A Framework for Efficient Storage Security in RDBMS

Bala Iyer[1], Sharad Mehrotra[2], Einar Mykletun[2], Gene Tsudik[2], and Yonghua Wu[2]

[1] IBM Silicon Valley Lab
balaiyer@us.ibm.com
[2] University of California, Irvine, Irvine CA 92697, USA
{yonghuaw,mykletun,sharad,gts}@ics.uci.edu

Abstract. With the widespread use of e-business coupled with the public's awareness of data privacy issues and recent database security related legislations, incorporating security features into modern database products has become an increasingly important topic. Several database vendors already offer integrated solutions that provide data privacy within existing products. However, treating security and privacy issues as an afterthought often results in inefficient implementations. Some notable RDBMS storage models (such as the N-ary Storage Model) suffer from this problem. In this work, we analyze issues in storage security and discuss a number of trade-offs between security and efficiency. We then propose a new secure storage model and a key management architecture which enable efficient cryptographic operations while maintaining a very high level of security. We also assess the performance of our proposed model by experimenting with a prototype implementation based on the well-known TPC-H data set.

1 Introduction

Recently intensified concerns about security and privacy of data have prompted new legislation and fueled the development of new industry standards. These include the Gramm-Leach-Bliley Act (also known as the Financial Modernization Act) [3] that protects personal financial information, and the Health Insurance Portability and Accountability Act (HIPAA) [4] that regulates the privacy of personal health care information.

Basically, the new legislation requires anyone storing sensitive data to do so in encrypted fashion. As a result, database vendors are working towards offering security- and privacy-preserving solutions in their product offerings. Two prominent examples are Oracle [2] and IBM DB2 [5]. Despite its importance, little can be found on this topic in the research literature, with the exception of [6], [7] and [8].

Designing an effective security solution requires, among other things, understanding the points of vulnerability and the attack models. Important issues

E. Bertino et al. (Eds.): EDBT 2004, LNCS 2992, pp. 147–164, 2004.

include: (1) selection of encryption function(s), (2) key management architecture, and (3) data encryption granularity. The main challenge is to introduce security functionality without incurring too much overhead, in terms of both performance and storage. The problem is further exacerbated since stored data may comprise both sensitive as well as non-sensitive components and access to the latter should not be degraded simply because the former must be protected.

In this paper, we argue that adding privacy as an afterthought results in suboptimal performance. Efficient privacy measures require fundamental changes to the underlying storage subsystem implementation. We propose such a storage model and develop appropriate key management techniques which minimize the possibility of key and data compromise. More concretely, our main contribution is a new secure DBMS storage model that facilitates efficient implementation. Our approach involves grouping sensitive data, in order to minimize the number of necessary encryption operations, thus lowering cryptographic overhead.

Model: We assume a client-server scenario. The client has a combination of sensitive and non-sensitive data stored in a database at the server, with the sensitive data stored in encrypted form. Whether or not the two parties are co-located does not make a difference in terms of security. The server's added responsibility is to protect the client's sensitive data, i.e., to ensure its confidentiality and prevent unauthorized access. (Note that maintaining availability and integrity of stored data is an entirely different requirement.) This is accomplished through the combination of encryption, authentication and access control.

Trust in Server: The level of trust in the database server can range from fully trusted to fully untrusted, with several intermediate points. In a fully untrusted model, the server is not trusted with the client's cleartext data which it stores. (It may still be trusted with data integrity and availability.) Whereas, in a fully trusted model, the server essentially acts as a remote (outsourced) database storage for its clients.

Our focus is on environments where server is partially trusted. We consider one extreme of fully trusted server neither general nor particularly challenging. The other extreme of fully untrusted server corresponds to the so-called "Database-as-a-Service" (DAS) model [9]. In this model, a client does not even trust the server with cleartext queries; hence, it involves the server performing encrypted queries over encrypted data. The DAS model is interesting in its own right and presents a number of challenges. However, it also significantly complicates query processing at both client and server sides.

1.1 Potential Vulnerabilities

Our model has two major points of vulnerability with respect to client's data:

- **Client-Server Communication:** Assuming that client and server are not co-located, it is vital to secure their communication since client queries can involve sensitive inputs and server's replies carry confidential information.

– **Stored Data:** Typically, DBMS-s protect stored data through access control mechanisms. However, as mentioned above, this is insufficient, since server's secondary storage might not be constantly trusted and, at the very least, sensitive data should be stored in encrypted form.

All client-server communication can be secured through standard means, e.g., an SSL connection, which is the current *de facto* standard for securing Internet communication. Therefore, communication security poses no real challenge and we ignore it in the remainder of this paper. With regard to the stored data security, although access control has proven to be very useful in today's databases, its goals should not be confused with those of data confidentiality. Our model assumes potentially circumvented access control measures, e.g., bulk copying of server's secondary storage. Somewhat surprisingly, there is a dearth of prior work on the subject of incorporating cryptographic techniques into databases, especially, with the emphasis on efficiency. For this reason, our goal is to come up with a database storage model that allows for efficient implementation of encryption techniques and, at the same time, protects against certain attacks described in the next section.

1.2 Security and Attack Models

In our security model, the server's memory is trusted, which means that an adversary can not gain access to data currently in memory, e.g., by performing a memory dump. Thus, we focus on protecting secondary storage which, in this model, can be compromised. In particular, we need to ensure that an adversary who can access (physically or otherwise) server's secondary storage is unable to learn anything about the actual sensitive data.

Although it seems that, mechanically, data confidentiality is fairly easy to obtain in this model, it turns out not be a trivial task. This is chiefly because incorporating encryption into existing databases (which are based on today's storage models) is difficult without significant degradation in the overall system performance.

Organization: The rest of the paper is organized as follows: section 2 overviews related work and discusses, in detail, the problem we are trying to solve. Section 3 deals with certain aspects of database encryption, currently offered solutions and their limitations. Section 4 outlines the new DBMS storage model. This section also discusses encryption of indexes and other database-related operations affected by the proposed model. Section 5 consists of experiments with our prototype implementation of the new model. The paper concludes with the summary and directions for future work in section 6.

2 Background

Incorporating encryption into databases seems to be a fairly recent development among industry database providers [2] [5], and not much research has been de-

voted to this subject in terms of efficient implementation models. A nice survey of techniques used by modern database providers can be found in [10].

Some recent research focused on providing database as a service (DAS) in an untrusted server model [7] [9]. Some of this work dealt with analyzing how data can be stored securely at the server so as to allow a client to execute SQL queries directly over encrypted tuples. As far as trusted server models, one approach that has been investigated involves the use of tamper resistant hardware (smart card technology) to perform encryption at the server side [8].

2.1 Problems

The incorporation of encryption within modern DBMS's has often been incomplete, as several important factors have been neglected. They are as follows:

Performance Penalty: Added security measures typically introduce significant computational overhead to the running time of general database operations. This performance penalty is due mainly to the underlying storage models. It seems difficult to find an efficient encryption scheme for current database products without modifying the way in which records are stored in blocks on disk. The effects of the performance overhead encountered by the addition of encryption has been demonstrated in [10], where a comparison is performed among queries performed on several pairs of identical data sets, one of which contains encrypted information while the other does not.

Inflexibility: Depending on the encryption granularity, it might not be feasible to separate sensitive from non-sensitive fields when encrypting. For example, if row level encryption is used and only one out of several attributes needs to be kept confidential, a considerable amount of computational overhead would be incurred due to un-necessary encryption and decryption of all other attributes. Obviously, the finer the encryption granularity, the more flexibility is gained in terms of selecting the specific attributes to encrypt. (See section 3.2 for a discussion of different levels of encryption granularity.)

Meta data files: Many vendors seem content with being able to claim the ability to offer "security" along with their database products. Some of these provide an incomplete solution by only allowing for the encryption of actual records, while ignoring meta-data and log files which can be used to reveal sensitive fields.

Unprotected Indexes: Some vendors do not permit encryption of indexes, while others allow users to build indexes based on encrypted values. The latter approach results in a loss of some of the most obvious characteristics of an index – range searches, since a typical encryption algorithm is not order-preserving. By not encrypting an index constructed upon a sensitive attribute, such as U.S. Social Security Number, record encryption becomes meaningless. (Index encryption is discussed in detail in section 4.6.)

3 Database Encryption

There are two well-known classes of encryption algorithms: *conventional* and *public-key*. Although both can be used to provide data confidentiality, their goals and performance differ widely. Conventional, (also known as symmetric-key) encryption algorithms require the encryptor and decryptor to share the same key. Such algorithms can achieve high bulk encryption speeds, as high as 100-s of Mbits/sec. However, they suffer from the problem of *secure key distribution*, i.e., the need to securely deliver the same key to all necessary entities.

Public-key cryptography solves the problem of key distribution by allowing an entity to create its own public/private key-pair. Anyone with the knowledge of an entity's public key can encrypt data for this entity, while only someone in possession of the corresponding private key can decrypt the respective data. While elegant and useful, public key cryptography typically suffers from slow encryption speeds (up to 3 orders of magnitude slower than conventional algorithms) as well as secure public key distribution and revocation issues.

To take advantage of their respective benefits and, at the same time, to avoid drawbacks, it it usual to bootstrap secure communication by having the parties use a public-key algorithm (e.g., RSA [11]) to agree upon a secret key, which is then used to secure all subsequent transmission via some efficient conventional encryption algorithm, such as AES [12].

Due to their clearly superior performance, we use symmetric-key algorithms for encryption of data stored at the server. We also note that our particular model does not warrant using public key encryption at all.

3.1 Encryption Modes and Their Side-Effects

A typical conventional encryption algorithm offers several modes of operation. They can be broadly classified as *block* or *stream* cipher modes.

Stream ciphers involve creating a key-stream based on a fixed key (and, optionally, counter, previous ciphertext, or previous plaintext) and combining it with the plaintext in some way (e.g, by xor-ing them) to obtain ciphertext. Decryption involves reversing the process: combining the key-stream with the ciphertext to obtain the original plaintext. Along with the initial encryption key, additional state information must be maintained (i.e., key-stream initialization parameters) so that the key-stream can be re-created for decryption at a later time.

Block ciphers take as input a sequence of fixed-size plaintext blocks (e.g., 128-bit blocks in AES) and output the corresponding ciphertext block sequence. It is usually necessary to pad the plaintext before encryption in order to have it align with the desired block size. This can cause certain overhead in terms of storage space, resulting in the some data *expansion*. A chained block cipher (CBC) mode is a blend of block and stream modes; in it, a sequence of input plaintext blocks is encrypted such that each ciphertext block is dependent on all preceding ciphertext blocks and, conversely, influences all subsequent ciphertext blocks.

We use a block cipher in the CBC mode. Reasons for choosing block (over stream) ciphers include the added complexity of implementing stream ciphers, specifically, avoiding re-use of key-streams. This complexity stems from the dynamic nature of the stored data: the contents of data pages may be updated frequently, requiring the use of a new key-stream. In order to remedy this problem, a certain amount of state would be needed to help create appropriate distinct key-streams whenever stored data is modified.

3.2 Encryption Granularity

Encryption can be performed at various levels of granularity. In general, finer encryption granularity affords more flexibility in allowing the server to choose what data to encrypt. This is important since stored data may include non-sensitive fields which, ideally, should not be encrypted (if for no other reason than to reduce overhead). The obvious encryption granularity choices are:

- **Attribute value:** smallest achievable granularity; each attribute value of a tuple is encrypted separately.
- **Record/row:** each row in a table is encrypted separately. This way, if only certain tuples need to be retrieved and their locations in storage are known, the entire table need not be decrypted.
- **Attribute/column:** a more selective approach whereby only certain sensitive attributes (e.g., credit card numbers) are encrypted.
- **Page/block:** this approach is geared for automating the encryption process. Whenever a page/block of sensitive data is stored on disk, the entire block is encrypted. One such block might contain one or multiple tuples, depending on the number of tuples fitting into a page (a typical page is 16 Kbytes).

As mentioned above, we need to avoid encrypting non-sensitive data. If a record contains only a few sensitive fields, it would be wasteful to use row- or page-level encryption. However, if the entire table must be encrypted, it would be advantageous to work at the page level. This is because encrypting fewer large pieces of data is always considerably more efficient than encrypting several smaller pieces. Indeed, this is supported by our experimental results in section 3.6.

3.3 Key Management

Key management is clearly a very important aspect of any secure storage model. We use a simple key management scheme based on a two-level hierarchy consisting of a single master key and multiple sub-keys. Sub-keys are associated with individual tables or pages and are used to encrypt the data therein. Generation of all keys is the responsibility of the database server. Each sub-key is encrypted under the master key. Certain precautions need to be taken in the event that the master key is (or is believed to be) compromised. In particular, re-keying strategies must be specified.

3.4 Re-keying and Re-encryption

There are two types of re-keying: periodic and emergency. The former is needed since it is generally considered good practice to periodically change data encryption keys, especially, for data stored over a long term. Folklore has it, that the benefit of periodic re-keying is to prevent potential key compromise. However, this is not the case in our setting, since an adversary can always copy the encrypted database from untrusted secondary storage and compromise keys at some (much) later point, via, e.g., a brute-force attack.

Emergency re-keying is done whenever key compromise is suspected or expected. For example, if a trusted employee (e.g., a DBA) who has access to encryption keys is about to be fired or re-assigned, the risk of this employee mis-using the keys must be considered. Consequently, to prevent potential compromise, all affected keys should be changed before (or at the time of) employee termination or re-assignment.

3.5 Key Storage

Clearly, where and how the master key is stored influences the overall security of the system. The master key needs to be in possession of the DBA, stored on a smart card or some other hardware device or token. Presumably, this device is somehow "connected" to the database server during normal operation. However, it is then possible for a DBA to abscond with the master key or somehow leak it. This should trigger emergency re-keying, whereby a new master key is created and all keys previously encrypted under the old master key are updated accordingly.

3.6 Encryption Costs

Advances in general processor and DSP design continuously yield faster encryption speeds. However, even though bulk encryption rates can be very high, there remains a constant start-up cost associated with each encryption operation. (This cost is especially noticeable when keys are changed between encryptions since many ciphers require computing a key schedule before actually performing encryption.) The start-up cost dominates overall processing time when small amounts of data are encrypted, e.g., individual records or attribute values.

Experiments: Recall our earlier claim that encrypting the same amount of data using few encryption operations with large data units is more efficient than many operations with small data units. Although this claim is quite intuitive, we still elected to run an experiment to support it. The experiment consisted of encrypting 10 Mbytes using both large and small unit sizes: blocks of 100-, 120-, and 16K-bytes. The two smaller data units represent average sizes for records in the TPC-H data set [13], while the last unit of 16-Kbytes was chosen as it is the default page size used in MySQL's InnoDB table type. We used MySQL to implement our proposed storage model; the details can be found in section 4.

Table 1. Many small vs. few large data blocks encrypted. All times in msec include initialization and encryption cost.

Encryption Alg	**100 Bytes * 100,000**	**120 Bytes * 83,3333**	**16 KBytes * 625**
AES	365	334	194
DES	372	354	229
Blowfish	5280	4409	170

We then performed the following operations: 100, 000 encryptions of the 100-byte unit, 83, 333 encryptions of the 120-byte unit, and 625 encryptions of the 16-Kbyte unit. Our hardware platform was a Linux box with a 2.8 Ghz PIV with 1-Gbyte of RAM. Cryptographic software support was derived from the well-known OpenSSL library [14]. We used the following three ciphers: DES [15], Blowfish [16], and AES [12], of which the first two operate on 8-byte data blocks, while AES uses 16-byte blocks. Measurements for encrypting 10-Mbytes, including the initialization cost associated with each invocation of the encryption algorithms, are are shown in Table 1.

As pointed out earlier, a constant start-up cost is associated with each algorithm. This cost becomes significant when invoking the cipher multiple times. Blowfish is the fastest of the three in terms of sheer encryption speed, however, it also incurs the highest start-up cost. This is clearly illustrated in the measurements of the encryption of the small data units. All algorithms display reduced encryption costs when 16-Kbyte blocks are used.

The main conclusion we draw from these results is that encrypting the same amount of data using fewer large blocks is clearly more efficient than using several smaller blocks. The cost difference is due mainly to the start-up cost associated with the initialization of the encryption algorithms. It is thus clearly advantageous to minimize the total number of encryption operations, while ensuring that input data matches up with the encryption algorithm's block size (in order to minimize padding). One obvious way is to cluster sensitive data which needs to be encrypted. This is, in fact, a feature of the new storage model described in section 4.2.

4 Partition Plaintext and Ciphertext (PPC) Model

The majority of today's database systems use the N-ary Storage Model (NSM) [17] which we now describe.

4.1 N-ary Storage Model (NSM)

NSM stores records from a database continuously starting at the beginning of each page. An offset table is used at the end of the page to locate the beginning of each record. NSM is optimized for transferring data to and from secondary storage and offers excellent performance when the query workload is highly selective

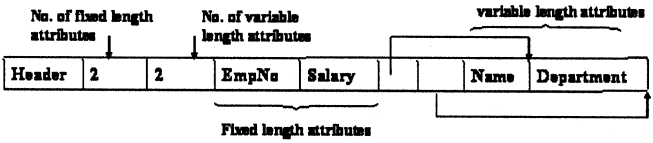

Fig. 1. NSM structure for our sample relation.

and involves most record attributes. It is also popular since it is well-suited for online transaction processing; more so than another prominent storage model, the Decomposed Storage Model (DSM) [1].

NSM and Encryption: Even though NSM has been a very successful RDBMS storage model, it is rather ill-suited for incorporating encryption. This is especially the case when a record has both sensitive and non-sensitive attributes. We will demonstrate, via an example scenario, exactly how the computation and storage overheads are severely increased when encryption is used within the NSM model. We assume a sample relation that has four attribute values: *EmpNo, Name, Department, and Salary.* Of these, only *Name* and *Salary* are sensitive and must be encrypted. Figure 1 shows the NSM record structure.

Since only two attributes are sensitive, we would encrypt at the attribute level so as to avoid unnecessary encryption of non-sensitive data (see section 3.2). Consequently, we need one encryption operation for each attribute-value.[1]

As described in section 3.1, using a symmetric-key algorithm in block cipher mode requires padding the input to match the block size. This can result in significant overhead when encrypting multiple values, each needing a certain amount of padding. For example, since AES [12] uses 16-byte input blocks, encryption of a 2-byte attribute value would require 14 bytes of padding.

To reduce thes costs outlined above, we must avoid small non-continuous sensitive plaintext values. Instead, we need to cluster them in some way, thereby reducing the number of encryption operations. Another potential benefit would be reduced amount of padding: per cluster, as opposed to per attribute value.

Optimized NSM: Since using encryption in NSM is quite costly, we suggest an obvious optimization. It involves storing all encrypted attribute values of one record sequentially (and, similarly, all plaintext values). With this optimization, a record ends up consisting of two parts: the ciphertext attributes followed by the plaintext (non-sensitive) attributes. The optimized version of NSM reduces padding overhead and eliminates multiple encryptions operations within a record. However, each record is still stored individually, meaning that, for each record, one encryption operation is needed. Moreover, each record is padded individually.

[1] If we instead encrypted at record or page level, non-sensitive attributes *EmpId* and *Department* would be also encrypted, thus requiring additional encryption operations. Even worse, for selection queries that only involve non-sensitive attributes, the cost of decrypting the data would still apply.

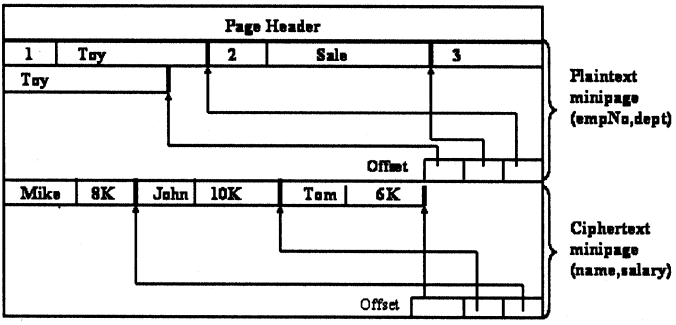

Fig. 2. Sample PPC page

4.2 Partition Plaintext Ciphertext Model (PPC)

Our approach is to cluster encrypted data while retaining NSM'a benefits. We note that, recently, Partition Attribute Across (PAX) model was proposed as an alternative to NSM. It involves partitioning a page into mini-pages to improve upon cache performance [18]. Each mini-page represents one attribute in the relation. It contains the value for this attribute of each record stored in the page. Our model, referred to as Partition Plaintext and Ciphertext (PPC), employs an idea similar to that of PAX, in that pages are split into two mini-pages, based on plaintext and ciphertext attributes, respectively. Each record is likewise split into two sub-records.

PPC Overview: The primary motivation for PPC is to reduce encryption costs, including computation and storage costs, while keeping the NSM storage schema. We thus take advantage of NSM while enabling efficient encryption. Implementing PPC on existing DBMS's that use NSM requires only a few modifications to page layout. PPC stores the same number of records on each page as does NSM.

Within a page, PPC vertically partitions a record into two sub-records, one of which contains the plaintext, while the other – ciphertext, attributes. Both sub-records are organized in the same manner as NSM records. PPC stores all plaintext sub-records in the first part of the page, which we call a *plaintext mini-page*. The second part of the page stores a *ciphertext mini-page*. Each mini-page has the same structure as a regular NSM page and records within two mini-pages are stored in the same relative order. At the end of each mini-page is an offset table pointing to the end of each sub-record. Thus, a PPC page can be viewed as two NSM mini-pages. Specifically, if a page does not contain any ciphertext, PPC layout is identical to NSM. Current database systems using NSM would only need to change the way they access pages in order to incorporate our PPC model.

Figure 2 shows an example of a PPC page. In it, three records are stored within a page. The plaintext mini-page contains non-sensitive attribute values EmpNo and Department and the ciphertext mini-page stores encrypted Name and Salary attributes. The advantage of encryption at the mini-page level can be

seen by observing that only one encryption operation is needed *per page*, and, of course, only one decryption is required when the page is brought into memory and any sensitive attributes are accessed.

The PPC page header contains two mini-page pointers (in addition to the typical page header fields) which include the starting addresses for plaintext and ciphertext mini-pages.

Buffer Manager Support: When a PPC page is brought into a buffer slot, depending upon the nature of the access, the page may first need to be decrypted. The buffer manager, besides supporting a *write bit* to indicate whether a page has been modified, also needs to support an *encryption bit* to determine whether the ciphertext mini-page has already been decrypted. Initially, when the page is brought into a buffer slot, its write bit is off, and its encryption bit is on. Each read or update request to the buffer manager indicates whether a sensitive field needs to be accessed.

The buffer manager processes read requests in the obvious manner: if the encryption bit is on, it requests the mini-page to be decrypted and then resets the encryption bit. If the bit is off, the page is already in memory in plaintext form. Record insertion, deletion and update are also obvious: the write bit is set, while the encryption bit is only modified if any ciphertext has to be decrypted.

Whenever a page in the buffer is chosen by the page replacement policy to be sent to secondary storage, the buffer manager first checks if the page's encryption bit is off and the write bit is on. If so, the cipher mini-page is first re-encrypted before being stored.

4.3 Analysis and Comparisons of the PPC Model

We now compare NSM with the proposed PPC model. As will be seen below, PPC outperforms NSM irrespective of the encryption level of granularity in NSM. Two main advantages of PPC are: 1) considerably fewer encryption operations due to clustering of sensitive data, and 2) overhead for queries involving only non-sensitive attributes.

NSM with attribute-level encryption: The comparison is quite straightforward. NSM with attribute-level encryption requires as many encryption operations per record as there are sensitive attributes. Records are typically small enough such that a large number can fit into one page, resulting in a large number of encryption operations per page. As already stated, only one decryption is per page is needed in the PPC model.

NSM with record-level encryption: One encryption is required for each record in the page. PPC requires only one encryption per page.

NSM with page level encryption: one encryption per page is required in both models. The only difference is for queries involving both sensitive and non-sensitive attributes. NSM performs an encryption operation regardless of the types of attributes, whereas, PPC only encrypts as necessary.

Optimized NSM: We mentioned the optimized NSM model in section 1. It is similar to NSM with record-level encryption (each record requires one encryption). It only differs from NSM in that extra overhead is incurred for non-sensitive queries. Again, PPC requires only one encryption operation per page as opposed to one per record for the optimized NSM.

Note that, for each comparison, the mode of access is irrelevant, i.e., whether records within the page are accessed sequentially or randomly, as is the case with the DSS and OLTP workloads. For every implementation, other than NSM with page-level encryption, one still needs to access and perform an encryption operation on the record within the page, regardless of how the record is accessed.

From the above discussion, we establish that PPC has the same costs as the regular non-encrypted NSM when handling non-sensitive queries. On the other hand, when sensitive attributes are involved, PPC costs a single encryption operation. We thus conclude that *PPC is superior to all NSM variants* in terms of encryption-related computation overhead.

Although we have not yet performed a detailed comparison of storage overheads (due to padding), we claim that PPC requires the same (or less) amount of space than any NSM variant. In the most extreme case, encrypting at the attribute level requires each sensitive attribute to be padded. A block cipher operating on 128-bit blocks (e.g., AES) would, on the average, add 64 bits of padding to each encrypted unit. Only encrypting at page level would minimize padding overhead, since only a single one unit is encrypted. This is the case for both NSM with page-level encryption and PPC.

4.4 Database Operations in PPC

Basic database operations (insertion, deletion, update and scan) in PPC are implemented similar to their counterparts in NSM, since each record in PPC is stored as two sub-records conforming to the NSM structure. During insertion, a record is split into two sub-records and each is inserted into its corresponding mini-page (sensitive or non-sensitive). As described in Section 4.2, the buffer manager determines when the ciphertext mini-page needs to be en/de-crypted. Implementation of deletion and update operations is straight-forward.

When running a query, two scan operators are invoked, one for each mini-page. Each scan operator sequentially reads a sub-record in the corresponding mini-page. If the predicate associated with the scan operator refers to an encrypted attribute, the scan operator indicates in its request to the buffer manager that it will be accessing an encrypted attribute. Scans could be implemented either using sequential access to the file containing the table, or using an index. Indexing on encrypted attributes is discussed in section 4.6 below.

4.5 Schema Change

PPC stores the schema of each relation in the catalog file. Upon adding or deleting an attribute, PPC creates a new schema and assigns it a unique version ID. All schema versions are stored in the catalog file. The header in the beginning

of each plaintext and ciphertext sub-record contains the schema version that it conforms to. The advantage of having schemas in both plaintext and ciphertext sub-records is that, when retrieving a sub-record, only one lookup is needed to determine the schema that a record conforms to.

Adding or deleting an attribute is handled similar to NSM, with two exceptions: (1) a previously non-sensitive attribute is changed to sensitive (i.e., it needs to be encrypted), and (2) a sensitive attribute is changed to non-sensitive, i.e., it needs to be stored as plaintext.

To handle the former, a new schema is first created and assigned a new version ID. Then, all records containing this attribute are updated according to the new schema. This operation can be executed in a lazy fashion (pages are read and translated asynchronously), or synchronously, in which case the entire table is locked and other transactions prevented from accessing the table until the reorganization is completed.

Changing an attribute's status from sensitive to non-sensitive can be deferred (for bulk decryption) or done in a lazy fashion, since it is generally not urgent to physically peform all decryption at once. For each accessed page, the schema comparison operation will indicate whether a change is necessary. At that time, the attribute will be decrypted and moved from the ciphertext to the plaintext mini-page.

4.6 Encrypted Index

Index data structures are crucial components of any database system, mainly to provide efficient range and selection queries. We need to assess potential impact of encryption on the performance of the index. Note that an index built upon a sensitive attribute must be encrypted, since it contains attribute values present in the actual relation.

There are basically two approaches to building an index based on sensitive attribute values. In the first, the index is based upon *ciphertext*, while, in the second, the intermediate index is based upon *plaintext* and the final index is obtained by encrypting the intermediate index. Based upon characteristics which make encryption efficient in PPC, we choose to encrypt at page level, thereby encrypting each node independently. Whenever a specific index is needed in the processing of a query, necessary parts of the data structure are brought into memory and decrypted. This approach provides full index functionality while keeping the index itself secure.

There are certain tradeoffs associated with either of the two approaches. Since encryption does not preserve order, the first approach is infeasible when index is used to process range queries. Note that exact-match selection queries are still possible if encryption is deterministic.[2] When searching for an attribute value, the value is simply encrypted under the same encryption key as used in the index

[2] Informally, if encryption is non-deterministic (randomized), the same cleartext encrypted twice yields two different ciphertexts. This is common practice for preventing ciphertext correlation and dictionary attacks.

before the search. The second approach, in contrast, can support range queries efficiently, albeit, with the additional overhead of decrypting index pages.

Given the above tradeoff, the first strategy appears preferable for hash-indices or B-tree indices over record identifiers and foreign keys where data access is expected to be over equality constraints (and not a range). The second strategy is better when creating B-trees where access may include range queries. Since the second strategy incurs additional encryption overhead, we discuss its performance in further detail.

When utilizing a B-tree index, we can assume that plaintext representation of the root node already resides in memory. With a tree of depth two, we only need to perform two I/O operations for a selection: one each for the node at level 1 and 2, and, correspondingly, two decryption operations. As described in section 3.6, we measure encryption overhead in the number of encryption operations. Since one decryption is needed for each accessed node, the total overhead is the number of accessed nodes multiplied by the start-up cost of the underlying encryption algorithm.

5 Experiments

We created a prototype implementation of the PPC model based on MySQL version 4.1.0-alpha. This version provides all the necessary DBMS components. We modified the InnoDB storage model, by altering its page and record structure to create plaintext and ciphertext mini-pages. We utilized the OpenSSL cryptographic library as the basis for our encryption-related code. For the actual measurements we used the Blowfish encryption algorithm. The experiments were conducted on a 1.8 Ghz PIV machine with 384MB of RAM running Windows XP.

5.1 PPC Details

Each page in InnoDB is, by default, 16KB. Depending on the number of encrypted attributes to be stored, we split the existing pages into two parts to accommodate the respective plaintext and ciphertext mini-pages. Partitioning of one record into plaintext and ciphertext sub-records only takes place when the record is written to its designated page. InnoDB record manager was modified to partition records and store them in their corresponding mini-pages.

If a record must be read from disk during a query execution, it is first accessed by InnoDB which converts it to MySQL record format. We modified this conversion in the record manager to determine whether the ciphertext part of the record is needed in the current query. If so, the respective ciphertext mini-page is first decrypted before the ciphertext and plaintext sub-records are combined into a regular MySQL record. The ciphertext mini-page is re-encrypted if and when it is written back to disk.

5.2 Overview of Experiments

For each experiment, we compared running times between *NSM with no encryption* (NSM), *NSM with page level encryption* (NSM-page), and *PPC* (PPC). Our goal was to verify that PPC would outperform NSM-page and, when no sensitive values are involved in a query, would perform similar to NSM. Lastly, when all attributes involved are encrypted, we expect PPC and NSM-page to perform roughly equally.

As claimed earlier, PPC handles mixed queries[3] very well. To support this, we created a schema where at least one attribute from each table – and more than one in larger tables (2 attributes in *lineitem* and 3 in *orders*) – was encrypted. Specifically, we encrypted the following attributes: *l_partkey, l_shipmode, p_name, s_acctbal, ps_supplycost, c_name, o_orderdate, o_custkey, o_totalprice, n_name, and r_name.*

We ran three experiments, all based on the TPC-H data set. First, we measured the loading time during bulk insertion of a 100-, 200-, and 500-MB database. Our second experiment consisted of running TPC-H query number 1, which is based only on the *lineitem* table, while varying the number of encrypted attributes involved in the query, in order to analyze PPC performance. Finally, we compared query response time of a sub-set of TPC-H queries, attempting to identify and highlight the properties of the PPC scheme.

5.3 Bulk Insertion

We compared bulk insertion loading time required for a 100-, 200-, and 500-MB TPC-H database for each of the three models. The schema described in section 5.2 was used to define the attributes to be encrypted in PPC. On the average, NSM-page and PPC incur a 24% and 15% overhead, respectively, in loading time as compared to plain NSM. As expected, PPC outperforms NSM-page since less data is encrypted (not all attributes are considered sensitive).

We note that a more accurate PPC implementation would perform some page reorganization if the relations contained variable length attributes, in order to make best use of space in the mini-pages. As stated in [18], the PAX model suffers a 2-10% performance penalty due to page reorganization, depending on the desired degree of space utilization in each mini-page. However, the PAX model creates as many mini-pages as there are records, while we always create only two.

5.4 Varying Number of Encrypted Attributes

In this experiment, we ran queries within a single table to analyze PPC performance while increasing the number of encrypted attributes. Query 1, which only involves table *lineitem*, was chosen for this purpose, and executed over a 200MB database. It contains 16 attributes, 7 of which are involved in the query. We then

[3] Queries involving both sensitive and non-sensitive attributes.

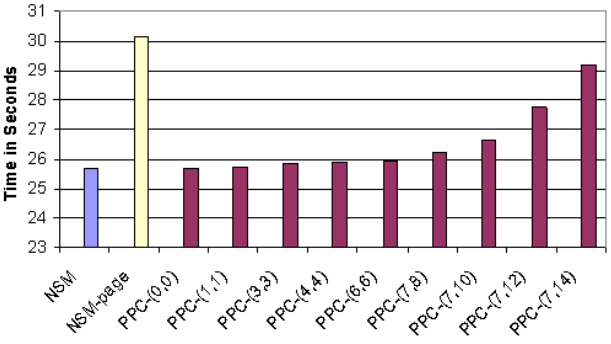

Fig. 3. Varying the number of encrypted attributes in TPC-H Query 1. PPC-(x, y) indicates that y attributes in lineitem were encrypted with x of them appearing in the query

computed the running time for NSM and NSM-page, both of which remained constant as the number of encrypted attributes varied. Eight different instances of PPC were run: the first having no encrypted attributes and the last having 14 (we did not encrypt the attributes used as primary keys). Figure 3 illustrates our results.

The results clearly show that the overhead incurred when encrypting additional attributes in PPC is rather minimal. As expected, PPC with no (or only a few) encrypted attributes exhibits performance almost identical to that of NSM. Also, as more attributes are encrypted, PPC performance begins to resemble that of NSM-page. Two last instances of PPC have relatively longer query execution times. This is due to encryption of the *lineitem* variables *l_shipinstruct and l_comment*, which are considerably larger than any other in the table.

5.5 TPC-H Queries

Recall that PPC and NSM-page each require only one encryption operation per page. However, NSM-page executes this operation whether or not there are encrypted attributes in the query. In the following experiment, we attempted to exploit the advantages of PPC, by comparing its performance with NSM-page when executing a chosen subset of TPC-H queries on a 200MB database. As in the bulk insertion experiment, we utilized a pre-defined schema (see section 5.2) to determine the attributes to encrypt when using PPC.

In figure 4, we refer to individual queries to highlight some of the interesting observations from the experiment. Queries 1 and 6 are range queries and, according to our PPC schema, have no sensitive attributes involved. The running time of NSM and PPC are, as expected, almost identical. However, since the *tables* involved contain encrypted attributes, NSM-page suffers from over-encryption and consequently has to decrypt records involved in the query, thereby adding to its running time. Due to the simplicity of these queries, the NSM-page encryption overhead is relatively small.

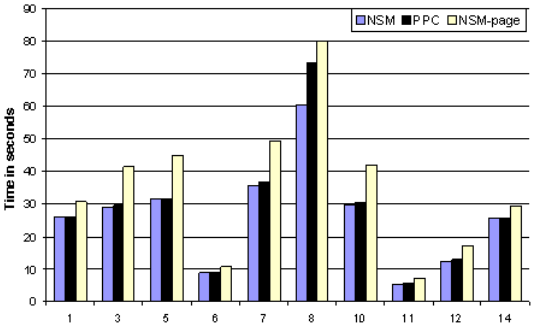

Fig. 4. Comparison of running times from selected TPC-H queries over a 200 MB database for NSM, PPC, and NSM-page.

Query 8 involved encrypted attributes from each of the three largest tables (*lineitem, partsupp, orders*), causing both PPC and NSM-page to incur relatively significant encryption-caused overhead. Again, PPC outperforms NSM-page since it has to decrypt less data. In contrast, query 10 involves encrypted attributes from four tables, only one of which is large (table *orders*). In this case, PPC performs well as compared to NSM-page, as the latter needs to decrypt *lineitem* in addition to other tables involved.

Overall, based on 10 queries shown in figure 4, PPC and NSM-page incur 6% and 33% overhead, respectively, in query response time. We feel that this experiment illustrated PPC's superior performance for queries involving both sensitive and non-sensitive attributes.

6 Conclusion

In this paper, we proposed a new DBMS storage model (PPC) that facilitates efficient incorporation of encryption. Our approach is based on grouping sensitive data in order to minimize the number of encryption operations, thus, greatly reducing encryption overhead. We compared and contrasted PPC with NSM and discussed a number of important issues regarding storage, access and query processing. Our experiments clearly illustrate advantages of the proposed PPC model.

References

1. Copeland, G. P., Khoshafian, S. F.: A Decomposition Storage Model. ACM SIGMOD International Conference on Management of Data. (1985) 268-269
2. Oracle Corporation: Database Encryption in Oracle9i. url=otn.oracle.com/deploy/security/oracle9i. (2001)
3. Department of Health and Human Services (U.S.). Gramm-Leach-Bliley (GLB) Act. FIPS PUB 81. url=www.ftc.gov/bcp/conline/pubs/buspubs/glbshort.htm. (1999)

4. Federal Trade Commission (U.S.): Health Insurance Portability and Accountability Act (HIPAA). url=www.hhs.gov/ocr/hipaa/privacy.html. (1996)
5. IBM Data Encryption for IMS and DB2 Databases, Version 1.1. url= http://www-306.ibm.com/software/data/db2imstools/html/ibmdataencryp.html. (2003)
6. He, J., Wang, M.: Cryptography and Relational Database Management Systems. Proceedings of the 5th International Database Engineering and Applications Symposium. (2001) 273-284
7. Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over Encrypted Data in the Database Service Provider Model. ACM SIGMOD Conference on Management of Data (2002)
8. Bouganim, L., Pucheral, P.: Chip-Secured Data Access: Confidential Data on Untrusted Servers. VLDB Conference, Hong Kong, China. (2002)
9. Hacigümüş, H., Iyer, B., Mehrotra, S.: Providing Database as a Service. ICDE. (2002)
10. Karlsson, J. S.: Using Encryption for Secure Data Storage in Mobile Database Systems. Friedrich-Schiller-Universitat Jena. (2002)
11. Rivest, R. L., Shamir, A., Adleman, L. M.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM. vol. 21, (1978)
12. NIST. Advanced Encryption Standard. FIPS PUB 197. (2001)
13. TPC Transaction Processing Performance Council. url=http://www.tpc.org
14. OpenSSL Project. url=http://www.openssl.org
15. NIST. Data Encryption Standard (DES). FIPS 46-3. (1993)
16. Schneier, B.: Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). Fast software Encryption, Cambridge Security Workshop Proceedings. (1993) 191-204
17. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 2nd Edition, WCB/McGraw-Hill. (2000)
18. Ailamaki, A., DeWitt, D. J., Hill, M. D., Skounakis, M.: Weaving Relations for Cache Performance. The VLDB Journal. (2001) 169-180