Cryptography and Relational Database Management Systems

Jingmin He and Min Wang

IBM T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532, USA
{jingmin, min}@us.ibm.com

Abstract

Security is becoming one of the most urgent challenges in database research and industry, and the challenge is intensifying due to the enormous popularity of e-business. In this paper we study database security from a cryptographic point of view. We show how to integrate modern cryptography technology into a relational database management system to solve some major security problems. Our study shows that cryptographic support is an indispensable ingredient for a modern RDBMS to provide a secure environment for storing and processing huge amount of business data.

1 Introduction

Surfing the net has become part of the daily life of our society. Internet shopping has become very popular and e-business is reshaping the way we do business. In the mean time, when using all the old and new services, people are worried about the privacy and integrity of their business data. Recently, there was an attack on a popular web site which resulted in the possible stealing of the credit card numbers of several thousand customers [18]:

> The safety of online commerce suffered another blow with the disclosure that the credit card database of a health products supplier was opened to hackers for a few hours this week. Word of the security breach at Global Health Trax Inc. comes as credit card companies are canceling thousands of cards because someone pilfered their numbers from CD Universe, a Web music seller. The card companies say the CD Universe case, uncovered Monday, has resulted in the largest mass-cancellation of cards they can recall.

Security is the primary concern of companies that want to do business on the Word Wide Web [9]. By design, internet is a free and open system, but it should also be a society where people can still maintain privacy and feel safe about their "internet properties".

A Relational DataBase Management System (RDBMS) plays a critical role in the new business. Huge amount of business data are gathered, stored and processed, which demands more powerful support from the backend database (DB) server. If we look at a purchase activity of a web customer and trace the data flow, we can see that there are two security issues to be addressed:

1. *Secure data transmission:* When a customer submits her/his confidential information (e.g., credit card number) through her/his web browser, the information should remain confidential on its way to the web server, the application server, and the backend DB server.
2. *Secure data storage and access:* When the confidential customer data arrive at the DB server, the data should be stored in such a way that only people with proper authorization can access them.

The secure transmission of data is well studied and well supported in today's e-business market. Almost all web browsers and web servers support SSL (Secure Socket Layer) [8] or TLS (Transport Layer Security) [7]. A credit card number is well protected on its way from a web browser to a web server via an SSL connection. However, once the data arrive at the backend, there is no sufficient support in storing and processing them in a secure way. For example, an RDBMS might not even provide an encryption mechanism to securely store the credit card numbers. Although the general problem of secure data storage is well studied, *the importance of secure data storage in an RDBMS has not been fully understood, and no serious work has been done on how to encrypt DB data.* We believe that it is vital to integrate cryptographic support into RDBMSs. Cryptographic support is another important dimension of database security. It is complementary to access control and both should be used to guide the storage and access of confidential data in a database system.

When we consider integrating cryptographic support into an RDBMS, there are three general approaches. The first approach is *loose coupling.* A third-party crypto service can be consulted by a database server and there are only minor changes on the server side. For example, a set of stored procedures can be pre-installed in the server. Each stored procedure provides a special crypto service to the database users by calling the crypto primitives supplied by the third party package. One example is an encryption PL/SQL package that encrypts a table column with a user supplied encryption key [19].

The second approach is *tight coupling.* A complete set of basic crypto primitives are built into the database server as a set of new SQL statements, together with the necessary control and execution context to ensure that those new SQL statements can be executed securely. This approach is a much harder task than the previous one in terms of implementation, but it is preferable in the long run. The reason is simple: loose coupling is likely to open many security holes.

The third approach is somewhere in between of the above two. To accommodate the urgent need for security enhancement, only a small subset of crypto primitives are integrated into the database server, based on which other services can be built using other database utilities such as user defined functions and stored procedures.

In this paper we take the third approach. We focus on two problems: how to enhance the access control mechanism to make user management more secure and close some major security holes of current RDBMSs, and how to support database encryption.

The contribution of this paper can be summarized as follows:

1. We introduce the important concept of *security dictionary* that can serve as the basis for many security services in a database system.
2. Based on the new concept of security dictionary, we show how to enhance the security of the current user management mechanism deployed in all RDBMSs.
3. We propose several database encryption methods and show how to do key management, again based on the security dictionary.

The rest of the paper is organized as follows: In Section 2 we analyze the security features implemented in current RDBMSs and point out their weakness. In Section 3 we introduce the important concept of *security dictionary.* Based upon the new concept, in Section 4 we show how the security of the user management subsystem can be enhanced. We propose several secure database encryption methods in Section 5 and make concluding remarks in Section 6.

2 The Problem

Although access control has been deployed as a security mechanism almost since the birth of large database systems, for a long time security of a DB was considered an *additional* problem to be addressed when the need arose, and after threats to the secrecy and integrity of data had occurred [3]. Now many major database companies are adopting the loose coupling approach and adding "optional" security support to their products. The approach of adding security support as an optional feature is not very satisfactory, since it would always penalize the system performance, and more importantly, it is likely to open new security holes.

Database security is a wide research area [6, 3] and includes topics such as statistical database security [1], intrusion detection [14], and most recently privacy-preserving data mining [2]. In this section we focus on the topics of user management, access control and database encryption. We briefly review how they are supported in current RDBMSs, analyze their security, and point out the potential problems.

2.1 Use Management

The first security-related component in an RDBMS (and actually in most systems) is user management. A user account needs to be created for anyone who wants to access database resources. However, how to maintain and manage user accounts is not a trivial task.

User management includes user account creation, maintenance, and user authentication. A *DBA* (DataBase Administrator) is responsible for creating and managing user accounts. When a DBA creates an account for user Alice, she/he also specifies how Alice is going to be authenticated, for example, by using a database password. The accounts and the related authentication information are stored and maintained in system catalog tables. When a user logs in, she/he must be authenticated in the exact way as specified in her/his account record. However, there is a security hole in this process.

Let us look at a concrete example from a major commercial RDBMS product [Oracle]. Suppose a DBA creates an account for user Alice by doing the following:

```
CREATE USER Alice
IDENTIFIED BY 'mypass';
```

Suppose the hash value of 'mypass' is '1A2B3C4D5E6F7G8H'. It is the hash value of a password that is stored in the database catalog table. The DBA can "become" user Alice anytime by going through the following steps:

1. Query the data dictionary to obtain the current encrypted password for Alice.
2. Change Alice's password by running the following command:

   ```
   ALTER USER Alice
   IDENTIFIED BY 'temppass';
   ```

3. Access Alice's account using the temporary password 'temppass'.
4. Reset Alice's password back to its original value obtained in Step 1 by running the following command:

   ```
   ALTER USER Alice
   IDENTIFIED BY VALUES
     '1A2B3C4D5E6F7G8H';
   ```

In other words, a DBA can impersonate any other user by changing (implicitly or explicitly) the system catalogs and she/he can do things on a user's behalf without being authorized/detected by the user, which is a security hole. As we will see in Section 5.4, a DBA's capability to impersonate other users would allow her/him to access other users' confidential data even if the data are encrypted.

2.2 Access Control

Access control is the major security mechanism deployed in all RDBMSs. It is based upon the concept of privilege. A subject (i.e., a user, an application, etc.) can access a database object if the subject has been assigned the corresponding privilege. Access control is the basis for many security features. Special views and stored procedures can be created to limit users' access to table contents.

However, a DBA has all the system privileges. Because of her/his ultimate power, a DBA can manage the whole system and make it work in the most efficient way. In the mean time, she/he also has the capability to do the most damage to the system.

2.3 Encryption

Current RDBMSs provide limited support for data encryption. Data are stored in tables in the form they are loaded, mostly in their plaintext form, which is not sufficient to meet high level security and privacy requirement.

Let us consider the following example. Suppose there is a database table Customer created by the following SQL statement:

```
CREATE TABLE Customer
  (userid       integer PRIMARY KEY
  lastname      varchar(25),
  firstname     varchar(25),
  ccnum         char(16)
  );
```

where column ccnum records customers' credit card numbers. The following statement creates a new record for customer John Smith whose credit card number is 1111222233334444:

```
INSERT INTO Customer VALUES
  (100, 'Smith', 'John', '1111222233334444');
```

The credit card number will be stored in its original plain-text form. The owner of table Customer or anyone with appropriate privilege can read this number with a simple **SELECT** statement. For convenience, we assume user Alice is the owner of table Customer. Note that Alice might be a pseudouser whose existence is to ensure that an application can run smoothly and her account is created by a DBA.

If the table Customer is stored in operating system files, the whole table can be protected by using the file-permission mechanism of the operating system. Only users with the correct permissions can access the file and thus the table [1]. The problem with this approach is that file protection restricts the access to the whole table, not just a particular column, and thus would make the database hard to use. Another problem is that database security is tied up with the underlying operating system which itself is rather vulnerable to both insider and outsider attacks. For example, a DBA, with the help of a system administrator, can still gain access to any data.

A variation of the above approach is to detach the physical medium that is used for storing the data from the system and keep it in a safe place. A similar approach is to encrypt the whole operating system file for the table (plus possible index files). Obviously neither of them is an attractive alternative, and we will not elaborate on them in this paper.

Recently a major database vendor has started to support database encryption [19]. It provides a PL/SQL package to encrypt/decrypt data. In using the package, key management must be handled programmatically by an outside application server. This

[1] In some cases the database server itself manages its own disk space, and the operating system mechanism cannot be used.

is a loose-coupling approach, and a standard PL/SQL package suffers from the same drawback as any other database object, that is, a DBA can drop it first, and recreate one with the same name but with built-in trapdoors. Through those trapdoors, a DBA can easily get hold of any confidential information. Therefore, the PL/SQL package can not support truly secure database encryption.

Another more serious problem with the loose coupling approach is that a database is used only as a passive data repository. When encrypted data need to be processed, they need to be fetched out of the database by another application server first. The database engine itself can do nothing useful about them. For example, indexing on an encrypted column is impossible, and no useful database operation can be performed against encrypted data.

2.4 The Role of DBAs

From an administration point of view, a DBA is playing an important and positive role. However, when security and privacy become a big issue, we cannot simply trust particular individuals to have total control over other people's secrecy. This is not just a problem of trustiness, it is a principle.

Technically, if we allow a DBA to control security without any restriction, the whole system becomes vulnerable because if the DBA is compromised, the security of the whole system is compromised, which would be a disaster. On the other hand, if we have a mechanism in which each user could have control over his/her own secrecy, the security of the system is maintained even if some individuals do not manage their security properly.

2.5 Notations

To simplify subsequent discussions, we introduce some notations. We use E and D to represent encryption and decryption operation, respectively. We will not specify what particular encryption/decryption functions are used. The pair (E, D) might be a symmetric cryptosystem (like DES [16]) or a public key system (like RSA [22]). For any plaintext m and key k, $E(k, m)$ is the ciphertext after applying encryption operation to m with the key k, and $D(k, c)$ is the result after applying decryption operation to the ciphertext c with a key k.

We use H to denote a secure hash function, e.g., H might be *SHA* [15] or *MD5* [21]. For an input x of arbitrary length, $H(x)$ will produce an output y of a specified length. Usually the length of y is smaller than that of x. We use $RAND$ to denote a (pseudo)random number generator that is cryptographically strong [13, 5]. Given a seed s, $RAND(s)$ will produce a random number whose length can be specified in advance. Usually the length of the random number is larger than that of the seed s. When no seed is given, $RAND()$ also returns a (pseudo)random number. (Please refer to [25] for more details.)

3 Concepts

In this section we introduce several important concepts that are the basis of our database security study.

3.1 Secure Operating Environment

If an operation involves any secret data, that operation should be conducted in an environment that would not cause secret leak. We call such an environment a *Secure Operating Environment (SOE)*. A good example of SOE is any smart card application where a user's private secret stays in and never gets out of a tamper-free card [10, 20]. If an operation involves the secret, the operation itself will be done inside the card and only the operation result will be output to the outside world.

However, for an RDBMS, things are much more complicated because a database system is much more complicated than a simple smart card application where only a small amount of data is involved. A smart card is not adequate for an RDBMS. A better choice would be a secure processor such as the IBM Programmable Secure Coprocessor.

The current database engine internal can also be considered as a relatively good SOE. In general, database server processes are OS objects and are protected only by OS itself. If the server is running on a machine where a strong security policy has been implemented, then the database engine as a whole can be treated as an SOE.

There are obviously many issues surrounding SOE. For the purpose of this paper, we assume that the current database server processes form an SOE and is where our trust lies.

3.2 Security Dictionary

A traditional data dictionary stores all of the information that is used to manage the objects in a database. A data dictionary consists of many catalog tables and views. It is generally recommended that users (including DBAs) do not change the contents of a catalog table manually. Instead, those catalogs will be maintained by the DB server and updated only through the execution of system commands. However, a DBA can still make changes in a catalog table if she/he wants to do so.

To prevent unauthorized access to important security-related information, we introduce the concept of *security catalog*. A security catalog is like a traditional system catalog but with two security properties:

(a) It can never be updated manually by anyone, and (b) Its access is controlled by a strict authentication and authorization policy.

One example would be a table SEC_USER that records user account information. Some columns in the table store security-related information and are called *security columns* (*security fields*). Each security field has a corresponding security flag field that specifies how the value of the field can be accessed (particularly updated). For example, a password field could be a security field, and its security flag could be set to "updatable by anyone with appropriate access privilege to the SEC_USER table", "updatable by the defined user only", or "never updatable by anyone". In the second case, only a user herself/himself can change her/his password. We will discuss the detail in Section 4.

A *security dictionary* consists of all the security catalogs. Security dictionary is an important security concept. Basically, it provides a secure and reliable repository where information can be safely stored. It is not a secret box that people cannot have a glance into its content (since that would be too much for any system to support). However, its update is under strict security control.

The implementation of a security dictionary is flexible. One approach is to implement it within a database server. The server internally control the access to the security dictionary, probably based on some system security configuration that is derived from an enterprise security policy, which is not necessarily done by a DBA. Another approach is to implement the security dictionary as a service outside of the database server. For the sake of simplicity and convenience, we assume the first approach. Again, we emphasize that the access to a security catalog is controlled by a specific security policy associated with it.

The concept of security dictionary is different from that of a virtual private database [19]. A virtual private database is actually a loose way to shift the responsibility of refining access control from an application server to the database server, and everything in the database server is still under the control of a traditional access control mechanism. A DBA still has all the system privileges. By deploying a security dictionary, everyone (including DBAs) can be restricted in accessing other people's critical security information, and only in this way can an RDBMS provide a real secure operating environment for all users.

The integrity of secure catalogs can be well maintained. First of all, the access to a security field is strictly controlled by its corresponding security flag, and its content cannot be modified without its security flag be changed first. Second, a security flag can be put under the control of an SOE. Or, a security policy can be established that does not allow the change of a security flag after it is first created (see Section 4.) Third, the hash value of a security column (or even the security flag column) can be computed and stored in an SOE so that any accidental change (e.g., caused by physical disk damage) can be detected.

Another issue is backup. Critical security information should be backed up so that a reasonable recovery can be expected. An SOE should have its own backup mechanism. A subtle problem is that of a lost secret key. For example, some data are protected by a secret key: either they are encrypted using the key, or the hash value of the data is computed using the secret key. If the key is lost, the data become useless. This is the typical key recovery problem in cryptography and many approaches can be taken.

In the next two sections, we elaborate on the use of security catalogs with two important database security tasks: (a) user management, and (b) encryption and key management. We discuss relevant security issues and outline the general design principles.

4 User Management

To access database resources, a user must have an account with the database. User account management is the basis for the overall database system security. A DBA has the responsibility to create and maintain all DB user accounts, which is a large portion of her/his system administration effort. At the account creation time, the DBA specifies how the newly created user will be authenticated, and what system resources the user can use. When a user wants to connect to a database, she/he must identify herself/himself to the server and the server will verify her/his identity using the pre-specified authentication method. Current commercial RDBMSs support many different kinds of identification and authentication methods, among them are password-based authentication [12], host-based authentication [4, 12, 11], PKI (Public Key Infrastructure) based authentication [19], and other third party-based authentications such as Kerberos [17], DCE (Distributed Computing Environment [23]) and smart card [20]. Essentially, all methods rely on a secret known only to the connecting user.

It is vital that a user should have total control over her/his own secret. For example, only she/he should be able to change her/his password. Other people can change a user's password only if they are authorized to do so. In a DB system, a DBA can reset a user's password upon the user's request, probably because the user might have forgotten her/his password. However, as we have noticed before, the DBA can temporarily change a user's password without being detected and caught by the user, because the DBA has the capability to update (directly or indirectly) the system catalogs.

By using a security catalog, no one (including the DBA) would be able to manipulate other users' important security information, and no one can impersonate other people without being detected and caught. When a DBA creates a user account, besides specifying the usual account information, the DBA must also specify some security characteristics (whether and how this account can be modified) so that a specific security policy is associated with this account. All the account information is stored in a security catalog table SEC_USER that contains the following columns, among others:

userid User login name. No one is allowed to change this field.

auth_type How this user is authenticated. Possible values: db, os, sc (smart card), or any other supported authentication methods.

auth_flag Security flag, indicating if other users are allowed to update **auth_type** field of this record. Possible values: `'yes'`, `'no'`, or `'never'`.

passwd Hash value of password.

passwd_flag Security flag for passwd, indicating if other users are allowed to update **passwd** field of this record. Possible values: `'yes'`, `'no'`, or `'never'`.

updateby The userid who updated the passwd field most recently.

The default user for any SEC_USER record is the user whose identifier is specified in the **userid** field. The **auth_flag** (security flag) field takes three possible values: `'yes`, `'no'`, or `'never'`. Value `'yes'` means anyone with the privilege to access the table SEC_USER can update this record; value `'no'` means only the default user has the update privilege; value `'never'` means no one (including the default user) can update the record.

Let us consider a concrete example. Suppose a DBA creates an account for user Alice by running the following SQL statement:

CREATE USER `Alice`
IDENTIFIED BY `db` **UPDATE** `never`
PASSWORD `'mypass'` **UPDATE** `no`;

The statement creates a user with userid `'Alice'` and the newly created user will be authenticated by database password `'mypass'`. The only way Alice can login is to use the database password and the authentication method cannot be changed. No one except Alice herself can change her login password. The security portion of the SEC_USER record for Alice contains the following information, among others:

$(\texttt{userid} = \texttt{Alice}, \texttt{auth\_type} = \texttt{db},$
$\texttt{auth\_flag} = \texttt{never}, \texttt{passwd} = H(\texttt{'mypass'}),$
$\texttt{passwd\_flag} = \texttt{no}, \texttt{updateby} = \texttt{'\ '}).$

An empty `updateby` field means this record is newly created. Alice can change her password if a password utility is provided, or she can do so by using the following SQL statement:

ALTER USER `Alice`
PASSWORD `'newpass'`;

The server will check the identity of the command issuer (Alice) against the SEC_USER record for the user specified in the statement (Alice) and decide if the command is authorized. Since the `passwd_flag` value is `'no'` which means Alice herself can update her own password, the command is executed and Alice will have a new database password. However, if a DBA (assuming she/he is not Alice) issues the above statement, after checking the record for Alice in SEC_USER, the server knows the change is not authorized and thus the command is rejected. In case the `passwd_flag` value for Alice is `'never'`, even Alice herself is not allowed to change her password, which is usually not recommended because changing password periodically is considered a good security practice. Another interesting and troublesome case is what if Alice forgets her password. On the one hand, password management is a big topic on the users' side and many assistant tools can be utilized (like smart card, wallet manager, etc.). On the other hand, the database server does need to handle the case when it happens. One possible solution is to allow a DBA to change a user's password. When she/he creates the user account, she/he can specify the value of `passwd_flag` for Alice to be `'yes'`. In this case, besides Alice herself, the DBA is also allowed to alter her password. However, whenever someone changes Alice's password, that person's userid will be recored in Alice's SEC_USER record and Alice can query that record to make sure any update is authorized. Later, Alice may choose to change her `passwd_flag` to `'no'` by issuing the following statement:

ALTER USER `Alice`
PASSWORD UPDATE `no`;

After that, no one except Alice can change her password and in case Alice forgets her password, there is no recovery for it.

Another (possibly) better solution is to use secret sharing [24]. When creating a user account, the DBA could specify a threshold value t. Whenever a password is created/changed, the new password will be protected with n *shadows* ($n \geq t$). The shadows will be kept by n security officers. Later, when it becomes necessary to recover a lost password, t (or more) shadows can be gathered and used together to recover the secret password.

Although a DBA might not have the authority to alter a user's security characteristics, she/he can change other non-security parameters as usual. Furthermore, she/he can drop a user account. If she/he runs the command "**DROP USER** `Alice`;", the SEC_USER record for Alice will be deleted. The DBA may recreate another user with the name Alice, however, this Alice has no relationship whatsoever to any previous existing Alice. Especially, the new Alice cannot read any confidential data encrypted by the previous Alice. This will guarantee that a DBA cannot intrude Alice's privacy by recreating her (pseudo)identity in the server. (See Section 5 for more details.)

5 Encryption and Key Management

In this section we propose several database encryption methods and analyze their security. Please note that the encryption techniques we discussed here are all standard ones. We only adapt them for our use in a database environment based on the central mechanism of security catalogs.

The most important problem in using encryption/decryption is key management. When we consider incorporating encryption in a database server, there are two design issues:

1. There should be a way for a user to indicate that some data should be encrypted before storage.
2. There should be a way for a user to specify (explicitly or implicitly) a key that will be used for data encryption.

The two issues are not totally separated. When a user indicates that a table column should be encrypted, she/he may supply an encryption key at the same time, or supply the key later when actual data are loaded. The second issue is more important. It basically reduces to the problem of key management.

When defining a table, a user may specify explicitly that she/he wants the content of some columns to be encrypted when the data are loaded. She/He can do this by issuing the following **CREATE TABLE** statement:

```
CREATE TABLE Customer
   (userid    integer PRIMARY KEY
   lastname   varchar(25),
   firstname  varchar(25),
   ccnum      char(16) ENCRYPTION
              UPDATE no
   );
```

Whenever a credit card number is inserted into the table, it will be encrypted (by the database server) first and the encrypted version will be stored in the database. We assume user Alice is the creator (and thus the owner) of table `Customer`.

We need a new security catalog SEC_ENCRYPTION. Whenever a user creates a database object that is to be encrypted, a corresponding record is created in the security catalog SEC_ENCRYPTION. Each catalog record specifies a database object, its owner (the object creator), and a security flag that indicates how the object is to be updated. There is also an **updateby** field that store the identifier of the user who updated this record most recently. The pair (`owner`, `object`) forms a *primary key*. For example, for the above **CREATE TABLE** statement, the following record is inserted in SEC_ENCRYPTION:

(`Alice`, table '`Customer`', column '`ccnum`',
`enc_flag`, `updateby` = `Alice`).

The `enc_flag` specifies whether any user can update the record, and can take the values: '`yes`', '`no`', or '`never`'. The value '`yes`' means any user with the privilege to access table `Customer` can change the definition of the `ccnum` column; '`no`' means only Alice (the owner) can change the definition of `ccnum`; '`never`' means nobody can make the change. In the above example, only Alice can change the definition of column `ccnum` (e.g., she can drop the encryption requirement for this column).

Before any database object is altered, the security dictionary will be checked first. If the change violates the security specification for that object, the change request will be rejected. For example, if a DBA wants to change the definition of column `Customer(ccnum)` by issuing the following SQL statement:

```
ALTER TABLE Customer
MODIFY ccnum DROP ENCRYPTION;
```

the command will be rejected because the SEC_ENCRYPTION record for `Customer` shows that Alice is the owner of the table and only the owner can change the definition of column `ccnum`. However, if it is Alice who issues the above statement, the change will be made. There is one question that needs to be answered here. When the column definition changes, what about those credit card numbers that were already inserted and stored in encrypted form? There are two alternatives. One is that the old column values remain unchanged in the encrypted form. The other alternative is to decrypt all the old values, which would require Alice to provide correct decryption key(s). Similarly, when Alice changes a column by enabling the encryption option, she could either leave the old column values alone, or provide a key to encrypt them. Again, we will not elaborate on this in much detail here.

In the following subsections, we will show how to manage keys used for database encryption.

5.1 Password Based Encryption

All RDBMSs in the market can authenticate a user through a password mechanism. When a user types in the correct user name with a matching password, she/he is connected to the server and can start using any database resources she/he is granted privilege to.

The server can always keep a copy of the user password in memory during the session when a user is connected. This password will be used to do any necessary encryption for this particular user.

For example, suppose Alice's password is $'$`mypass`$'$. When Alice logs in, the database server will get a memory copy of this password. When Alice creates a new customer record by issuing

```
INSERT INTO Customer VALUES
    (100, 'Smith', 'John', '1111222233334444');
```

the database server will first encrypt the credit card number with key $'$`mypass`$'$ and then store the ciphertext in the table `Customer`.

Later, when Alice wants to get John Smith's credit card number, she runs the command

```
SELECT ccnum FROM Customer
WHERE userid = 100;
```

Since Alice must have logged in to run the SELECT statement, the server must have already obtained a copy of Alice's password $'$`mypass`$'$. The server will first decrypt the content of column `Customer(ccnum)` with $'$`mypass`$'$ and then return the original card number $D($'`mypass`', `Customer(ccnum)`$)$.

A much better approach is to use a variation of the user password. When a user logs in, her/his password is used as a seed to generate a *working key* that is used in all encryption operations. The advantage of this approach is that the user password is not used directly in the possibly frequent encryption operations. This approach can be pushed even further. For each column, a combination of the table name, column name and the password can serve as the seed for working key generation and thus different columns are protected with different keys. Furthermore, a unique row identifier can be incorporated into the working key generation process so that identical value appeared in the same column but different rows will be encrypted to different ciphertexts. For example, a working key for `Customer(ccnum)` can be generated by $k = H($'`mypass`', '`Customer`', `userid`, '`ccnum`'$)$.

Security The password based approach is secure. Once the encrypted version of a credit card number is stored in the table `Customer`, only a user with the correct password (i.e., Alice) can decrypt and thus access the original card number. A DBA might be able to temporarily replace the encrypted password with another one, but she/he still does not know Alice's password and thus still unable to read the credit card numbers.

One drawback of the password based approach is that when a user changes her/his password, all the encrypted data need to be decrypted with the old password and re-encrypted using the new password, which is a big performance overhead. Notice that changing password periodically is considered a good practice from the password security point of view. In the case when a user forgets her/his password, there is no way to recover unless the password has been guided by some secret recovery mechanism.

Password-based encryption relies on the fact that the database server can grab a user's login password. However, a database server might rely on the underlying operating system entirely for user authentication. The database server might direct the user login to the OS without bothering to get a copy of his password. In this case, the approach cannot be used.

5.2 Public Key Based Encryption

Public key and PKI can be used to do database encryption in a more robust and efficient way.

A PKI infrastructure is the basis for the effective use of *public key* technology. We assume there is a directory service (e.g., an LDAP server) that is consulted by the database server. Whenever needed, the database server can consult with the directory server to obtain a *certificate* of a public key for a particular user. Stored together with the public key certificate is the matching *private key* that is encrypted using the user's password.

For example, suppose Alice has a certificate $\mathtt{CERT_A}$ stored in the directory server. The certificate $\mathtt{CERT_A}$ certifies a public key $\mathtt{PK_A}$ with a matching private key $\mathtt{SK_A}$. In the directory server, the following record has all the key information for user Alice:

$$(\mathtt{Alice},\ \mathtt{CERT_A},\ E('\mathit{mypass}',\ \mathtt{SK_A}))$$

When Alice issues the following statement:

INSERT INTO `Customer` **VALUES**
(100, '`Smith`', '`John`', '1111222233334444');

the database server queries the directory server and obtains $\mathtt{CERT_A}$ and thus the public key $\mathtt{PK_A}$. $\mathtt{PK_A}$ can be used to encrypt a working key K that is generated by the system. The working key K is used to encrypt John Smith's credit card number, and K itself is encrypted using $\mathtt{PK_A}$. The encrypted working key might be stored in the directory server if its generation involves randomness. For example, the working key can be simply generated as $K = \mathtt{RAND}()$ and $E(\mathtt{PK_A}, K)$ will be stored in the directory server.

Later, when Alice wants to read John Smith's credit card number, she first needs to login to the database server. If the login is successful, the database server has a copy of Alice's password that can then be used to fetch her private key $\mathtt{SK_A}$, which in turn can be used to decrypt the encrypted working key. Finally, the working key can be used to decrypt the stored version of the credit card number.

When Alice creates another customer record, depending on the design, the DB server can use the same working key generated before to encrypt the new credit card number, or it generates a new working key. In the former case, the server can use Alice's password to decrypt and obtain the old working key K. In the later case, the server does key generation and encryption as before.

When Alice changes her password, only her private keys need to be re-encrypted and this can be done efficiently.

In the above approach, we use user passwords to encrypt private keys. Following is a modified approach that does not use user passwords.

In the directory server, user Alice's key record contains the following information:

$$(\mathtt{Alice},\ \mathtt{CERT_A}).$$

When Alice inserts John Smith's record into the `Customer` table, a *working key K* is generated to encrypt the column `Customer(ccnum)`, and a record of the following form is stored in the directory server:

$$(\mathtt{Alice},\ '\mathtt{Customer}',\ '\mathtt{ccnum}',\ E(\mathtt{PK_A}, K)).$$

When Alice wants to access John Smith's credit card number, she issues the following statement:

SELECT `ccnum` **FROM** `Customer`
WHERE `userid` = 100 **PRIVATE KEY** $\mathtt{SK_A}$;

That is, Alice explicitly supplies her private key. The database server first fetches the encrypted working key $E(PK_A, K)$ from the directory server, then uses the user supplied private key SK_A to decrypt it to obtain K, uses K to decrypt the encrypted credit card number $E(K,\ '1111222233334444')$ and finally returns the result.

When Alice creates a new customer record, if the design is such that the previously generated working key should be used, Alice needs to supply her private key when doing any insertion:

INSERT INTO `Customer` **VALUES**
(200, '`Case`', '`Steve`', '5555666677778888')
PRIVATE KEY $\mathtt{SK_A}$;

The DB server will fetch $E(\mathtt{PK_A}, K)$ from the directory server, decrypt it using the matching private key $\mathtt{SK_A}$ to obtain K, and then use K to encrypt the new credit card number.

Security The secure use of the private key $\mathtt{SK_A}$ guarantees the security of the approach. When Alice's password is used to protect her private key $\mathtt{SK_A}$, no one can get hold of it except Alice herself, and therefore no one except Alice can obtain the original working key to decrypt the credit card number.

In the non-password version, Alice dynamically supplies her private key $\mathtt{SK_A}$ and $\mathtt{SK_A}$ will never be stored in the database even in an encrypted form. Therefore, the method is secure.

If we store Alice's certified public key in a traditional database table or system catalog, a DBA might easily break in. For example, if the following record is stored in the database:

$$(\mathtt{Alice},\ \mathtt{CERT_A})$$

the DBA can update the second component and replace it with her/his own certificate that contains her/his public key, say $\mathtt{PK_C}$. Later, when Alice creates a new customer record, the credit card number will be encrypted using $\mathtt{PK_C}$, not $\mathtt{PK_A}$, and the DBA can easily read the encrypted credit card number.

5.3 Encryption Based on User-Supplied Keys

The most flexible database encryption approach is using keys dynamically supplied by the users. To accommodate this case, we extend the previous **CREATE TABLE** statement and add one more option:

```
CREATE TABLE Customer
   (userid          integer PRIMARY KEY
   lastname         varchar(25),
   firstname        varchar(25),
   ccnum            char(16) ENCRYPTION
                    [WITH KEY key_value]
   );
```

When **WITH KEY** `key_value` is present, `key_value` will be the default encryption key when any credit card number is inserted into the table. If a user supplies another key, that key will be used instead. The new SQL statement for this is:

```
INSERT INTO table_name VALUES
   value_specification [KEYS key_list]
```

where `key_list` is a list of keys separated by comma. The list elements correspond to the columns that are to be encrypted. For example, to create a new record for John Smith, Alice would use the following statement:

```
INSERT INTO Customer VALUES
   (100,'Smith', 'John', '1111222233334444')
   KEYS ('1234567890');
```

The database server will use the string '1234567890' as key to encrypt the credit card number and store the encrypted version in the database.

When Alice wants to access John Smith's credit card number, she issues the following statement:

```
SELECT ccnum FROM Customer
WHERE userid = 100 KEYS ('1234567890');
```

The general form of the extended SELECT statement is as follows:

```
SELECT projection_list ... (other clauses)
   [KEYS key_list];
```

where the elements of the key list correspond to the encrypted columns in the projection list.

The approach is self-contained. No directory service or certificate is needed. Basically, the task of key management is shifted to the application server, and the database server only provides the framework to do encryption/decryption.

One main advantage of this approach is that it is much easier to integrate with an existing database product. All the major commercial database server products in the market are very large software product. Any big change to their architecture would not be easily acceptable to the R&D teams very soon. However, adding or extending several SQL statements is a much easier task.

Security. Since a user dynamically supplies her/his working key, there is no chance for any other people to decrypt an encrypted credit card number and the method is secure.

5.4 Group Encryption

In this subsection we consider the case when a group of users want to share access to an encrypted column.

For example, when Alice creates the table `Customer`, she may want to allow Bob to read user credit card numbers also. One obvious solution is that Alice gives the working keys directly to Bob. If a working key is protected by Alice's public/private keys, there would be no way for Bob to gain access unless Alice also gives out her private key, which Alice would almost definitely refuse to do if her public/private key pair is used in any other business.

To allow group access to encrypted columns, we can generalize the public key based approach of Section 5.2.

First, when Alice creates the table `Customer`, she could explicitly specify who will be allowed to access the unencrypted credit card numbers:

```
CREATE TABLE Customer
   (userid          INT PRIMARY KEY
   lastname         varchar(25),
   firstname        varchar(25),
   ccnum            char(16) ENCRYPTION
                    USER user_list
                    UPDATE no
   );
```

where `user_list` lists the names of all user that are allowed to access the (unencrypted) credit card numbers. For simplicity, we assume the user list contains only Bob.

The security catalog SEC_ENCRYPTION is extended to record this extra information, that is, a record in SEC_ENCRYPTION contains the following information, among others:

$$\begin{aligned}(&\texttt{userid} = \texttt{Alice}, \texttt{table} = \texttt{Customer},\\ &\texttt{column} = \texttt{ccnum}, \texttt{enc\_flag} = \texttt{no},\\ &\texttt{updateby} = \texttt{Alice}, \texttt{user\_list} = \{\texttt{Bob}\},\\ &\texttt{user\_flag} = \texttt{no}).\end{aligned} \tag{1}$$

When Alice creates John Smith's record in table `Customer`, the encryption is done the same way as before, except that the working key K will be encrypted twice, once using Alice's public key PK_A and once using Bob's public key PK_B, and the following two entries will be stored in the directory server:

$$(\texttt{Alice}, '\texttt{Customer}', '\texttt{ccnum}', E(\texttt{PK}_\texttt{A}, K))$$
$$(\texttt{Bob}, '\texttt{Customer}', '\texttt{ccnum}', E(\texttt{PK}_\texttt{B}, K))$$

Alice can access the credit card number as before. For Bob, he can supply his own (matching) private key when issuing the following **SELECT** statement:

SELECT `ccnum` **FROM** `Customer`
WHERE `userid` = 100 **PRIVATE KEY** $\texttt{SK}_\texttt{B}$;

The database server will first check SEC_ENCRYPTION to see if Bob is the owner of table `Customer` or listed as a sharing user for (`Customer`, `ccnum`). If so, the database server will fetch $E(\texttt{PK}_\texttt{B}, K)$ from the directory server, decrypt it using the user supplied $\texttt{SK}_\texttt{B}$, and use the recovered working key K to decrypt the encrypted credit card number.

Alice can also update the sharing user list later (using **ALTER TABLE** statement) because she has specified `user_flag` = `no`. When she issues the **ALTER TABLE** statement, she needs to supply her private key. Similarly, she can grant Bob access to the encrypted column. All the engine needs to do is encrypting the working data encryption key K with the new user's public key.

The new catalog SEC_ENCRYPTION and our enhanced user management mechanism together improve the system security. For example, suppose *user_flag* =no and Alice (the owner of table `Customer`) is allowed to change the SEC_ENCRYPTION record to add new sharing users for column `ccnum`. We first consider the case with the old user management mechanism. As explained in Section 2.1, suppose a DBA resets Alice's password so that she/he can login as Alice and change the above record (1) to the following

$$\begin{aligned}(&\texttt{userid} = \texttt{Alice}, \texttt{table} = \texttt{Customer},\\ &\texttt{column} = \texttt{ccnum}, \texttt{enc\_flag} = \texttt{no},\\ &\texttt{updateby} = \texttt{Alice}, \texttt{user\_list} = \{\texttt{Bob}, \texttt{Clark}\},\\ &\texttt{user\_flag} = \texttt{no})\end{aligned} \tag{2}$$

where "Clark" is the name of another user whose private key is known to the DBA. After the update, the DBA logs off and restores Alice's old password. Later, when Alice creates a record for a new customer Frank in table `Customer`, Frank's credit card number will be encrypted with a working key K that is in turn encrypted by the public keys of Alice, Bob, and Clark separately, which means Clark (and thus the DBA) will be able to obtain the working key K to decrypt the new customer's credit card number.

However, if our enhanced user management mechanism is in place, a DBA will not be able to obtain information she/he is not supposed to obtain. In the above example, after reseting Alice's password, the DBA will not be able to restore Alice's original password. The next time when Alice tries to login, she will fail and she immediately knows that someone has changed her password without her authorization. The detection of unauthorized password change serves as the basis for starting any further security management process.

In the above discussion we assume that users' public key certificates, encrypted private keys and encrypted working keys are stored in a directory service. A better way is to store them locally in the security dictionary. We can use a security catalog SEC_CERT to store all certificates, use a security catalog SEC_WORKINGKEY to store all encrypted working keys, and use a security catalog SEC_PRIVATEKEY to store all encrypted private keys if necessary. The advantage of doing this is two folds. First, a user's public key certificate stored in an independent PKI directory service might change dynamically, but his public key for database encryption is relatively static. Therefore, there is a need for synchronization which might be difficult. Besides, once a public key PK_A is used to encrypt a working key, that working key must be decrypted using a private key that must exactly match PK_A, not any newly updated public key. Thus it is a good practice for the database server to store those public key certificates used for database encryption locally. Second, local storage is good for efficiency reason. Frequent access to an outside directory service would slow down the whole system significantly.

6 Conclusions

In this paper we investigate the role cryptography can play in database security. We analyze the security features of current RDBMSs and point out their weaknesses. We introduce the key concept of security dictionary and proposed several secure user management and encryption methods based upon the new concept.

Another important research problem is how to do database operation against encrypted columns, such as indexing and join. We will discuss this in a forthcoming paper.

References

[1] N. R. Adam and J. C. Wortmann. Security-control methods for statistical databases: a comparative study. *ACM Computing Surveys*, 21(4):515–556, 1989.

[2] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, 2000.

[3] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1995.

[4] J. Cook, R. Harbus, and T. Shirai. *DB2 Universal Database v6.1, 3rd Edition*. Prentice Hall, 2000.

[5] D. Coppersmith, H. Krawczyz, and Y. Mansour. The shrinking generator. In *Lecture Notes in Computer Science 773*, pages 22–39, 1994.

[6] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, Inc., 1982.

[7] T. Dierks and C. Allen. *The TLS Protocol - Version 1.0, Internet-Draft*. November 1997.

[8] A. Freier, P. Karlton, and P. Kocher. *The SSL Protocol Version 3.0, Internet-Draft*. November 1996.

[9] S. Garfinkel and G. Spafford. *Web Security & Commerce*. O'Reilly & Associates, Inc., 1997.

[10] S. B. Guthery and T. M. Jurgensen. *Smart Card Developer's Kit*. Macmillan Technical Publishing, 1998.

[11] Informix. *Informix-Online Dynamic Server Administrator's Guide, Version 7.1*. INFORMIX Software, Inc., 1994.

[12] G. Koch and K. Loney. *Oracle8: The Complete Reference*. Osborne/McGraw-Hill, 1997.

[13] J. C. Lagarias. Pseudo-random number generators in cryptography and number theory. In *Cryptology and Computational Number Theory*, pages 115–143. American Mathematical Society, 1990.

[14] T. F. Lunt. A survey of intrusion detection techniques. *Computer & Security*, 12(4), 1993.

[15] National Bureau of Standards FIPS Publication 180. *Secure Hash Standard*, 1993.

[16] National Bureau of Standards FIPS Publication 46. *Data Encryption Standard (DES)*, 1977.

[17] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, 1994.

[18] San Jose Mercury News. Web site hacked; cards being canceled, Jan. 20, 2000.

[19] Oracle Technical White Paper. *Database Security in Oracle8i*, November 1999.

[20] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons Ltd, 1997.

[21] R. Rivest. *The MD5 Message-Digest Algorithm, RFC1321 (I)*. April 1992.

[22] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signature and public key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[23] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.

[24] A. Shamir. How to share a secret. *Communication of the ACM*, 22(11):612–613, 1979.

[25] D. R. Stinson. *Cryptography; Theory and Practice*. CRC Press, Inc., 1995.