Design and Analysis of Querying Encrypted Data in Relational Databases

Mustafa Canim and Murat Kantarcioglu

Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083
{mxc054000, muratk}@utdallas.edu

Abstract. Security and privacy concerns as well as legal considerations force many companies to encrypt the sensitive data in databases. However, storing the data in an encrypted format entails non-negligible performance penalties while processing queries. In this paper, we address several design issues related to querying encrypted data in relational databases. Based on our experiments, we propose new and efficient techniques to reduce the cost of cryptographic operations while processing different types of queries. Our techniques enable us not only to overlap the cryptographic operations with the IO latencies but also to reduce the number of block cipher operations with the help of selective decryption capabilities.

1 Introduction

Sensitive data ranging from medical records to credit card information are increasingly being stored in databases and data warehouses. At the same time, there are increasing concerns related to security and the privacy of such stored data. For example, according to a recent New York Times article [1], records belonging to more than hundred million individuals have been leaked from databases in the last couple of years. Although currently criminals have not taken advantage of such disclosures effectively, there is an obvious need for better protection techniques for sensitive data in databases. Common techniques such as access controls or fire-walls do not provide enough security against hackers that use zero-day exploits or protection from insider attacks. Once a hacker gets an administrator access to a server that stores the critical data, he/she can easily bypass the database access control system and reach the entire database files. Although it brings some extra cost, encrypting the sensitive data is considered as an effective last line of defense, to counter against such attacks [2]. Also recently, legal considerations [3] and new legislations such as California's Database Security Breach Notification Act [4] require companies to encrypt sensitive data. To assist its customers with legislation compliance hurdles, Microsoft recently developed a new SQL server that comes with built in encryption support [5]. IBM also offers a similar functionality in its DB2 server, in which data is encrypted (and decrypted) using a row-level function [6]. Clearly, if the encryption

keys are not compromised, a hacker (or a malicious employee) that controls the system will not be able to read all the sensitive data stored on the hard disk.

Cryptographic operations, required to store sensitive data securely, entail significant amount of cumbersome arithmetic calculations. Coupled with bad design choices, expensive cryptographic operations needed for querying and processing encrypted data could decrease the system performance dramatically. On the other hand, good design choices may drastically affect the overall performance. With this in mind, in this paper, we discuss some of the design issues to encrypt and query the sensitive data that resides in database disks.

To reduce the performance loss that arises from using cryptographic techniques, we analyzed different block cipher modes of operations and compared their performances under various disk access patterns to see which modes are suitable for databases and allow us to parallelize the IO latencies with the cryptographic operations. Based on our experiments and analyses, we suggest using an efficient and provably secure encryption mode of operation that also enables selective decryption of large data blocks. We also propose a new approach for storing and processing encrypted data and we show that by using suitable encryption mode, the additional time needed for processing different types of queries can be significantly reduced. Before describing our approach, we briefly discuss the threat model assumed in this paper.

1.1 Threat Model

In this paper, we assume a threat model similar to the one considered in [7], where the database server is trusted and only the disk is vulnerable to compromise.

We consider an adversary that can only see the files stored at the disk but not the access patterns. In this case, we just need to satisfy the security under chosen plain text attacks. In other words, we guarantee that (assuming used blockcipher (e.g. AES) is secure) by looking at the contents of the disk, any polynomial-time adversary will have negligible probability of learning anything about the sensitive data.

Specifically, we assume that (See Figure 1):

- *The storage system used by the database system is vulnerable to compromise.* Only the files stored in the storage system are accessible to the attacker.
- *Query Engine and Authentication Server is trusted.* We assume that queries are executed on a trusted query engine.
- *All the privacy-sensitive data will be stored encrypted.* From the previous work related to inference controls, we know that probability distribution information may create unintended inference channels. Since we do not know such inference channels in advance, we want our solution not to create any inference problem. Therefore, in addition to sensitive data, all the information that can reveal sensitive data (e.g. log files) will be stored encrypted.

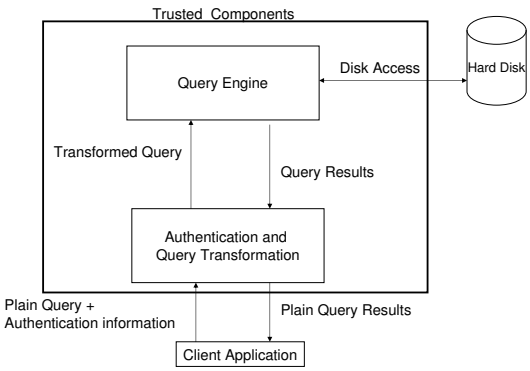

Fig. 1. Information flow and trust model for querying the encrypted data

1.2 Related Work

In [8], Bayer and Metzger explored the idea of using block ciphers and stream ciphers for encrypting B+ trees and random access files. They suggest generating different keys for each page based on the page id to break potential correlation attacks (pages encrypted with the same key and the same data will have the same cipher text). The disadvantage of this approach is, changing keys for each page imposes big key initialization costs. Hardjono and Seberry [9] suggested special combinatoric structures to disguise keys in B+ trees. Unfortunately, their combinatoric approach has not received the level of scrutiny required to achieve trust (as have others such as AES).

Querying encrypted data where even the database server is not trusted was first suggested in [10]. Hacigumus et al. [10] suggested partitioning the client's attribute domains into a set of intervals. The correspondence between intervals and the original values kept at the client site and encrypted tables with interval information are stored in the database. Efficient query of the data is made possible by mapping original range and equality query values to corresponding interval values. In subsequent work, Hore et al. [11] analyzed how to create the optimum intervals for minimum privacy loss and maximum efficiency. In [12], the potential attacks for interval based approaches were explored and models were developed to analyze the trade off between the efficiency and the disclosure risk. This line of work is different from our current problem because we assume that the database server is trusted and only the disk is untrusted.

In [7], Agrawal et al. suggested a method for order preserving encryption (OPES) for efficient range query processing on encrypted data. Unfortunately, OPES only works for numeric data and is not trivial to extend to discrete data.

In [13], Iyer et al. suggested data structures to store and process sensitive and non-sensitive data efficiently. The basic idea was to group encrypted attributes

on one mini-page (or one part of the tuple) so that all encrypted attributes of a given table can be decrypted together. In [14], Elovici et al. also suggested a different way for tuple level encryption.

Our consideration is different from the ones in [13, 14] in many aspects. Unlike the previous work, we provide:

- Complete analysis of block cipher modes suitable for databases,
- Analysis of the experiments about overlapping the IO latencies with the cryptographic operations by using multi-threading,
- A new approach for storing encrypted data in database pages by utilizing selective decryption property of CTR mode.

1.3 Organization of the Paper

In Section 2, we analyze the performance of different encryption modes under different granularity and disk access patterns. In Section 3, we discuss tuple level, page level, and mini page level encryption approaches and introduce a new approach for keeping records encrypted. Finally, we conclude the paper with the discussion of other related issues in querying encrypted data.

2 Block Cipher Modes Suitable for Databases

To store privacy-sensitive data securely, we need to use encryption methods secure against chosen plaintext attacks. Also we need general solutions that could support all kinds of data that needs to be encrypted. For example, the Order Preserving Encryption (OPES) idea suggested in [7] works for only numeric data.

Securely encrypting any kind of data is well studied in the cryptography domain. Any kind of long data is encrypted using operation modes based on secure block ciphers (e.g, AES[15]). All of these encryption modes process the long data by dividing into fixed size blocks(e.g 16 bytes in AES[15]) that can be processed by the block cipher. The obvious question is which block cipher mode is preferable for encrypting sensitive data in databases. To choose the best mode possible, we need to analyze the effect of these modes under specific database operation conditions. Specifically we look at:

- **Performance of Encryption Modes under Different Granularity:** We need to encrypt and decrypt the data at different granularity. If we are encrypting at the tuple level, we may need to decrypt small blocks at a time, if we use page level encryption, we need to decrypt potentially a page long encrypted data. Ideally, we should have a mode where different granularity has little effect on the total performance.
- **Performance of Encryption Modes under Different Disk Access Patterns:** We need to have an encryption method that enables efficient decryption under different disk access patterns.

First, in section 2.1, we give an overview of the block cipher modes that are suggested by the National Institute of Standards and Technology (NIST). Later on, in section 2.2 and 2.3, we analyze the performance of different block cipher modes under different encryption granularity and disk access patterns. Finally, we conclude this section with the discussion of which block cipher mode to choose for database encryption.

2.1 Overview of Block Cipher Modes

Before we briefly give an overview of different block cipher modes, we discuss the notations we use through out the paper. Let $E_K()$ be the encryption operation with the key K and $D_K()$ be the decryption operation with the key K. Let P_i be the i^{th} plain text block, C_i be the i^{th} ciphertext block, IV be the initial random vector and b be the block cipher input size (e.g 128 bits for AES[15]). Let $LSB_s(x)$ be the s least significant bits of x and similarly let $MSB_s(x)$ be the s most significant bits of x. Also below, $S[i..j]$ denotes the i^{th} through j^{th} bit of string S , $||$ denotes the string concatenation, and $\oplus$ denotes the bitwise xor operation. Table 1 summarizes the notations used for describing block cipher modes.

Table 1. Notations used for describing block cipher modes

$E_K()$	Block cipher encryption with the key K
$D_K()$	Block cipher decryption with the key K
b	Block cipher block length in bits
C_i	i^{th} ciphertext block
P_i	i^{th} plain text block
$LSB_s(x)$	s least significant bits of x
$MSB_s(x)$	s most significant bits of x
$S[i..j]$	i^{th} through j^{th} bit of string S
$\|\|$	String concatenation
$\oplus$	binary xor operation

Also, for the sake of simplicity, we assume that the plaintext P is n blocks long (i.e. $n \cdot b$ bit long plaintext). Under these assumption, the block cipher modes of operations suggested by NIST can be summarized as follows:(The details can be found in FIPS-SP 800-38A [16])

- **Electronic Code Book(ECB):** In ECB mode, each block of the plaintext is encrypted independently. Similarly each block of the ciphertext decrypted independently. Unfortunately, this mode is not secure since it reveals distribution information. (Note that if $P_i = P_j$ then $C_i = C_j$)

  ECB Encryption: $C_i = E_K(P_i),\ i = 1..n$ **ECB Decryption:** $P_i = D_K(C_i),\ i = 1..n$

– **Cipher Block Chaining(CBC):** Each block of the ciphertext is created by encrypting the result of the previous ciphertext block xored with the current plaintext block.

CBC Encryption:	**CBC Decryption**
$C_1 = E_K(P_1 \oplus IV)$	$P_1 = D_K(C_1) \oplus IV$
$C_i = E_K(P_i \oplus C_{i-1}),\ i = 2..n$	$P_i = D_K(C_i) \oplus C_{i-1},\ i = 2..n$

– **The Cipher Feedback Mode:(CFB)** In this mode, successive segments (let s be the size of these segments where $1 \leq s \leq b$ and C_i^s, P_i^s denote the s-bit sized segments of ciphertext and plaintext respectively.) of ciphertext are concatenated to create an input for forward cipher to create blocks that are xored with plaintext segments.

CFB Encryption:	**CFB Decryption:**
$I_1 = IV$	$I_1 = IV$
$I_i = LSB_{b-s}(I_{i-1}) \vert C_{i-1}^s, i = 2..n$	$I_i = LSB_{b-s}(I_{i-1}) \vert C_{i-1}^s, i = 2..n$
$O_i = E_K(I_i), \quad i = 1..n$	$O_i = E_K(I_i), \quad i = 1..n$
$C_i^s = P_i^s \oplus MSB_s(O_i), \quad i = 1..n$	$P_i^s = C_i^s \oplus MSB_s(O_i), \quad i = 1..n$

– **Output Feedback Mode(OFB):** This mode is very similar to CFB with s taken as b. In this simplified description of OFB, we assume that ciphertext is $n * b$ bits long.

OFB Encryption:	**OFB Decryption:**
$I_1 = IV$	$I_1 = IV$
$I_i = O_{i-1}, \quad i = 2..n$	$I_i = O_{i-1}, \quad i = 2..n$
$O_i = E_K(I_i), \quad i = 1..n$	$O_i = E_K(I_i), \quad i = 1..n$
$C_i = P_i \oplus O_i, \quad i = 1..n$	$P_i = C_i \oplus O_i, \quad i = 1..n$

– **Counter Mode(CTR):** In this mode a counter(ctr) is incremented and the encrypted counter value is xored with plaintext block to get the ciphertext block.

CTR Encryption:	**CTR Decryption:**
$C_i = P_i \oplus E_K(ctr + i), i = 0..n-1$	$P_i = C_i \oplus E_K(ctr + i), i = 0..n-1$

2.2 Evaluating the Performance of Encryption Modes under Different Encryption Granularity

In [13], Iyer et al. states that encrypting the same amount of data using few encryption operations with large data blocks is more efficient than many operations with small data blocks. In order to support this claim, they conduct an experiment using three ciphers: AES[15], DES[17] and Blowfish[18] under different data blocks: 100 bytes, 120 bytes, 16 KBytes. Based on the results of this experiment, they conclude that encrypting data few small units at a time takes longer than encrypting the same amount of data using larger data blocks.

As they stated in their paper [13], the key initialization costs constitute the most important cause of the high amount of time spent in encrypting small blocks

of data. We expect to observe less time difference when the data is encrypted under different granularity if we can use **one key** per database table. Using one key per database table is feasible for many practical applications, because we can encrypt 2^{128} bits of data securely [19] using CTR mode of encryption with a given key. To test this view, we conducted some experiments to investigate the effect of key initialization. These experiments are performed under different encryption modes to see which mode is more appropriate to use under different encryption granularity.

In our experiments we used the OpenSSL [20] crypto library's CBC, CTR, OFB and CFB implementation of AES[15]. We chose AES because it is the current standard and supported by various companies. ECB is not used since it is not regarded as a secure mode of operation. We also implemented a slightly modified version of CTR (referred as CTR4). Modified CTR implementation encrypts all the counter values first (i.e. calculates all the $E_K(ctr + i)$ first) and then xors it with the plain text data. Algorithms 1 and 2 describe the details of the modified CTR implementation. In Algorithm 1, we first create a string S using the encrypted counter values and then xor the first l bit of S with plaintext message P to get the ciphertext C. Since encrypted counter values are xored with the plaintext, we do not need to do any padding if the size of the plaintext is not the multiple of the block-cipher length b.

Algorithm 1 Modified CTR Encryption (CTR4)

Require: Plain text P with length l, initial counter value ctr, and block-cipher length b.
Set $S = E_K(ctr + 0)||E_K(ctr + 1)|| \ldots ||E_K(ctr + \lceil \frac{l}{b} \rceil - 1)$
Set $C = P \oplus S[0..l - 1]$
return (ctr, C)

Similarly, in Algorithm 2, we first create a string S using the encrypted counter values and then xor the first l bit of S with ciphertext C to get the plaintext P

Algorithm 2 Modified CTR Decryption (CTR4)

Require: Ciphertext C with length l, initial counter value ctr, and block-cipher length b.
Set $S = E_K(ctr + 0)||E_K(ctr + 1)|| \ldots ||E_K(ctr + \lceil \frac{l}{b} \rceil - 1)$
Return P where $P = C \oplus S[0..l - 1]$

Since all of the block cipher operations are done independent of ciphertext data, CTR4 decryption can be executed in a multi-threaded fashion. One thread could be used for creating the encrypted counters (i.e., for computing S in Algorithm 2), and other thread could be used to read the encrypted data from

the disk (i.e. reading C from the disk in Algorithm 2). The original implementation of CTR mode in OpenSSL crypto library calculates C_{i-1} before calculating $E_K(ctr + i)$. Thus it is not suitable for multi-threading.

Experiments: All of our experiments are conducted on a 2.79 GHz Intel Pentium D machine with 2 GB memory on Windows XP platform. We ran each experiment five times and reported the average results.

In the first experiment, we encrypted a total of one GB data which is cached in memory 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192 bytes at a time. In this experiment, the same key is used for encrypting the entire one GB data, and the key initialization is done only once. In other words, AES_set_encrypt_key() function is called once. [1] In Table 2, we report the results of experiment 1 in seconds.

Table 2. Encryption of 1 GB data under different block sizes

	Experiment 1 (**With** key initialization)					Experiment 2 (**Without** key initialization)				
Block size (Byte)	CBC	CTR	OFB	CFB	CTR4	CBC	CTR	OFB	CFB	CTR4
16	24.98	26.19	27.09	25.64	23.02	49.27	50.77	50.50	50.84	47.91
64	24.31	25.75	26.52	25.03	22.81	30.44	32.13	31.41	31.53	29.16
128	23.78	25.56	26.27	25.20	22.31	26.92	28.83	28.27	28.58	25.44
256	23.56	25.50	26.39	25.44	22.11	25.17	27.42	26.61	27.09	23.95
512	23.42	25.50	26.48	25.41	22.17	24.34	26.52	25.89	26.22	23.23
1024	23.38	25.52	26.47	25.39	22.03	24.00	26.23	25.45	25.75	22.78
2048	23.33	25.48	26.55	25.42	22.05	23.78	25.98	25.23	25.66	22.63
4096	23.36	25.53	26.34	25.38	22.28	23.72	25.84	25.25	25.34	22.78
8192	23.25	25.56	26.39	25.41	22.30	23.53	25.84	25.14	25.42	22.75

In the second experiment, we again encrypted one GB of data similar to experiment 1 but in this experiment, key initialization is done at the beginning of each data block. Average results of experiment 2 are shown in Table 2 in seconds.

Experiment 1 indicates that CTR4 mode is the fastest, if the key initialization done just once at the beginning of the encryption process. Also, if we compare the results of experiment with or without key initialization for CTR4 (shown in figure 2), we can observe that encrypting larger blocks requires less amount of time if the key initialization is performed every time before encrypting each data block. However, the time required to encrypt the data does not depend on block size if key initialization is done just once. In other words, if the blocks used are not too small (i.e. larger or equal to 64 bytes) and one key is used per database table,

[1] Initialization means generating substitution tables based on the secret key and this can be a costly operation as reported in [13].

the encryption block size does not effect the total decryption times. Actually, it is easy to see this fact from the API of modern crypto packages like the OpenSSL Library. For example in OpenSSL, to start encrypting with a key (similar for decryption), the key initialization function (AES_set_encrypt_key()) should be called first. After the initialization is done, encryption function can be called on various size blocks.

At the same time, experiment 1 indicates that encrypting small blocks (i.e. less than 64 bytes) at a time could be a problem. Fortunately, clever usage of CTR4 mode can solve the small block size problem. (This is not possible with other modes) Since each block is encrypted independently, we can combine blocks from different tuples (i.e., combine 4 blocks from 4 different tuples to create a 64 byte block) and decrypt/encrypt them together.

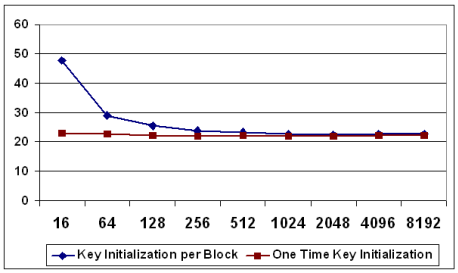

Fig. 2. Effect of key initialization under different granularity

2.3 Performance of Encryption Modes under Different Disk Access Patterns

In this part we conducted four experiments to evaluate the performance of encryption modes under random and sequential disk access patterns. In each experiment, data is accessed 4 KB page at a time, since it is the default page size in many DBMSs such as IBM DB2 [21]. We describe those four experiments below.

- **Experiment 3:**
  In the first part, one GB of encrypted file cached in memory is read **sequentially** and decrypted using different encryption modes. In the second part, the same experiment is repeated with the same file which is accessed from disk (i.e. file is not cached in memory). The results of the experiment are shown in Table 3.
  The results show that whether the data resides in memory or not, does not affect the time for reading and decrypting data sequentially. This implies that during the decryption of the current page disk controller can prefetch the next page.

Table 3. Results of experiment 3 (reading and decrypting 1 GB file sequentially)

Crypto Mode:	CBC	CTR	OFB	CFB	CTR4
from memory	29.02	31.17	30.92	30.89	27.89
from disk	28.91	31.94	30.5	31.05	27.31

– **Experiment 4:**
In the first part, 512 MB of the 1 GB encrypted file is accessed **randomly** 4K page at a time from memory and decrypted using different encryption modes. In the second part, the same experiment is repeated with the same file which is accessed from the disk. The results of the experiment are shown in Table 4

Table 4. Results of experiment 4 (reading one GB file randomly and decrypting 512 MB)

Crypto Mode:	CBC	CTR	OFB	CFB	CTR4
from memory	15.7	16.73	16.38	16.52	15.03
from disk	262.63	262	271.94	266.3	267.64

The results of the experiment 4 indicate that random access to the pages causes significant delays if the data is not cached in the memory.

– **Experiment 5:**
Multi-threaded version of CTR4 [2] is used to decrypt the 512 MB **randomly** accessed data, which took 257.3 seconds in the average. This experiment is not performed with the other modes since they do not allow multi threading during decryption.
If the result of experiment 5 is compared with experiment 4, we can see that multi threaded version of CTR4 reduces the decryption cost significantly, by overlapping the IO operations with decryption operations. In experiment 4, decrypting 512 MB of 1 GB file randomly takes 266.1 seconds in the average. However, performing the same experiment takes 257.3 seconds in this experiment. Therefore, multi threaded version of CTR4 runs almost 9 seconds faster than other modes of operations in the average.
In order to support this claim, we have calculated the time to decrypt 512 MB allocated memory space sequentially with CTR4, which took 11.2 seconds to perform. This result indicates that 9 seconds of running experiment 5 is overlapped with IO operations.
This implies that using hardware accelerators, most of the time required for cryptographic operations can be overlapped with the random access IO operations.

[2] i.e One thread reads the data, other thread encrypts the counter values. When both threads are done, their results are xored to get the plain text.

– **Experiment 6:**
Multi-threaded CTR4 is used to decrypt the 1 GB file **sequentially**. The average of the results indicates that it takes 28.05 seconds to decrypt the file sequentially. When compared with the results of experiment 3, it can be concluded that the time to read and decrypt one GB file sequentially is not significantly reduced by using the multi threaded version of CTR.

The above results imply that the cost of the cryptographic operations can be overlapped with the cost of the IO operations using fast hardware based multi-threaded implementation of CTR mode, which could improve the performance especially in storage area networks where disk access latency could be higher.

2.4 Which Mode?

We would like to have a block cipher mode that is suitable for **efficient** processing of encrypted data in databases. Due to reasons stated below, CTR mode emerges as the **best** choice among classic and proven to be secure block cipher modes for efficiently querying encrypted data. The properties of CTR mode that is useful for database encryption can be given as follows:

– **Efficient Implementation:** In all of our experiments, modified version of CTR (CTR4) was the fastest. Since the encryption of each block is independent, modern processor architecture's properties such as aggressive pipelining, multiple cores, and large number of registers can be utilized for even more efficient implementation [22]. For example, in [22], optimized version of CTR mode is four times faster than the optimized version of CBC mode. Also CTR mode is suitable for parallel processing.
– **Selective Decryption** CTR mode could be used to decrypt arbitrary parts of the plaintext. For example, for each tuple encrypted using CTR mode, during the selection operation, we may just decrypt the selection field first and decrypt the rest if the selection criteria is satisfied. To see why we can decrypt the arbitrary substring of a given ciphertext, consider the algorithm 3. In algorithm 3, first the counter values that are used to encrypt the $P[u..v]$ are calculated. Later on, using the counter values $\lfloor \frac{u}{b} \rfloor$ and $\lceil \frac{v}{b} \rceil$, encrypted counter values are created. Finally, the appropriate segment of the encrypted counter values are xored with the required part of the ciphertext (i.e. $C[u..v]$) to compute $P[u..v]$.
Other block cipher modes such as CBC, CFB, OFB defined for AES[15] in the FIPS-SP 800-38A[16] standard do not allow for selective decryption because the encryption of a block depends on the encryption of the previous block. This implies that we cannot selectively decrypt the required part of the data. ECB and CTR modes of operation encrypt each block independently. Unfortunately, ECB mode reveals the underlying distribution of the data. Therefore, it does not provide the desired level of security.
– **Preprocessing** In CTR mode, most costly part of the encryption and decryption (evaluating $E_K(ctr+i)$) can be done without seeing the data. Actually, we used this property in Experiment 5 and showed that this can reduce

the encryption cost significantly. This is not possible for all the modes except OFB mode but OFB mode does not allow selective decryption.

Algorithm 3 Decryption of arbitrary substring of ciphertext in CTR mode

Require: Ciphertext C with length l, initial counter value ctr, block-cipher length b, decryption start index u and decryption end index v
Set $S = E_K(ctr + \lfloor \frac{u}{b} \rfloor) || \ldots || E_K(ctr + \lceil \frac{v}{b} \rceil - 1)$
return $P[u..v] = C[u..v] \oplus S[(u - \lfloor \frac{u}{b} \rfloor \cdot b)..(v - \lfloor \frac{u}{b} \rfloor \cdot b)]$

3 A New Approach for Storing Encrypted Data in Database Pages

Granularity of encryption is another important design issue that needs to be considered for efficient storage of encrypted data. The overall database performance can potentially be affected by small changes in the design choices regarding the way of keeping the records in the pages. Tuple level and page level encryption are the most well known options for this purpose. Alternatively, we can use the mini-page approach suggested in [13].

In tuple level encryption, each tuple is encrypted or decrypted separately. If the database needs to retrieve some of the tuples, there is no need to decrypt all of the tuples in the table.

Page level encryption will correspond to a mechanism where a particular page is completely decrypted when buffer pool needs to access the page from the disk. After some modifications, the page is encrypted again and written to disk.

Mini page level for database encryption was first suggested for encrypted data in [13] based on the work of Ailamaki et.al [23]. In this technique, when a tuple is inserted into a page, its attributes are partitioned and stored in corresponding mini pages on the same page.

Deciding to use one of these approaches depends on two factors:

- the cost of the required block cipher operations under different types of queries
- the amount of modification required to implement these approaches in conventional databases

The mini page level encryption, as discussed in [13], is designed in such a way that the sensitive attributes of the records are encrypted with AES [15] in CBC mode and inserted at the beginning of the page. When a page is read from disk, just the first part of the page is decrypted. Since all the sensitive attributes of the records are located at the beginning of the page, the decryption process continues until all sensitive attributes are decrypted.

Although this idea is a neat solution for encrypting sensitive data, there are two issues that are not addressed. The first issue is that it does not allow selective decryption since the encryption mode is selected as CBC mode. When a projection query needs to get specific attributes among the sensitive attributes, all of the encrypted attributes should be decrypted. This cost will linearly increase if the length of the sensitive attributes increases proportionally to the total length of the records.

Secondly, the mini page level encryption requires significant amount of modification in the page structure of conventional databases. This is because, the attributes of the records are kept in two different parts of the page rather than in a consecutive order.

In the page level encryption, the whole of the page is encrypted before writing into disk and decrypted before loading into buffer pool. If all the data needs to be kept secret, this level of encryption is appropriate. In addition to that, implementing such an approach requires less modification in the page structure of existing databases. However, this is not preferable since it unnecessarily encrypts nonsensitive data along with sensitive data.

Because of the problems discussed above, we propose a new encryption method which we refer to as page level CTR. This method basically utilizes the selective decryption property of CTR4 and combines the useful aspects of page level and tuple level encryption methods. Unlike the mini page approach, sensitive and nonsensitive attributes of the records are kept consecutively. The decryption is performed for each sensitive attribute of the records separately after a page is read from the disk. Therefore, it resembles to tuple level approach.

In our encryption method, we propose using CTR4 for cryptographic operations. So we need to somehow store the counter values in the page. Since each counter value requires a 16-byte of location, keeping one counter value for each record will entail a nonnegligible storage cost. Fortunately, we can use one counter value per page instead of one record. With respect to storage of counter values, our page level CTR approach is similar to page level encryption. Also we modified and optimized the increment function of CTR mode to increment counter values as much as needed at a time, instead of just one by one.

The page structure of the proposed method is illustrated in figure 3 with an example, where we assume that a projection query requires to read the encrypted attributes of two consecutive records. The start index of the attribute of record 1 is $\alpha_1 = 220$ bytes and of record 2 is $\alpha_2 = 430$ bytes, with respect to the beginning of the page. Before starting the decryption, the counter value of the page is read from the beginning of the page. Then this value is incremented by δ_1 where $\delta_1 = \lfloor \frac{\alpha_1}{\beta} \rfloor = \lfloor \frac{220}{16} \rfloor = 13$ with the optimized increment function of page level CTR approach ($\beta = 16$ since the AES block size is 16 bytes). After finding the necessary counter value, the CTR4 is used to decrypt the sensitive attribute of record 1. Then, the same operation is repeated for record 2. After reading the counter value from the beginning of the page, it is incremented by δ_2 where $\delta_2 = \lfloor \frac{\alpha_2}{\beta} \rfloor = \lfloor \frac{430}{16} \rfloor = 26$. Using this value, the sensitive attribute of record 2 is decrypted with CTR4.

Here, we illustrated decrypting the sensitive attributes of two records in a page. However, this operation will be repeated for all records in every page, if there is a projection query that needs to read all of the records in a table.

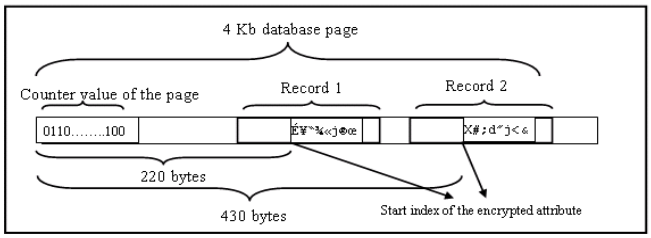

Fig. 3. Illustration of Page Level CTR approach

In the next section, the performance of mini page and page level CTR methods are compared based on our experiment results. In these experiments, we observed that selective decryption aspect of page level CTR causes significant performance gain under different query types.

3.1 Experiments and Analyses

In order to compare the performance of Mini page and Page level CTR methods, we conducted some experiments. The first experiment is implemented to analyze the performance when all incoming queries to database are projection queries. The second experiment differs from experiment 1 since the queries are selection queries.

In both of these experiments it is assumed that the tuple lengths are 500 bytes and 70% of the tuples (350 byte/record) are encrypted. In each run, a 512 MB file is read sequentially from the disk and processed. In each read operation, 4 KB data is read from the disk since it is the default page size in many DBMSs.

The mini page method is implemented as it is discussed in the previous section. When a page is read from the disk, only the first part of the page is decrypted. Since all the sensitive attributes of the records are located at the beginning of the page, the decryption process continues until all sensitive attributes are decrypted.

On the other hand, page level CTR method is implemented by consecutively locating each record, which has sensitive and nonsensitive attributes. As it is discussed before, when a particular sensitive attribute of a record needs to be read, the counter value of that page is incremented as much as needed. Then CTR4 is used to decrypt the required sensitive attribute.

Projection Experiments: To observe the selective decryption property of page level CTR encryption method, we wanted to analyze the performance of

projection queries in this experiment. As it is shown in figure 4, in the page level CTR method, the time required to decrypt the encrypted projection attribute is proportional to the length of the attribute. However, it is independent in the mini page level since this method decrypts all sensitive attributes to read the projection attributes of tuples. In contrast, page level CTR, decrypts as much data as needed to be decrypted. Therefore, both of the techniques require almost the same amount of time when the projection attribute is the only sensitive attribute in the record.

As a result of this experiment, we observed that, compared to mini page level encryption, page level CTR has a significant impact on reducing the cost of projection queries.

Selection Experiments: In projection experiments of page level CTR we were decrypting a certain amount of data for each record. However, in this experiment, in addition to selection attribute, we may need to decrypt the rest of the other sensitive attributes if the selection criterion is satisfied during query processing. This can be illustrated via an example, in which we have a table T with attributes A1, A2, and A3, where A1 and A2 are encrypted but A3 is not. In order to process a query such as "SELECT * FROM T WHERE A1 = X", the attribute A1 of each tuple should be decrypted. If the result of the condition A1 = X is true, not only A1 but also A2 should be decrypted. So, the performance of page level CTR depends on the selection condition of each tuples. Because of this reason, we repeated the experiments for different probability values of selecting a tuple.

The results of the experiments are shown in Figure 5, 6, and 7 where the probability of selecting a tuple is 30%, 60% and 100% respectively.

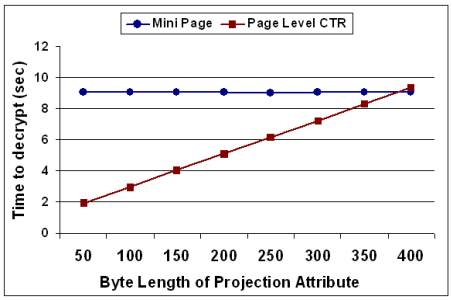

Fig. 4. Experiment results for projection queries

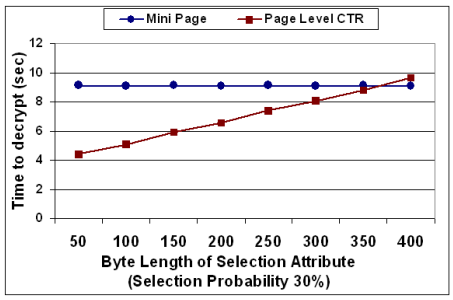

Fig. 5. Experiment results for selection queries

An important point to note is that the time required to decrypt tuples increases gradually when the probability of selecting tuples increases. When the probability is 100%, the cost of decrypting tuples becomes independent of the byte length of selection attribute. In addition, the cost of decrypting tuples in

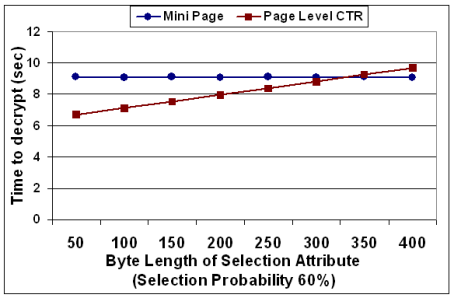

Fig. 6. Experiment results for selection queries

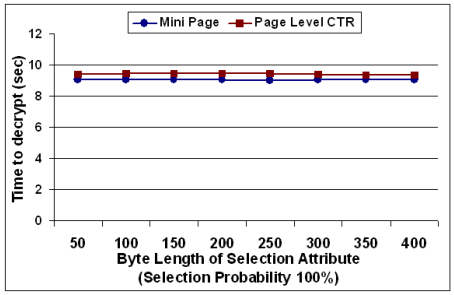

Fig. 7. Experiment results for selection queries

page level CTR is slightly more than the cost associated with the mini page as it is seen in Figure 7. The main reason for this is the overhead of extra index calculations in page level CTR approach.

In the selection experiment, we observed that selective decryption feature of page level CTR causes significant performance gain if the selection probability is low.

4 Discussion

We now discuss other encrypted data related issues in database management systems.

- **Key Management:** Using careful implementation of CTR mode, we can encrypt up to 2^{128} bits of data with a single key. Therefore, for most cases, we only need one encryption key per table. These encryption keys must be either stored in tamper-proof hardware or encrypted with a master key. If the master key option is chosen, during the system start, this master key can be loaded by the database administrator. If we do not want to trust the DB administrator with the master key, we can use classic threshold schemes to store the master key. For example, using a (k, t) secret sharing scheme, we can distribute this master key to t people and any k or more of them can come together to construct the master key [24].
  For security purposes, in the CTR mode, the same counter value should not be used for encrypting two different blocks. A simple way to solve this problem is to maintain a global counter value for each encryption key in use and update the counter value after each incrementation.
- **Insertion, Deletion, and Updates:** As mentioned above, counter values in CTR mode could not be used again. At the same time, we keep one initial counter value per page and calculate the counter values needed for selectively decrypting some attributes of the tuples using this initial counter values. When we need to update a part of the page, we need to encrypt the entire page with a different initial counter value.

– **Transaction Management** When there is an insert, delete, or update operation, DBMS will write a log to the log file. Therefore, to protect the sensitive data, we also need to encrypt the log file pages corresponding to encrypted tables. Otherwise, sensitive data values could be extracted from log files.

5 Conclusions

In this paper, we discussed the performance of different block cipher modes, under different encryption granularity and disk access patterns. Based on our experiments and analyses, we suggested a CTR based approach for encrypting data in DBMSs. We showed its potential for processing encrypted data faster by starting decryption process even before seeing the encrypted data. In addition to that, we proposed a page level encryption method by utilizing the selective decryption feature of CTR mode. Based on the results of our experiments, we compared the performance of the encryption method that we propose with the performances of other approaches and show its advantages under different query types. As a future work, we plan to implement our proposed approach using cryptographic accelerators in a distributed database environment.

Acknowledgments

We wish to thank Chris Clifton, Rakesh Agrawal and Sharad Mehrotra for helpful discussions.

References

1. Jr, T.Z.: An ominous milestone: 100 million data leaks. New York Times (December 18 2006)
2. Trinanes, J.A.: Database security in high risk environments. Technical report, governmentsecurity.org (2005) `http://www.governmentsecurity.org/articles/DatabaseSecurityinHighRiskEnvironments.php`.
3. : Standard for privacy of individually identifiable health information. Federal Register **67**(157) (August 14 2002) 53181–53273
4. : California database security breach notification act (September 2002) `http://info.sen.ca.gov/pub/01-02/bill/sen/sb\_1351-1400/sb\_1386\_bill\_20020926\_chaptered.html`.
5. Microsoft: Security features in microsoft sql server 2005. Technical report, Microsoft Corporation (2005) `http://www.microsoft.com/sql/2005/productinfo/`.
6. IBM: Ibm data encryption for ims and db2 databases. Technical report, IBM Corporation (2006) `http://www-306.ibm.com/software/data/db2imstools/db2tools/ibmencrypt.html`.
7. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order-preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD International C onference on Management of Data, Paris, France (June 13-18 2004)

8. Bayer, R., Metzger, J.K.: On the encipherment of search trees and random access files. ACM Trans. Database Syst. **1**(1) (1976) 37–52 http://doi.acm.org/10.1145/320434.320445.
9. Hardjono, T., Seberry, J.: Search key substitution in the encipherment of b-trees. In McLeod, D., Sacks-Davis, R., Schek, H.J., eds.: 16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings, Morgan Kaufmann (1990) 50–58
10. Hacigumus, H., Iyer, B.R., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the da tabase-service-provider model. In: Proceedings of the 2002 ACM SIGMOD In ternational Conference on Management of Data, Madison, Wisconsin (June 4-6 2002) 216–227 http://doi.acm.org/10.1145/564691.564717.
11. Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: Proceedings of the 30th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc. (2004)
12. Damiani, E., Vimercati, S.D.C., jodia, S.J., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relat ional dbmss. In: Proceedings of the 10th ACM conference on Computer and communications security, ACM Press (2003) 93–102 http://doi.acm.org/10.1145/948109.948124.
13. Iyer, B., Mehrotra, S., Mykletun, E., Tsudik, G., Wu, Y.: A framework for efficient storage security in rdbms. In: International Conference on Extending Database Technology (EDBT 2004). (2004)
14. Elovici, Y., Shmueli, E., nberg, R.W., Gudes, E.: A structure preserving database encryption sche me. In: Workshop on Secure Data Management in a Con nected World (SDM'04). (August 30 2004) http://www.extra.research.philips.com/sdm-workshop/RonenSDM.pdf.
15. 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.
16. : Recommendation for block cipher modes of operation methods and techniques. Technical Report NIST Special Publication 800-38A, National Institute of Standards and Technology (2001) http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.
17. : Data encryption standard (des). Technical Report FIPS PUB 46-2, National Institutes of Standards and Technology (January 22 1988)
18. Schneier, B.: The blowfish encryption algorithm. Dr. Dobb's Journal (April 1994) 38–40
19. Lipmaa, H., Rogaway, P., Wagner, D.: Ctr-mode encryption. In: NIST, Computer Security Resource Center, First Modes of Operation Workshop. (2000) http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/lipmaa-ctr.pdf.
20. Cox, M., Engelschall, R., Henson, S., Laurie, B.: The OpenSSL Project http://www.openssl.org/.
21. IBM: Table Space Design http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004935.htm.
22. Lipmaa, H.: Idea: A cipher for muldimedia architectures? In: Stafford Tavares and Henk Meijer, editors, Selected Areas in Cryptography '98, Springer-Verlag (1998)
23. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: Proceedings of the 27th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc. (2001) 169–180
24. Shamir, A.: How to share a secret. Commun. ACM **22**(11) (1979) 612–613