Securing Data at Rest: Database Encryption Solution using Empress Embedded Database

By: Srdjan Holovac

June 2012 *White Paper*

Contents

Introduction 2
Importance of security 2
Protecting data at rest 2
Terminology 2
Different Concepts of Empress Database Encryption Solution 4
Concept A: Implementing Hardware-Based Encryption Solution with Empress Embedded Database 5
Concept B: Implementing Software-Based Encryption Solution with Empress Embedded Database 6
How It Works 7
Unit of Encryption 8
Encrypting Binary Large Objects/ Character Large Objects 10
Encrypting Indexed Data 10
Type of Encryption 10
Key Management 10
Cipher Key Verification 11
Key Rotation/ Key Change Procedure 12
Performance and Size Considerations 12
Performance Considerations 12
Size Considerations 12
Embedded System Application Benchmark 14
Benchmark Database Size Results 15
Benchmark Performance Results 16
Secure Data at Rest in Embedded Systems 16
Use Case: Parking Ticket System 18
In Summary 22
Literature 22

Introduction

Importance of security

Enforcing data security is top priority for both governments and businesses worldwide. Recent legislation in many countries has set new standards for protecting customer information. There are standards for the security of medical records and standards for the financial industry regarding privacy and security of customers' personal financial information.

How can this confidential data be protected?

New technology advances, including encryption, offer significant security capability for confidential data protection.

Protecting data at rest

This paper focuses on security solutions for protecting data at rest, specifically protection of data that resides in databases and stored in a persistent storage device such as disk.

Even in-memory databases need to backup data and this data could end up in a persistent storage device in plaintext.

Even many embedded devices that contain an embedded database hold sensitive data that must be protected.

Encryption, the process of disguising data in such a way to hide its substance, is a very effective way to achieve security for data at rest.

Implementation of database encryption raises several important points that must be taken into consideration such as: Should the encryption be performed inside the database engine, in the application where the data is generated or in a hardware device? Should encryption keys be kept inside the database or somewhere else where it may be more secure? Should the granularity of data encryption be on a database, a table or a column? What are the performance and size tradeoffs of different approaches?

This paper describes the implementation of the encryption solution using Empress Embedded Database, summarizes the benefits of this solution for users, and discusses the issues mentioned above.

Terminology

The definitions of the basic terms are simplified for the usage in this paper. The consulted terminology sources were: **[PKCS], [FIPS]** and **[BRUCE**] (see Bibliography).

Encryption: The process of disguising data in such a way to hide its substance.

Cipher (Cryptographic Algorithm): The mathematical function used for encryption and decryption.

Cipher Key (Cryptographic Key, Encryption, Key): A parameter used in conjunction with a cipher that determines:

- the transformation of plaintext data into ciphertext data,
- the transformation of ciphertext data into plaintext data.

Plaintext (Cleartext): A block of data that has not been encrypted.

Ciphertext: A block of data that has been encrypted.

Decryption: The process of transforming **ciphertext** back into **plaintext**.

Padding: A string, typically added when the plaintext block is short. For example, if the block length is 4 bytes and the cipher requires 16 bytes, then 12 bytes of padding must be added. The padding string may contain zeros, alternating zeros and ones, or some other pattern.

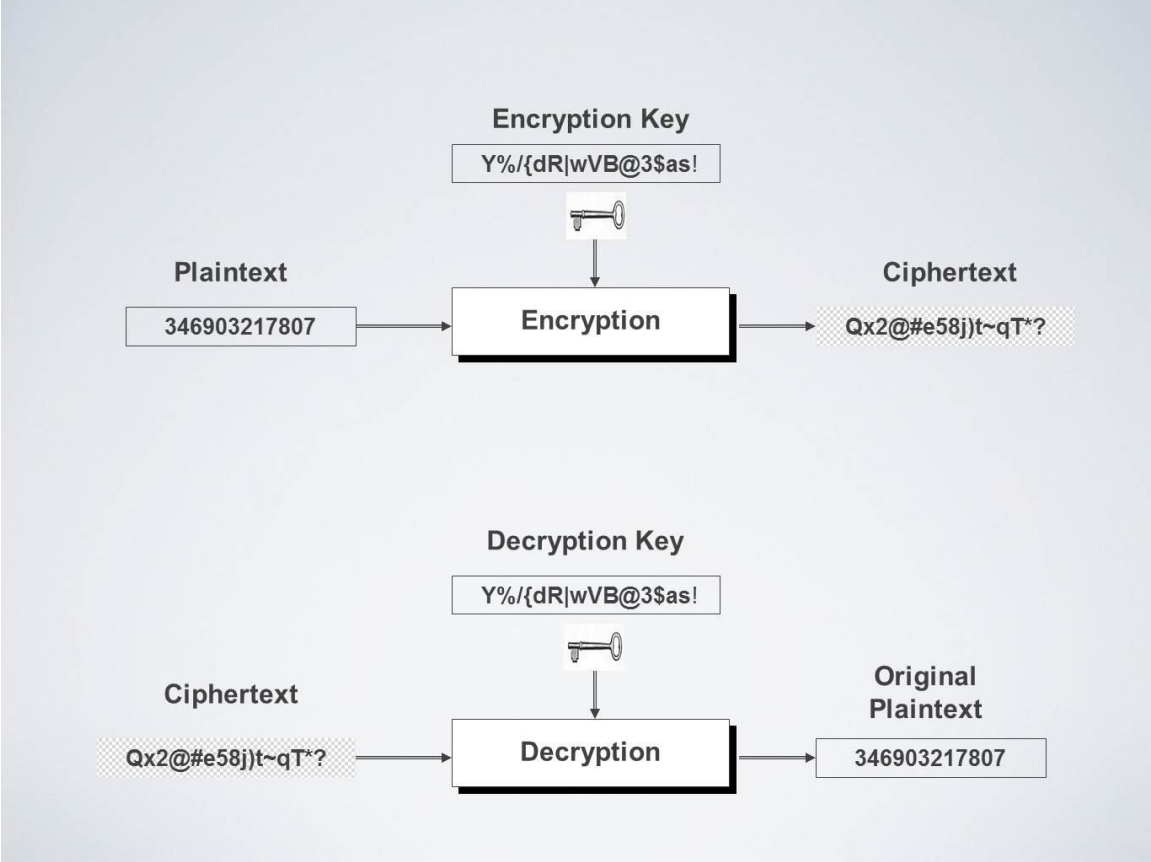


Figure 1. Encryption and Decryption

Figure 1 describes the process of encrypting and decrypting data. There are two general types of key-based algorithms: symmetric and public-key. In the case of most symmetric ciphers, the encryption and decryption keys are the same (as shown in **Figure 1**). Empress utilizes symmetric ciphers for its encryption solution.

Different Concepts of Empress Database Encryption Solution

The Empress database encryption solution is based on performing the encryption/decryption process either:

A) by a hardware-based solution using a hardware security device/appliance (**Figure 2**); For example, Empress Embedded Database can be configured with DataSecure appliances, dedicated hardware systems provided by Ingrian Networks **[NAE]**);

or

B) by a software-based solution using a Security Library that contains an encryption algorithm (**Figure 3**); For example, Empress Embedded Database can be configured with OpenSSL cryptographic library,

Microsoft Cryptographic Providers, libgcrypt, a general purpose library that contains various cryptographic algorithms, etc.

Concept A: Implementing Hardware-Based Encryption Solution with Empress Embedded Database

Empress Embedded Database incorporates C API calls from the Public Key Cryptography Standard (PKCS #11) that interface with the Security Adapter inside its Database Engine so that the encryption/decryption process is hidden from users **[PKCS]**.

PKCS is a suite of specifications developed by RSA Security in conjunction with industry, academic, and government representatives. PKCS #11 is the specification for the cryptographic token interface standard, defining a technology-independent programming interface for cryptographic applications.

Using the PKCS #11 standardized approach Empress Embedded Database is effectively integrated with the Ingrian Security Adapter – NAE PKCS #11 Provider **[NAE]**. Ingrian NAE Provider initiates encrypt and decrypt operations inside the Ingrian Security Device/Appliance, such as DataSecure. Both cipher key and cipher are contained inside the Security Device. DataSecure offers support for leading, standards-based ciphers: AES, 3DES, RSA and others.

Figure 2 describes the overall concept and placement of the Security Adapter in different Empress application scenarios. Empress standalone applications and Empress utilities, such as empsql or empclean, are directly linked to the Security Adapter. In the client-server scenario, such as ODBC and JDBC applications in **Figure 2**, the Security Adapter is linked to the Empress Connectivity Server.

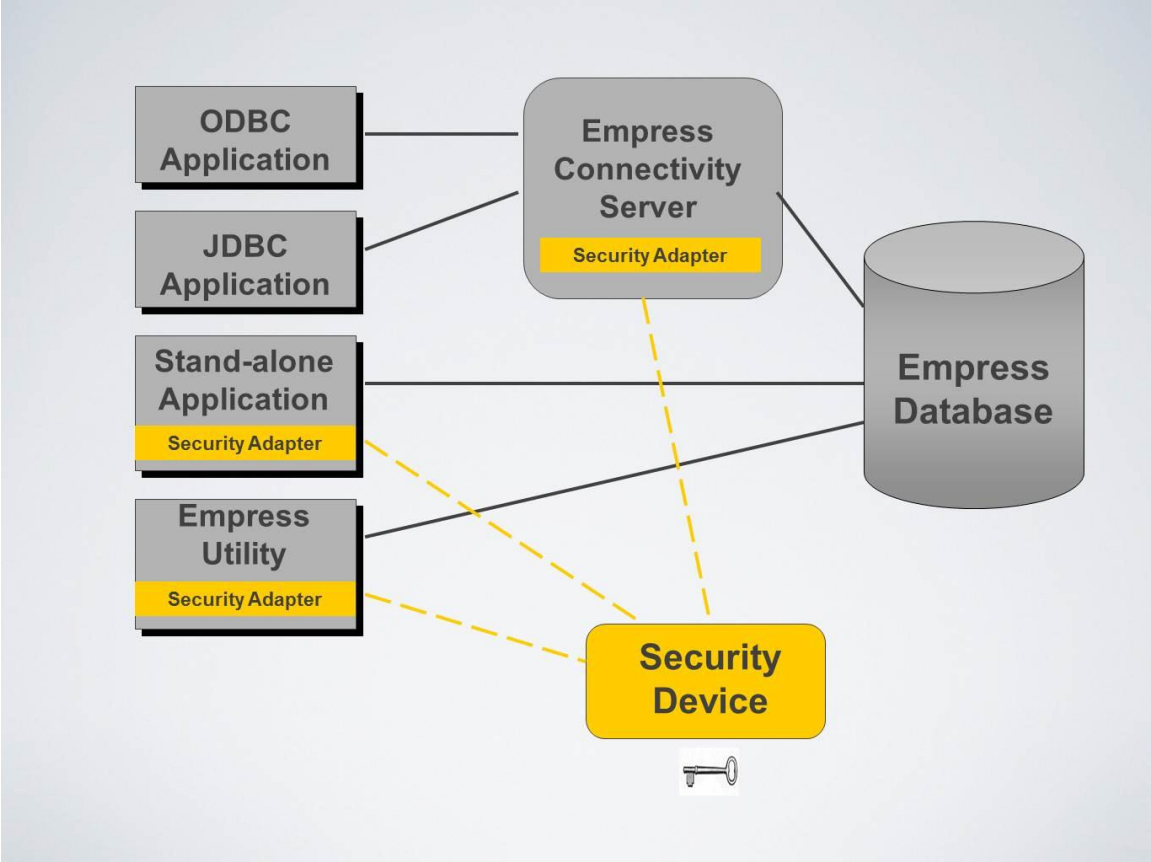


Figure 2. Concept A: Database Encryption Concept with a Security Device

Concept B: Implementing Software-Based Encryption Solution with Empress Embedded Database

Figure 3 describes the second concept of adding encryption capability to the Empress Embedded Database. This is a pure software solution that involves a Security Library, which contains a cipher. Empress Embedded Database is effectively integrated with several “crypto” libraries (such as OpenSSL crypto library), which performs data encryption and decryption. A cipher key is held either in a protected place in the file system, application/process environment or with a user.

Empress standalone applications and Empress utilities, such as empsql or empclean, are directly linked to the Security Library. In the client-server scenario, such as with ODBC and JDBC applications (**Figure 3)**, the Security Library is linked to the Empress Connectivity Server.

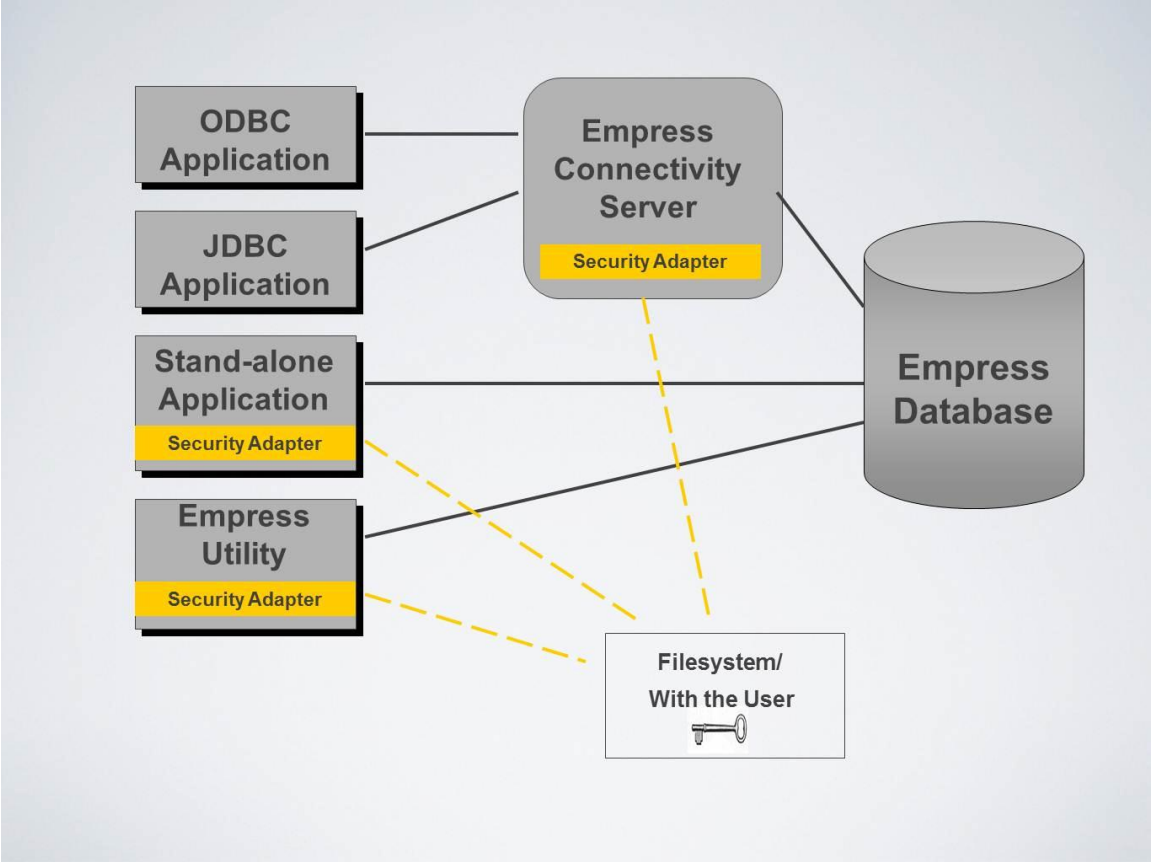


Figure 3. Concept B: Database Encryption Concept with a Security Library

How It Works

The following are the basic postulates regarding adding encryption to the Empress Embedded Database. They apply to both concepts described above.

The encryption is done on a column level. Users have the capability to define which columns are to be encrypted.

Let's assume a scenario where the database table **customer** has four columns **cust_no**, **name**, **ssn** and **address,** where customer number **cust_no** and social security number **ssn** have to be encrypted. To create such a table in an Empress database one would use the SQL CREATE TABLE command:

```
CREATE TABLE customer (
        cust_no LONGINTEGER NOT NULL ENCRYPTED,
        name CHAR(20),
        ssn CHAR(9) ENCRYPTED,
        address TEXT);
```

Since the column **cust_no** requires being searchable, an index has to be created too:

```
CREATE UNIQUE INDEX customer_index ON customer(cust_no);
```

Empress Embedded Database will encrypt data for the columns **cust_no** and **ssn** and decrypt data from those columns when the application needs it.

User applications that access table **customer** need NO changing. The same scenario works for all interfaces that Empress offers and also for Empress utilities.

Furthermore, users are given the ability to toggle between an encrypted and an unencrypted database by altering the database schema. Altering the schema changes the column encrypted properties. For example:

```
ALTER TABLE customer CHANGE ssn NOT ENCRYPTED;
```

Or to define the encryption on the column again:

```
ALTER TABLE customer CHANGE ssn ENCRYPTED;
```

One ALTER command can be issued in order to define encryption on multiple columns at once.

Users do NOT have to change the data type or the size of the encrypted column.

Unit of Encryption

How much data must be encrypted to provide security? Empress encryption provides flexibility to accommodate for the varying levels of encryption granularity. Users do not have to encrypt the whole database or the whole table.

Encryption is done on a column level in the database table.

Empress groups all encrypted columns in a table record.

Padding is done on the group of encrypted columns instead of padding on every encrypted column.

An Empress database is a directory in a file system. Users can have multiple databases on the same machine and switch databases as needed.

The unit of encryption varies depending on the type of file. Types of files that contain table column data in an Empress database are: main data (.rel) files

overflow data (.dtf) files, index data (.ix) files, transaction logs, recovery logs, backup files and temporary files.

File headers which contain no user data are not encrypted.

Figure 4 illustrates an example scenario that describes encrypted parts and non-encrypted parts of the files related to the table **customer** in the Empress database.

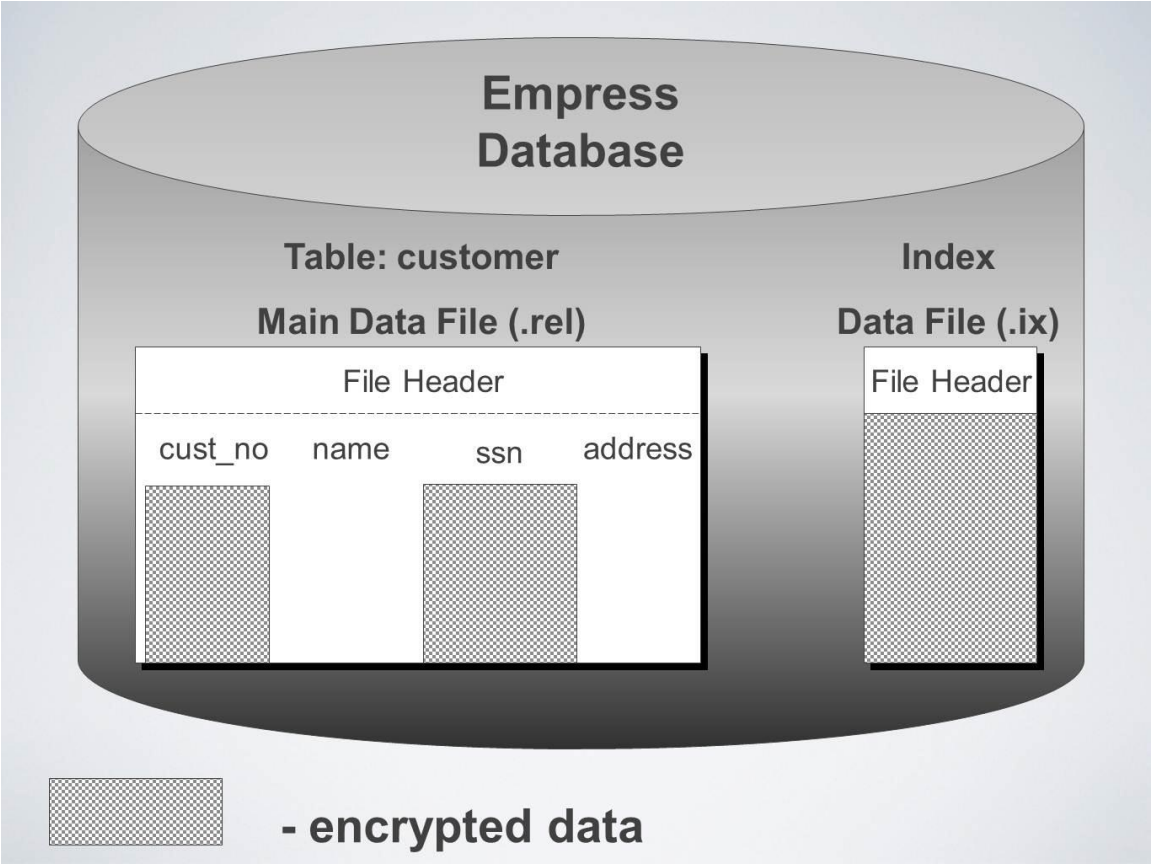


Figure 4. Encrypted and Unencrypted Parts of the Empress Database Table **customer**

There will be NO data from encrypted columns stored on the disk in plaintext.

Empress allows encryption on any column data type available in Empress Embedded Database (e.g. CHAR, TEXT, CLOB, BULK, BLOB, INTEGER, LONGINTEGER, REAL, DOUBLE PRECISION, DECIMAL, DATE, TIME, MICROSECONDTIMESTAMP, etc.).

Encrypting Binary Large Objects/ Character Large Objects

Some Empress Embedded Database data types require more work to encrypt or decrypt data. These data types are for storing Binary Large Objects (BLOBs) and for storing Character Large Objects (CLOBs). The amount of data stored in the columns of these data types may be very large. Empress does not impose limits on the size of those objects and does not force users to split data into chunks when dealing with large volumes of binary or text data.

Encrypting Indexed Data

Users have the ability to create indexes on the encrypted columns of any Empress data type that can be normally indexed. Empress has designed encryption in its indexes to make indexes usable for all kinds of searches, not only for equality searches. Hence, there is no difference in the usability of indexes for searches on encrypted or unencrypted columns.

Type of Encryption

For encrypting and decrypting data, symmetric cipher keys are used.

Various public domain ciphers, including AES are standard with Empress and are user-selectable.

Empress can embed user-defined ciphers on request.

The recommended cryptographic algorithm is Advanced Encryption Standard (**AES**). AES uses one of the 3 key lengths: 128, 192 and 256. The larger the key length the more computation it requires, and the greater security it provides.

All data in encrypted columns in a database is encrypted with the same cipher.

All the encrypted columns in a database are encrypted with one cipher key

Encryption is specified for a database when it is created. The cipher is selected at the time of creation and the selection is stored in the database. The cipher and the key should not be changed thereafter except by using the special Empress Key Rotation (Key Change) procedure.

Key Management

The Empress Embedded Database engine needs to obtain a cipher key for a given database. One of the essential key management questions is: Where should a cipher key reside? Some options to store a cipher key are:

1. in a database
2. outside of a database in persistent storage (e.g. access-protected files)
3. in memory
4. prompt for it each time encryption is needed
5. in a hardware security device

In **Figure 2 Concept A,** all cipher keys are stored in the Ingrian DataSecure Appliance. In order to encrypt/decrypt data in the Empress database, Empress Embedded Database must obtain **key info** (i.e. user name, password, key name) to pass to the Ingrian DataSecure Appliance in order to authenticate itself and gain access to the requested cipher key.

In **Figure 3 Concept B,** Empress Embedded Database must obtain a cipher key to pass to a Security Library.

The following options show how to address the requirements for both concepts.

1) The Empress variable MSCIPHERKEYINFO can be set to <database, key info> pairs (**Concept A**) or to <database, key> pairs (**Concept B**). Hence, key info or key resides in memory (i.e. environment of the Empress Application/Utility/Server).

2) The Empress variable MSCIPHERKEYINFOFILE can be set to the name of a file containing either key info (**Concept A**) or cipher key (**Concept B**). The file is called **credentials** file.

3) An additional option is to input key info or the key itself each time Empress application/utility is restarted.

Cipher Key Verification

A means of verifying that a cipher key is correct for a database is highly desirable so that meaningless data is not produced with an incorrect cipher key.

Concept A

During database creation, key info is encrypted with the cipher key it relates to and stored in the database. Thereafter, key info is encrypted with the cipher key it relates to and compared to the copy in the database to verify that it is correct for that database. If the key info is incorrect, a sleep period of several seconds is enforced before reporting the error.

Concept B

During a database creation, the cipher key is encrypted with itself and stored in the database. Thereafter, a given cipher key is encrypted with itself and compared to the copy in the database to verify that it is correct for that database. If the key is incorrect, a sleep period of several seconds is enforced before reporting the error.

Key Rotation/ Key Change Procedure

Empress provides a procedure for an off-line key rotation. The procedure is callable via a stand-alone script or as a sequence of calls inside the application and may be used for key rotation, either periodic or on an as needed basis.

The database needs to be off-line in order to re-encrypt all of the sensitive data within the database with a new cipher key. The following steps occur:

1. A second key is generated in the Security Device or in the **credentials** file.
2. A utility reads the encrypted data from the database, decrypts the information using the first key, encrypts with the second key, and writes the new encrypted data back to the database.
3. The key parameters are updated in the database to reflect the usage of the second key.

Performance and Size Considerations

Performance Considerations

What is an acceptable trade-off between data security and application performance?

The Empress encryption strategy includes multiple design and optimization consideration in order to ease the trade-off between data security and application performance.

Empress encryption initiates encrypt/decrypt API calls inside the Empress Engine. This preferable design offers a more efficient solution and requires much less overhead than other design alternatives such as via stored procedures/triggers.

Size Considerations

Empress groups columns that are to be encrypted at the record level to optimize the encryption process, thus decreasing the number of times it has to

encrypt/decrypt data. See **Figure 5** for sample Internal Empress Record Layout Optimization.

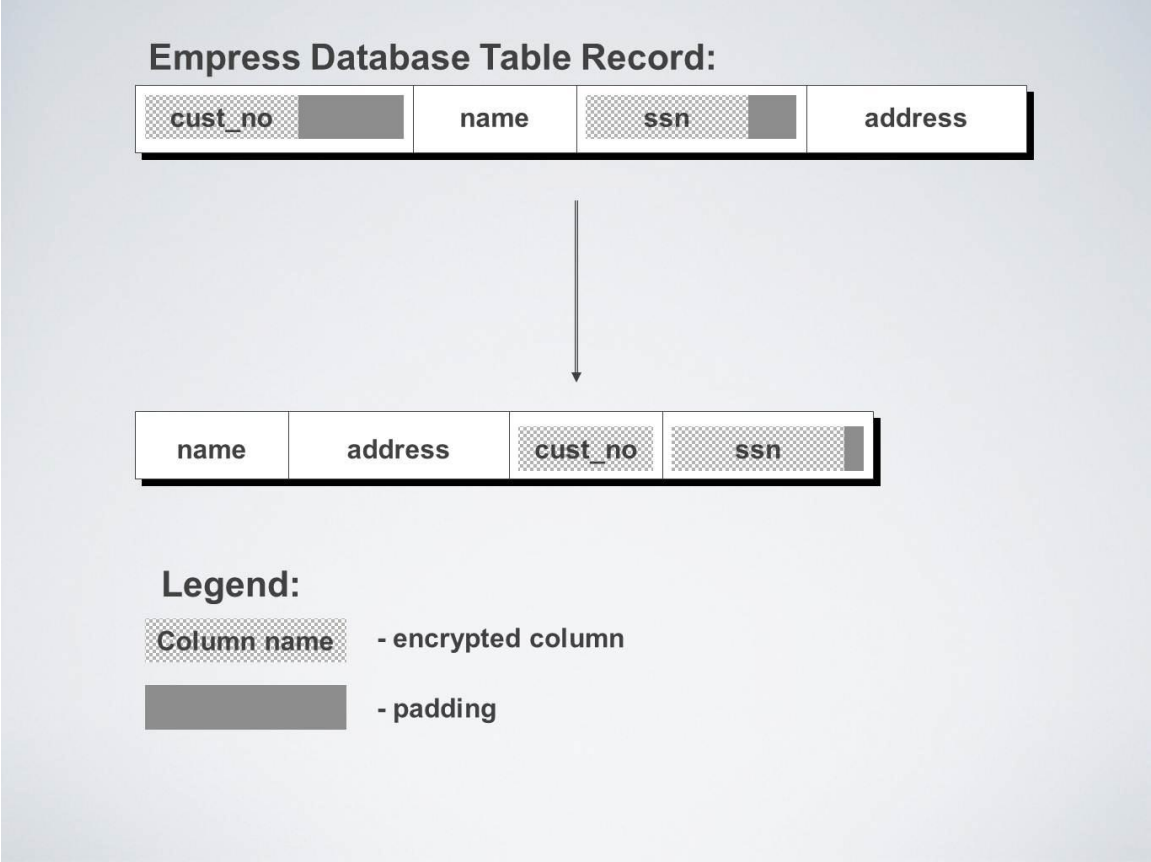


Figure 5. Internal Empress Record Layout Optimization

Empress encryption solution also saves space. Space is an issue with column encryption because, in general, encrypted columns are larger than unencrypted columns.

The overhead in an encrypted database size compared to the unencrypted database size is largely dependant on the cipher used. Let's assume that cipher requires padding to the nearest 16 bytes. Hence, a record of 508 bytes would have to be extended to 512 bytes, making the overhead insignificant.

This internal optimization with grouping of encrypted columns is not visible to users. Hence, the original order of table columns is preserved. If for example, a

user displays a table definition or retrieves data from a table, the user still get the original column order:

```
1* display customer all;
***  Table: customer  ***

  Attributes:
    cust_no                          longinteger Not Null Encrypted
    name                             character(20,1)
    ssn                              character(9,1) Encrypted
    address                          text(20,1024,1024,1)

  Creator:     srdjan
  Indices:     UNIQUE BTREE customer_index ON (cust_no)

. . .
2* select from customer;
    cust_no  name                 ssn        address
. . .
```

Embedded System Application Benchmark

Database size and performance were evaluated on a typical embedded system for common database access tasks in embedded system application.

The database **db** has two tables **jobinfo** and **hostinfo**:

```
  EMPRESS V10.20
  (c) Copyright Empress Software Inc. 1983, 2012
1* display db all into pager;
****  Database: db  ****

***  Table: hostinfo  ***

    jobid                            integer
    ipaddr                           character(16)
    macaddr                          character(16)
    protocol                         character(32)
    servicetype                      tinyint

***  Table: jobinfo  ***

    id                               integer
    pages                            integer
    quality_counts                   integer
    copy_counts                      integer
    login                            integer
    job_name                         character(128)
```

```
job_user                    character(32)
host                        character(32)
os                          character(32)
driver                      character(32)
printer                     character(32)
application                 character(32)
file                        character(32)
document                    character(128)
custom_string               character(32)
codeid                      character(4)
priority                    tinyint
submission                  character(128)
service_number              tinyint
group_name                  tinyint
authentification_id         character(32)
system_name                 character(32)
restriction_support         tinyint
restriction                 tinyint
status                      tinyint
job_date                    time
```

Another database **edb** was created with encrypted data. The database **edb** also has two tables: **jobinfo** with the attribute `authentification_id` encrypted and **hostinfo** with the attribute `macaddr` encrypted.

Benchmark Database Size Results

The non-encrypted database **db** size is 384 KB, while the encrypted database **edb** size is 392 KB. If we create an index on the attribute **macaddr** in the table **hostinfo**, the database sizes are shown in the Table 1.

	Non-encrypted Database db Size (KB)	Encrypted Database edb Size (KB)
Two tables	**384**	**392**
Two tables with an index	**416**	**424**

Table 1: Database Size (KB) for Non-encrypted and Encrypted Database with and without an index

Benchmark Performance Results

Performance was tested by performing standard SQL commands such as SELECT, INSERT, UPDATE and DELETE against non-encrypted database **db** versus encrypted database **edb**.

Results are given in the **Table 2**:

SQL Task	Non-encrypted Database Performance (seconds)	Encrypted Database Performance (seconds)
SELECT * FROM jobinfo WHERE pages >100;	**0.006**	**0.007**
SELECT * FROM hostinfo WHERE ipaddr > '19*';	**0.001**	**0.001**
INSERT INTO hostinfo VALUES (533,"191.234.33.45","00000000000008e5", "000000000001232abc0000000043",1);	**0.001**	**0.001**
UPDATE hostinfo SET ipaddr="191.234.33.46" WHERE jobid=533;	**0.001**	**0.001**
DELETE hostinfo WHERE jobid=533;	**0.001**	**0.001**

Table 2: Performance results for different SQL tasks performed against non-encrypted and encrypted database

Secure Data at Rest in Embedded Systems

Single Key per Database Approach

Key management in embedded systems imposes great challenges.
The current Empress security strategy is to use a single cipher key for each database. This simplifies key management and allows for performance optimizations when Empress issues encrypt/decrypt calls. This strategy is very beneficial for embedded systems since application developer doesn't need to cope with the complexities of dealing with multiple keys per single database.

Furthermore, the approach implemented in the embedded system affects the level of inconvenience imposed on end users. If the way sensitive data is stored is too hard to use, it will discourage users from using them. Some aspects of the user model examined in **[SARAH]** include:

- the number of times a person must enter an encryption key per session;
- the ease with which the method is invoked; and
- the number of encryption keys or passwords a person or a system must remember

With Empress single key approach per database, the ease-of-use is greatly enhanced.

Callable Admin API

Another useful Empress addition is Callable Admin API. Callable Admin API functions provide the support for the functionality given by Empress database administration utilities. Typical embedded system application cannot run Empress utilities as separate processes. Using Empress Callable Admin functions is the way for applications to run database administration tasks.

For example, for Android Embedded systems, Empress Callable Admin functions are implemented in Java. The following functions are supported via the DatabaseAdmin class:

createDB() - create a database
removeDB() - remove a database
exportDB() - export table(s) from a database
importDB() - import table(s) into a database
checkAndRepair() - check and repair database issues
encryptionSupported() - check whether encryption is supported
getCiphers() - get the list of all supported ciphers

Empress utilities can be invoked as callable administration functions when the application requires full control of database administration tasks.

Empress Callable Admin functions also support encryption. In the following text an example of exportDB function is given which includes encryption related parameters for Java and Android frameworks.

```
public static  void exportDB (
               String    dbName,
               String    cipherKey,
               String[]    tables,
               String    fileName,
               String    expCipher,
               String    expCipherKey)
throws SQLException
```

Parameters:

dbName: the physical path to the database
cipherKey: the cipher key hex string of the database if the database is encrypted, null otherwise, e.g. **"b1c23a3beefa4312"**.
tables: a list of tables to be exported. If it is null, all tables will be exported.

fileName: the physical path to the output file.
expCipher: the string that contains the cipher name, e.g. **"AES256".**
expCipherKey: the cipher key hex string that will be used to encrypt the output file, e.g. **"112244"**. If null, there will be no encryption on the output file**.**

Examples:
Export tables **'t1'** and **'t2'** from **/sdcard/db1** to **/sdcard/db1.exp**

```
DatabaseAdmin.exportDB("/mnt/sdcard/db1",
    new String[]{"t1","t2"}, "/sdcard/db1.exp");
```

Export all tables from **/sdcard/db1** to **/sdcard/db1.exp:**

```
DatabaseAdmin.exportDB("/mnt/sdcard/db1",
    "/sdcard/db1.exp");
```

Export all tables from **/sdcard/db1** to **/sdcard/db1.exp** and encrypt the output file with the cipher **AES256** and cipher key equals **"112244"**

```
DatabaseAdmin.exportDB("/mnt/sdcard/db1", null,
    null, "/sdcard/db1.exp", "AES256" ,"112244");
```

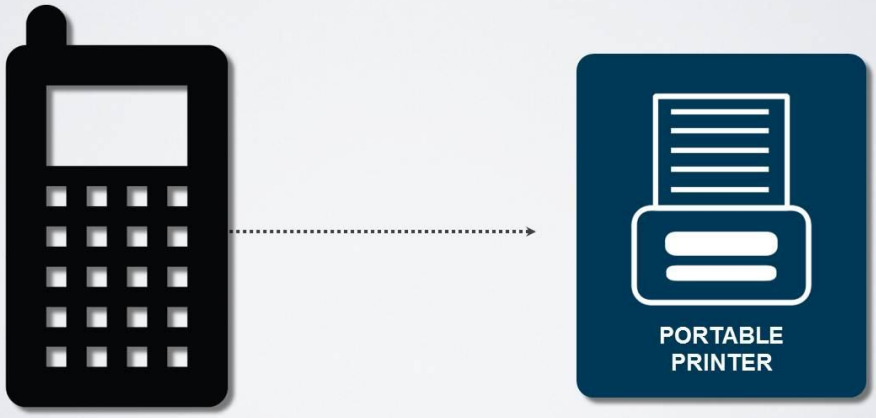
Use Case: Parking Ticket System

In the following text a use case for Empress Encryption for a parking ticket system is described.

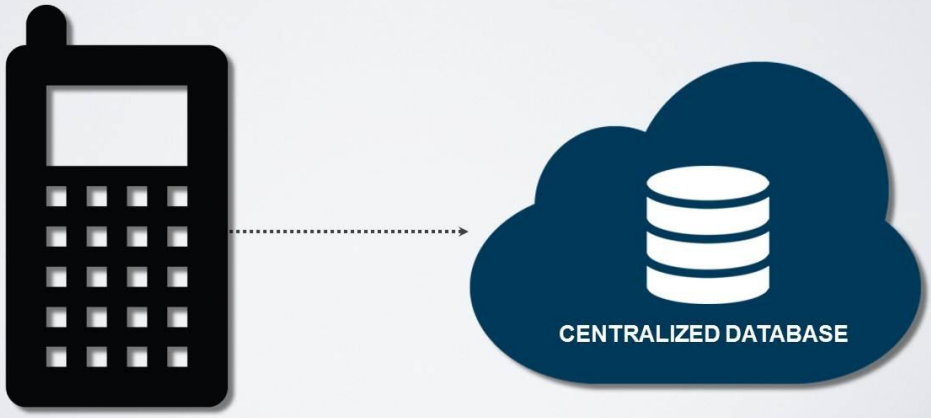
- Parking Authority of City ABC has a number of parking inspectors writing parking tickets
- Specialized Android mobile devices collect information needed for issuing a parking ticket

- Android device uses an Empress encrypted database to store vehicle image, GPS position, timestamp, license plate, etc.

- Android device is connected to a portable printer which prints the paper ticket



- After filling the parking ticket information the device is used to upload information to the centralized database

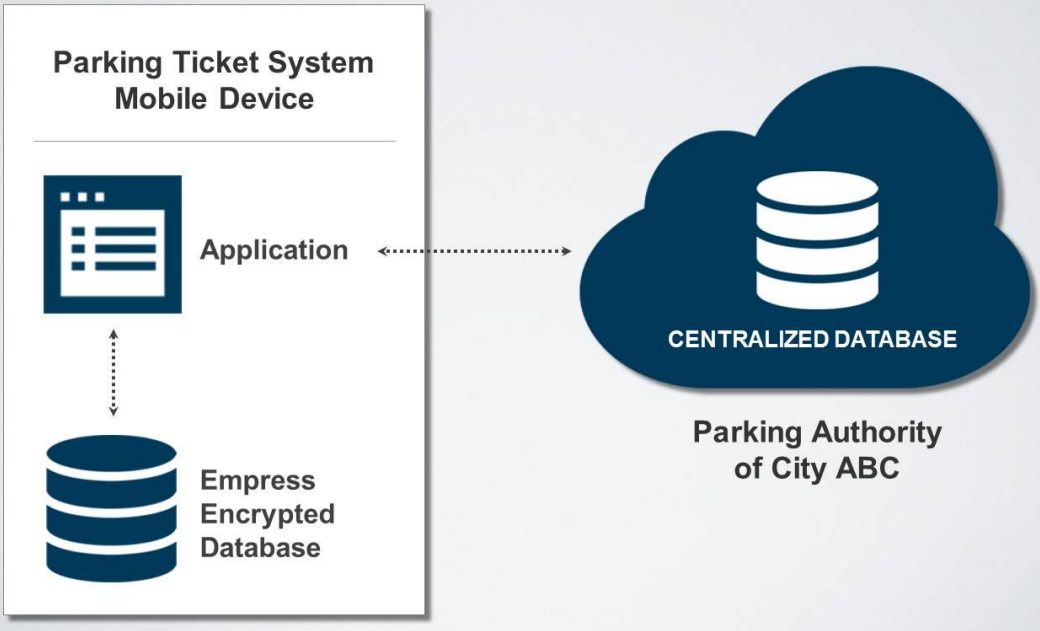


- Android device downloads a security key for the local Empress encrypted database before using the device

OR

- Parking inspector carries a card that contains the security key and swipes the card before using the device

Parking Ticket System
(Data Architecture)



In Summary

The main benefits for implementing the encryption solution using Empress Embedded Database are as follows:

- Secure all database data. A solution for protecting user data in a database including protection of all logs and backup files.

- Eliminate the potential that data at rest could be read by another party and minimize bad press, loss of customers, government intervention.

- Implement an efficient security solution. Small performance overhead when performing encryption/decryption using the pure software solution or using a hardware device/appliance. In addition, Empress with encryption, keeps the database size increase minimal.

- No change needed to application code for already written applications that use data in an unencrypted Empress database.

- No need to add external code in the database such as stored procedures, triggers or views to accommodate encryption.

- Painless solution for users who choose to convert their non-secure database solution to a secure one.

Literature

[BRUCE] Bruce Schneier: Applied Cryptography (Second Edition), John Wiley & Sons, 1996 ISBN 0-471-11709-9

[FIPS] FIPS PUB 140-2, Security Requirements for Cryptographic Modules, NIST, 2001.

[PKCS] PKCS #11 v2.11: Cryptographic Token Interface Standard, RSA Laboratories, Revision 1 — November 2001

[NAE] NAE Developer Guide for the PKCS #11 Provider, Ingrian Networks, 2005.

[SARAH] Sarah M. Diesburg, An-I Andy Wang: Modules, A Survey of Confidential Data Storage and Deletion Methods, Journal ACM Computing Surveys, Volume 43 Issue 1, 2010.