pseudo-randomness. They present an educated conjecture that 4-wise independence suffices to achieve cryptographic pseudo-randomness. FCE builds on this by using a 4-wise independent function family. We choose a rather natural and efficient construction of a 4-wise independent function family: random polynomials of degree k-1 (where k=4) as described in [12]:

$$p(x) = ax^3 + bx^2 + cx + d$$

where $a, b, c, d, x \in [0, P-1]$.

Clearly a higher "k" value in the k-wise independent function results in more security, but higher cost.

In summary, we end up with an FCE scheme that is *computationally* indistinguishable from r-FCE, which we have proved to be information theoretically secure. Therefore, we combine the concepts of information theoretic security and computational security.

We can analyze the security of some other encryption schemes using our security measure. It is not hard to show that the ideal (and impractical) schemes of One-Time-Pad and the CTR scheme using a random function in [3] are both *(n, n)* secure, and DES is *(56, n)* secure. We omit the analysis due to space constraints. While our analysis in Section 5.2 seems to suggest that r-FCE is more secure than a block cipher (e.g., DES), FCE, unlike r-FCE, uses a block cipher to encrypt 8 bytes per page to obtain the *a, b, c, d* values. Thus the security of FCE is bounded above by the security of the (subroutine) block cipher.

6. Indexing with FCE

In this section, we describe indexing on FCE encrypted data. We will be exclusively talking about widely used tree indexing (e.g., B+ trees), although FCE is also applicable to hash indexing, etc. Each page of the B+ tree will be laid out as usual on disk, except that each page will have a 64-bit seed at the top and parts of the page will be encrypted using FCE.

For an *internal node* of a B+ tree, we only encrypt the *key* values. We leave *pointers* (to other index nodes) in the clear. For a *leaf* node of a B+ tree, we encrypt both the *key* values and the *record ID*s. We need to encrypt record IDs, because otherwise from the leaf nodes, the order of the records could be inferred (which is exactly the problem with OPES). Efficient comparison between key values is the main challenge of indexing encrypted data. Therefore, we focus on how this works under FCE. Typically we are concerned with searching for a plaintext key value in the encrypted B+ tree, as this is what we need to do in processing a query. The comparison we do is between a plaintext and a ciphertext value, which, as we discussed in Section 3.2.2, is not much different from comparing two ciphertext values. As usual, the tree traversal starts from the root, and uses our special comparison method. Recall that we have the *Early Stopping* (ES) mechanism for comparisons. Observe that ES is more effective as the search is closer to the root of the B+ tree (upper levels), since it is more likely that the comparison is between two values with a big difference. As the search approaches the leaf level, key values approach the target value, and more byte comparisons are needed. Note that classical index key compression methods (on the plaintext key values), when applied, still work as usual, and in fact help ES, as redundant leading bytes are likely to be compressed, which further saves the CPU cost for decryption.

Unless the whole table is encrypted, a clustered index is in general not feasible with an encrypted column. This is because the order of the ciphertext values would be revealed by means of association with other columns otherwise (the same reason as needing to encrypt record IDs at leaf nodes). This is not specific to FCE and is universal for any encryption method. As a consequence, a sparse index is also not feasible in general.

With another classical encryption method, such as DES, B+ tree indexing is still possible in principle. The differences are:

It has bigger encryption blocks (e.g., 8 bytes for DES), hence a tree traversal may decrypt more than needed.

Its minimum unit of decryption is larger (e.g., 8 bytes for DES), so one has to perform more decryption all along the search path.

The decryption has more overhead than FCE.

As a result, it is less efficient to build an index with classical encryption methods. The performance comparison with indexing using DES is further conducted in the next section.

7. Experiments

We have mentioned that security has to be shown by analysis/proof and demonstrated it for FCE in Section 5. In this section, we study the following performance issues through experiments:

(1) How much overhead does FCE decryption/encryption have in the database context?
(2) How much does the Early Stopping mechanism help the index search?
(3) FCE has a small encryption block of one byte, compared to eight bytes of, say, DES. Combined with Early Stopping, what performance impact does this have on various kinds of queries (e.g., whether or not the index is *covering* for a query)?

7.1 Setup

We have implemented the FCE scheme, and B+ tree indexing on FCE encrypted columns, as well as DES encrypted columns. We extended the code to support DES and FCE encryption in C-Store on Debian Linux. We use the crypto library in OpenSSL 0.9.8b for DES. We use DES as the underlying block cipher of FCE, which is used to encrypt the 8-byte seed on each page. We have also implemented sort merge JOIN, which was not available in C-Store before. The algorithms were implemented in C++. The experiments were run on a Linux workstation with an AMD Athlon-64 2Ghz processor, 512 MB memory and a Samsung HD160JJ disk.

7.2 Overhead of FCE

Our first experiment compares the retrieval overhead of FCE with DES and unencrypted data. In this experiment, we select a single integer-valued column from a database. More precisely, we sequentially scan all the 64KB data pages, each containing 8K encrypted 8-byte <column value, recordID> pairs. All FCE runs include the cost of generating *a, b, c,* and *d* values for each page with DES. The file cache was warm in these experiments.

Figure 5 shows the retrieval cost per tuple. The FCE-1 line shows what happens when we only decrypt the column value, but not the record ID (since we are selecting a single column). With DES, in this case, we need to decrypt both the column values and the record IDs, as the DES block size (8 bytes) is larger than the column value size. (Some block ciphers require even bigger block sizes.) Therefore, in Figure 5 DES is decrypting twice as much ciphertext as

FCE-1. To compare DES and FCE when both decrypt the same amount of data, we included runs where FCE decrypts the record IDs as well. This is shown by the FCE-2 line in Figure 5, which is slightly faster than DES. For both FCE and DES, the encryption cost is about the same as decryption, so we don't include those measurements here.

For an arbitrary data type, the difference in performance between DES and FCE for sequential scans will fall somewhere between the relative performance of DES and FCE-1 or DES and FCE-2. The reason is that we need to decrypt a different amount of extra ciphertext depending on the size of a data value, especially if it is a variable size data type.

DES has been around for almost 30 years, and we believe that its OpenSSL implementation has been carefully tuned. Our implementation of FCE is not highly tuned, but is already outperforming DES. There is a good chance that with tuning, FCE can be even faster. This can also be seen from the cycle count comparison on the TMS320C6211 processor, in Section 3.2.2.

7.3 Indexing with FCE for Range Queries

In this section, we look at the performance of a simple SELECT query that has a predicate on the encrypted column: **SELECT COUNT(*) FROM t1 WHERE c1>? AND c1<?**, where c1 and c2 are an integer-valued columns. The query plan uses B+ indexes to find the record IDs that satisfy the range restrictions. The first step is to traverse the index to find the smallest value that satisfies the range restriction, then visit each subsequent value in the leaf nodes until it finds the smallest value that does not satisfy the range restriction. The count of the number of satisfying records is accumulated as the leaves are traversed. The data pages themselves are not visited, and the record IDs do not need to be decrypted.

Figure 6 shows the performance of this query under various data sizes, but fixed selectivity (25%). We can see that with DES encrypted columns, even though an index can be built (with more complex code changes), it is not as efficient as an FCE-based index, since an index search has to decrypt 8-byte DES blocks on the B+ tree search path. On the other hand, comparisons during an FCE index search are efficient for three reasons:

FCE has lower decryption overhead.

FCE does not need to encrypt and decrypt pointers in internal nodes, whereas DES may have to, due to the 8-byte block size.

Early Stopping (ES) happens during comparisons.

To evaluate how much savings ES contributes, we measure the cost with both ES disabled and enabled (*FCE-nES* and *FCE-ES1*, respectively, in Figure 6). Observe that the effectiveness of ES depends on the value distribution of the column. If the column has mostly small integer values (say, all less than 2^{16}, for *FCE-ES1*), then ES is less effective than when the column values are uniformly distributed in the range of $[0, 2^{31}]$ (*FCE-ES2*). Note that for some data types, such as character strings, it is less likely to have many common, redundant prefix bits between values and ES is more effective.

7.4 Variations of Queries

The next set of experiments investigates the performance impact of having to fully decrypt the indexed column at leaf nodes for a query. (In the previous COUNT query, we may not fully decrypt it due to Early Stopping.) There are at least two cases for a query:

The indexes are *covering* (i.e., only indexed columns appear in the query, thus there is no need to decrypt record IDs or read the base tables). Therefore only the keys need to be decrypted in the leaf nodes.

The indexes are non-covering, therefore both the keys and record IDs in an index need to be decrypted.

Example queries corresponding to these two cases are **SELECT c1 from t1 where c1>? AND c1<?**, and **SELECT c1, c2 from t1 where c1>? AND c1<?**, respectively, where integer-valued c2 is not encrypted and the index is on the encrypted integer-valued column c1. In Figure 7, the lines corresponding to these two queries are labeled *FCE-cover* and *FCE-nc*, respectively. We also compare them with DES encryption and with the COUNT query of Section 7.3, which is shown as *FCE-count* in Figure 7; the index is covering in this case. In all cases, we exclude the cost of retrieving data from the base tables, as that cost is independent of the encryption scheme.

Due to its 8-byte block unit of decryption, DES always decrypts the keys and record IDs in the leaf nodes, and hence the DES cost is the same for both queries. Observe that these small variations of queries cause a performance difference for FCE. Due to Early Stopping, *FCE-count*, which only does comparisons, is faster than *FCE-cover*, which decrypts all of the key bytes at the leaves. In turn, *FCE-cover* is faster than *FCE-nc* because FCE has a small encryption block size (1 byte) and decrypts only as needed (*FCE-cover* does not decrypt the record IDs in the leaves). In all of the cases, FCE is more efficient than DES.

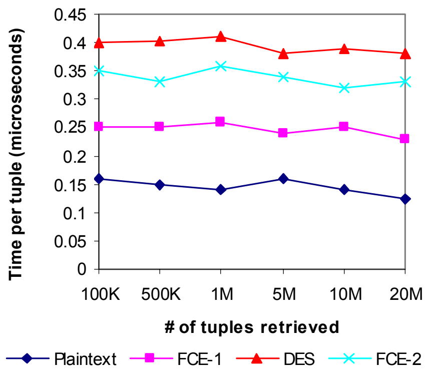


Figure 5: Tuple retrieval cost.

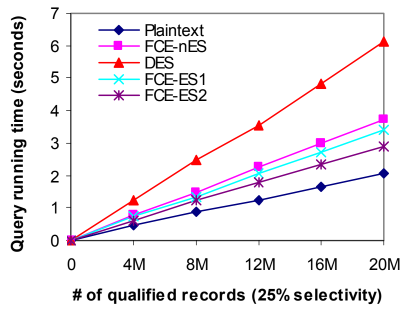


Figure 6: Performance of a range query utilizing an index built under different encryption schemes.

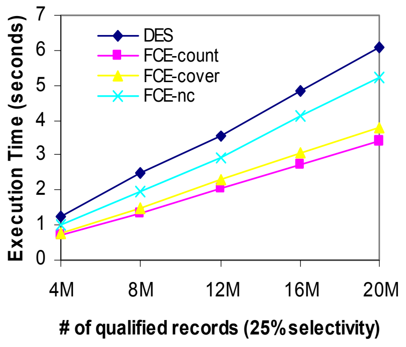


Figure 7: Performance of slightly different queries under DES and FCE.

8. Related Work

The work of [8] (Goldwasser and Micali) and [3] (Bellare et al.) studied the formal notions of security for encryption. Our information theoretic measure is based on the Real-Or-Random (ROR) definition against chosen-plaintext attack (CPA) in [3].

[1, 9, 15, 17] are similar to our work in that they typically propose a new scheme of encryption in such a way that efficient query processing on encrypted data is possible. Although there are similarities with our work in [17], their goal is that an untrusted server cannot learn anything about the plaintext, but still can perform search, which is *only* equality search. We have a different threat model, and our goal is to support fast queries in a DBMS, in particular, to use indices on ciphertext. The idea in [9] is to map encrypted values into buckets for early filtering without decrypting the value. The result of the rewritten query contains false hits that must be removed in a post-processing step. We have discussed its security problem in Section 1, and the performance and security problems are also discussed in [1, 11]. [1] proposes an order preserving encryption scheme. Although ideal for comparison, it has inherent security problems that we have discussed at length.

9. Conclusions

Encrypting sensitive data in a DBMS becomes more and more crucial for protecting it from being misused by intruders who bypass conventional access control mechanisms and have direct access to the database files. One must study the security of a new scheme in a systematic way. In this paper, we proposed the FCE database encryption scheme and demonstrated its security and efficiency for databases. We discussed indexing issues with FCE and experimentally evaluated the overhead and performance.

10. Acknowledgments & References

This work was supported by the NSF, under the grants IIS-0086057 and IIS-0325838, and a gift from Vertica Systems, Inc.

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. *ACM SIGMOD 2004* June 13-18, 2004, Paris, France.

[2] G. Bebek. Anti-tamper database research: Inference control techniques. *Technical Report EECS 433 Final Report*, Case Western Reserve University, November 2002.

[3] M. Bellare, A. Desai, E. Jokipii, P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Symposium on Foundations of Computer Science*, IEEE, 1997.

[4] H. Choi. TMS320C6211 Architecture Overview. http://cnx.org/content/m10872/latest/.

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. A Wiley-Interscience Publication, 1991.

[6] DES. Data Encryption Standard. *FIPS PUB 46, Federal Information Processing Standards Publication,* 1977.

[7] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2003.

[8] S. Goldwasser and S. Micali. Probabilistic Encryption. In *J. of Computer and System Sciences*, Vol. 28, April 1984, pp. 270-299.

[9] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Madison,Wisconsin, June 2002.

[10] S. Hoory, A. Magen, S. Myers and C. Rackoff. Simple permutations mix well. *The 31st International Colloquium on Automata, Languages and Programming (ICALP),* Lecture Notes in Computer Science 3142, Springer, 2004, pp. 770–781.

[11] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. *The 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*. August 7-10, 2005, Storrs, Connecticut.

[12] E. Kaplan, M. Naor, and O. Reingold. Derandomized constructions of k-wise (almost) independent permutations. In *APPROX-RANDOM*, pages 354–365, 2005.

[13] M. Mitzenmacher, E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[14] Oracle Corporation. *Database Encryption in Oracle 8i*, August 2000.

[15] G. Ozsoyoglu, D. Singer, and S. Chung. Anti-tamper databases: Querying encrypted databases. In *Proc. of the* 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security, Estes Park, Colorado, August 2003.

[16] R.S. Preissig. Data Encryption Standard (DES) Implementation on the TMS320C6000. In *Texas Instruments Application Report, SPRA702*, November, 2000.

[17] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symp. on Security and Privacy*, Oakland, California, 2000.

[18] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik. *C-Store: A Column Oriented DBMS*. In VLDB 2005, Norway.

[19] http://db.csail.mit.edu/projects/cstore/.

[20] TMS320C6211 Cache Analysis. In *Texas Instruments Application Report, SPRA472*, September, 1998.