New Solution Implementation to Protect Encryption Keys Inside the Database Management System

Karim El bouchti*, Soumia Ziti, Fouzia Omary, Nassim Kharmoum

Faculty of Sciences, Mohammed V University in Rabat, Intelligent Processing Systems & Security (IPSS) Team, Morocco

ARTICLE INFO
Article history:
Received: 31 December, 2019
Accepted: 13 February, 2020
Online: 09 March, 2020

Keywords:
Encryption keys protection
Database encryption
Database security
keys protection in Databases

ABSTRACT
Due to the attacks' growth on sensitive databases by deploying advanced tools, beyond access control and authentication mechanisms, the database encryption remains a useful and effective way to ensure robust security of data stored within it. Any database encryption solution is based on a specific encryption model that determines how data is encrypted inside it. A relevant database encryption model must necessarily adopt a strong security policy of data encryption keys. It determines how these keys are generated, stored, and protected. In this work, we will implement an original solution that protects the encryption keys when encrypting data occurred at the Database Management System level. Our solution suggests protecting the keys by their encryption with other ones named Master Keys, which are generated according to the encryption granularity defined by the database encryption model. The proposed solution protects the keys of two database encryption models: at the level of the columns and the level of tables.

1. Introduction

Database (DB) level encryption is a way to encrypt and decrypt data within the Database Management System (DBMS) using keys held by the DB server [1]. In fact, this encryption model offers major advantages, in particular, those related to the security of the encryption keys against external attacks performed outside the information system [2, 3]. Though the probability of the administrator attack is not negligible, he owns large privileges on DB. Hence, he can attack the DB directly, via a collaboration with a malicious external attacker (disclosure of the administrator account for example), or with an internal attacker such as a legitimate user [4, 5, 6]. Obviously, the DB administrator has the ability to perform all these attacks without leaving any traces [7, 8].

Data encryption at the DBMS level is based fundamentally on its specific encryption model. The management of encryption keys within this model is a crucial point, it defines the ways how keys are generated, stored, and protected [9, 10]. Actually, keys values, how users access them, and where they are stored are the ultimate goal of any attacker. Therefore, it should be necessary to establish a protection policy for encryption keys to minimizing their exposure face of malicious attackers. Indeed, many solutions have been developed in order to resolve this problem. For instance, the keys protection solutions implemented at DBMSs such as Oracle, Ms SQL Server and My SQL are mainly based on the use of "Wallets", Hardware Security Module (HSM) and "Security server" [11-14]. In fact, each one of these solutions has advantages and major limits, as have already explained in more detail in our previous work [8, 15].

Several studies proposed several solutions to protect DB encryption keys either in DB encryption models or separate solutions of keys protection [16, 17]. The authors of [16] proposed a model to protect encryption keys based on a concept of distributed keys representation. The first key part stored in the DB, and the other part is obtained by converting the user password. Elovici et al. presented a special DB encryption model that permits to protect keys using "Wallet" mechanism [17], although Itimar et al. used asymmetric key encryption [18]. The authors of [19] suggested a solution called "Server-HSM" which merges an HSM and a security server into a module called "HW Security Module" that might be integrated into a DB server. This module manages user privileges and protects keys with their encryption. El bouchti et al. presented a full package of encryption keys protection models inside the DBMS. Their method allows encrypting keys with master keys generated according to encryption granularity adopted by the DBMS encryption model [15]. Sesay et al. in their proposed
DB encryption model suggested protecting keys by the use of a particular way. In fact, they generate the encryption keys from a unique master key (Km) generated and stored in a tamper-proof controller [20].

Notwithstanding several DB security studies for improving the concept of encryption keys protection within DBMS, more investigations are required to develop, improve, and provide more security and flexibility to keys protection. Then, as discussed in our previous work presented in [7, 10, 15], most of the proposed solutions have their advantages and disadvantages. However, to our knowledge, a trusted and simple solution concretized by a real implementation, and adapted to more than one DB encryption model has not yet been proposed.

In this context, the present work aims to implement two solutions proposed in our previous work [15] to protect DB encryption keys. Our solutions protect encryption keys when encryption granularity adopted by the DBMS is fixed at the column level or the entire table level. They consist of encrypting the DB encryption keys of tables and columns using Km, generated by deploying two models. We consider that the main original features of our solution are its reliability and its principle. It does not require any control or management of key protection by a DB or security administrator as well as it resists strongly against attacks performed by administrators.

The present work will be structured as follows: section two presents the proposed models of Km and the common objects of the implementation of each model. It also explains how our Km models work with DB encryption models that we will implement. Section three explains the implementation and discusses the provided results. Finally, our article ends with a conclusion.

2. Definition of the proposed solution

In this section, we will define two models (1 and 2) of the Km generation that protects, respectively, the columns and tables keys. We will also describe the role of the common objects that have used between the two models' implementation and how those models work in an encryption/decryption process.

2.1. Proposed models of Km

The generation of the Km follows the two different models below:

- The Km (C) key used to protect the encryption keys of the DB columns. It is generated by the DBMS according to the model defined below:

\[ Km (C) = H (Table\_name \|\| Column\_name) \quad (1) \]

- The Km (T) key utilized to protect the encryption keys of the DB tables. It is generated by the DBMS according to the model defined below:

\[ Km (T) = H (Table\_name \|\| Database\_name) \quad (2) \]

In order to concretize the functioning and the response of the models (1) and (2), we have designed and implemented two models of DB encryption (A) and (B) having, respectively, two encryption levels: column and table. The functioning of each Km generation model is associated with the execution of a DB encryption model, as shown in Table 1.

Table 1: The generation model of Km and the DB encryption model associated

<table>
<thead>
<tr>
<th>Km generation model</th>
<th>Associated DB encryption model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model (1): Km generation model for protecting encryption keys of columns</td>
<td>Model (A): DB encryption model at the column level</td>
</tr>
<tr>
<td>Model (2): Km generation model for protecting encryption keys of tables</td>
<td>Model (B): DB encryption model at table level</td>
</tr>
</tbody>
</table>

2.2. The common objects of each model implementation

The creation of elements below is common for the implementation of the models (1) and (2).

- MYTABLE_ENCRYPT_OBJET: This DB table contains records of: i) the names of the objects on which encryption has defined, ii) the used encrypt algorithms, and iii) the encryption of the encryption keys using Km. It has the following structure:

\[ MYTABLE\_ENCRYPT\_OBJET (K\_OBJECT\_NAME, K\_ENCRYPT\_ALGO, K\_KEY) \]

- TEST_MANAGEMENT: This DB table contains the records of: i) the names of the objects on which encryption has defined, ii) the used encrypt algorithms, iii) the object encryption keys, and iv) the Km of each object name. In fact, TEST_MANAGEMENT table is not implemented in the real working case of our solution. Nevertheless, its main role is to illustrate the results of the creation of both encryption keys and the Km. The table has the following structure:

\[ TEST\_MANAGEMENT (C1\_OBJECT, C2\_ALGO, C3\_KEY, C4\_MASTERKEY) \]

where:

- C1\_OBJECT, K\_OBJECT\_NAME: the object on which we have defined encryption;
- C2\_ALGO, K\_ENCRYPT\_ALGO: the algorithm used for encryption;
- C3\_KEY: the encryption key;
- C4\_MASTERKEY: the generated Km;
- K\_KEY: the encrypting result of encryption keys using Km.

The Md5 hash function: it is used to create the encryption keys and the Km while implementing all models.
The AES256 algorithm: it is the algorithm used to encrypt / decrypt the data.

Note: The term “Objet” signifies the column and table names.

2.3. Principle of the encryption/decryption process

An encryption/decryption process in the model (A) follows the defined steps below:

When a user sends a query to be executed, the DBMS generates the Km for each column on which encryption is defined. The DBMS decrypts, using the generated Km, a value from the K_KEY column belonging to MYTABLE_ENCRYPT_OBJET table and which corresponds to the column desired to be encrypted or decrypted. This process generates the real encryption key of column that will encrypt (in the case of an inserting or updating query) or decrypt (in the case of a consulting query) the data.

The encryption/decryption process in the model (B) follows similar operations. In this case, the DBMS generates a single Km in response to the user’s query since the encryption is defined on the entire table.

2.4. Case study: the dosimetric monitoring of agents

A real case study has chosen to illustrate the models' implementation results.

Let have a database named "ORCL10G" of a nuclear power plant intended to manage the dosimetric monitoring of agents working under ionizing radiation. The "agent" table stores the accumulation of the different types of doses received by each agent during his period of work within the plant. We assume that all the data in the "agent" table are sensitive since the dose values are considered in the nuclear field as medical secret. The table "agent" has the following structure:

agent (idf_agent, name_agt, Dose_interne_agt, Dose_superficielle_agt, Dose_profonde_agt, Catégorie_agt)

3. Implementation of the proposed models

In this section, we will present the implementation of the models (1) and (2) generating Km as well as the corresponding DB encryption models (A) and (B).

3.1. Implementation of the model (1)

In order to create the "agent" table and defining encryption on all its sensitive columns, we have used the following SQL syntax:

```
Create table agent (idf_agent varchar2(100) encrypt using AES256, name_agt varchar2(100) encrypt using AES256, Dose_interne_agt varchar2(100) encrypt using AES256, Dose_superficielle_agt varchar2(100) encrypt using AES256, Dose_profonde_agt varchar2(100) encrypt using AES256, Catégorie_agt varchar2(100) encrypt using AES256);
```

The "Algo1" algorithm supports the execution task of this statement; it creates the "agent" table and defines encryption on its columns using the AES256 algorithm. In fact, the data column will be encrypted/decrypted with six keys Kc which will be protected by six masters key Km (C). In addition, the "Algo1" generates and stores the encryption keys of the columns (Kc) and the master keys (Km (C)) within MYTABLE_ENCRYPT_OBJET and TEST_MANAGEMENT according to the models defined below:

```
/* The model of the encryption key used in the model (A)*/
Kc = H (Column_name)

/* The Km generation model of a column */
Km (C) = H (Table_name || Column_name)
```

Algorithm1: Process managed by Algo1.

<table>
<thead>
<tr>
<th>Algo1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: Sensitive_column_query</td>
</tr>
<tr>
<td>Output: Created _sensitive_column_query</td>
</tr>
</tbody>
</table>

```
Begin
Loop
Decompose (Sensitive_column_query);
Sensitive_column_name := Extract (Sensitive_column_query);
Kc := Kc_Generator (Sensitive_column_name);
Km (C) := Km_Generator (Sensitive_column_name, Table_name);
End loop;
Insert into TEST_MANAGEMENT values (Sensitive_column_name, Used_algo, Km, Km (C));
Insert into MYTABLE_ENCRYPT_OBJET values (Sensitive_column_name, Used_algo, Encrypt_AES256 (Km (C), Km (C)));
Execute (Sensitive_column_query);
End;
```

The execution of the "Algo1" algorithm generates the following records:

**Table 2: Records created in the TEST_MANAGEMENT table in the model (1).**

<table>
<thead>
<tr>
<th>C1_OBJECT</th>
<th>2_ALGO</th>
<th>C3_KEY</th>
<th>C4_MASTERKEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>idf_agent</td>
<td>AES256</td>
<td>46D18AC7BD06518B5A33C650CA760D9C</td>
<td>36EE051A4ACDEC7D1C07D16FDCC09CBF</td>
</tr>
<tr>
<td>name_agt</td>
<td>AES256</td>
<td>4BF0A8B1EBB8CA12C2912ED25E4D4BDC5</td>
<td>F842E273F1957702DE160662923C0EA</td>
</tr>
<tr>
<td>Dose_interne_agt</td>
<td>AES256</td>
<td>C8FE8472457B1A9436E90E6D022178F</td>
<td>452C55A5353CF7056C1293FE8D4FC</td>
</tr>
<tr>
<td>Dose_superficielle_agt</td>
<td>AES256</td>
<td>1F8D289F7BE7BA2565D7FDD4B6A91179</td>
<td>44BB0C31437DB46B4125733C7C458</td>
</tr>
<tr>
<td>Dose_profonde_agt</td>
<td>AES256</td>
<td>F67D11D0A22D37E3AC03D69C43A8994</td>
<td>8D7144C964377DB9F0E539B3F230133</td>
</tr>
<tr>
<td>Catégorie_agt</td>
<td>AES256</td>
<td>82E1CDA2CFA7B1EF36B90523E337682</td>
<td>FE30063A2D96A6DB6597C7081D3BD5F</td>
</tr>
</tbody>
</table>
Table 3: Records created in the MYTABLE_ENCRYPT_OBJET table in the model (1).

<table>
<thead>
<tr>
<th>K_OBJECT_NAME</th>
<th>K_ENCRYPT_ALGO</th>
<th>K_KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>idf_agent</td>
<td>AES256</td>
<td>12BB4076B53A907324D42AFB9B0501006A93E1F347EC05536A7115E4554 CD8D06662D1EEEAC1CA8D71BC2A33EDF7810B</td>
</tr>
<tr>
<td>name_agt</td>
<td>AES256</td>
<td>90F4D05F7C5A11A5752E8A1354BE82D797BFE16051F78DD7018A60856 49A709515637DB88472F14509DB73FC76E6B0C</td>
</tr>
<tr>
<td>Dose_interne_agt</td>
<td>AES256</td>
<td>6B40D18BF9F9C712F4F034E8A11AF73FC8B422CCC218AF595FD0E6B4 E4AE4B83E8D5D0CC99E85743FCD9690A577D05D0</td>
</tr>
<tr>
<td>Dose_superficielle_agt</td>
<td>AES256</td>
<td>5F03CF21D034BFFBA1FEA4BD5CEA43A15A32E89F4FDE591AF842995B BD30FF3CC25849363BCF54B8FCC2E8FD7FF7A7</td>
</tr>
<tr>
<td>Dose_profonde_agt</td>
<td>AES256</td>
<td>3A468DD545A1283543199A45202E4CE3AA4D006CE4453BFC380377423 C243C3FA85A1BF68F31F6E011C062C5E19D2AE</td>
</tr>
<tr>
<td>Catégorie_agt</td>
<td>AES256</td>
<td>34FEFD5520D63C4A99281FC10A8189D5B27EDC303E9AB760B35C1A2 28C1F7B0EFDAE2B6B7E3F507297D351E64FFE9</td>
</tr>
</tbody>
</table>

Table 4: The "agent" table before encryption.

<table>
<thead>
<tr>
<th>idf_agent</th>
<th>name_agt</th>
<th>Dose_interne_agt</th>
<th>Dose_superficielle_agt</th>
<th>Dose_profonde_agt</th>
<th>Catégorie_agt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Azzaoui</td>
<td>10</td>
<td>15</td>
<td>25</td>
<td>A</td>
</tr>
<tr>
<td>1001</td>
<td>Rachidi</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>1002</td>
<td>Kharmoum</td>
<td>16</td>
<td>05</td>
<td>14</td>
<td>A</td>
</tr>
<tr>
<td>1003</td>
<td>Sajid</td>
<td>14</td>
<td>66</td>
<td>65</td>
<td>B</td>
</tr>
</tbody>
</table>

Table 5: The "agent" table after encryption.

<table>
<thead>
<tr>
<th>idf_agent</th>
<th>name_agt</th>
<th>Dose_interne_agt</th>
<th>Dose_superficielle_agt</th>
<th>Dose_profonde_agt</th>
<th>Catégorie_agt</th>
</tr>
</thead>
<tbody>
<tr>
<td>5754957F70316A9C02 01002A8CBBFE412</td>
<td>745A2F1FF2BDAD322 691098999432787</td>
<td>B5063DD0A036503D0 2FI1B36DF1B12B1</td>
<td>5CD6B2C380377423 583F211E428B2A7</td>
<td>D94729051643806225 89A97E850A60</td>
<td>D1D8EDE1CF67BF 3A8B34261052FF335</td>
</tr>
<tr>
<td>A6771D65572FB3F6992 D7042A22B3879C</td>
<td>1655061F89A015B 5B98765133E9782</td>
<td>5C6BEA8480996813 19175A05C41B2B1</td>
<td>BD706EE5E149EBDC E691475E94F8B1</td>
<td>D1D8EDE1CF67BF 12FD1AD54922682</td>
<td>D1D8EDE1CF67BF 3A334261052FF335</td>
</tr>
<tr>
<td>ABEF3A3B5CFCD2B38969475416169DF5D</td>
<td>67BC01E3BF62DB2D7BF 568ACC3A582E3CB8</td>
<td>8CB892229BB1E395 1290107C524B12A</td>
<td>1258100A5303F86E8 DC55F42D0B34CB</td>
<td>398AFED4A54A0A5 5313CB381052FF335</td>
<td>5315B4717B530AA72</td>
</tr>
<tr>
<td>14DB0A7462823F494E AC7C8D12A09FC6</td>
<td>7A1FBCDDE02BC191 0055CF76279F21A</td>
<td>23BF97F16393E6CA8 483F4F9EB74A</td>
<td>293A7297BD5C760 450FFF14CEC693</td>
<td>A2B800E15CAFDF3AFA 9A22FB04C257B8</td>
<td>9AC1CD9CB5F4C8B</td>
</tr>
</tbody>
</table>

In order to test model (1), the "Algo2" algorithm represents the functioning of Model (A). It supports the data encryption inserted by a user. Tables 4 and 5 show the result of inserting four lines in the "agent" table before and after the encryption.

Algorithm 2: The DB encryption using the model (1) and (A).

```sql
CREATE OR REPLACE TRIGGER Insert_Model_A
BEFORE INSERT ON agent
FOR EACH ROW
DECLARE

Kc1 varchar2(100); Kc2 varchar2(100); Kc3 varchar2(100); Kc4 varchar2(100); Kc5 varchar2(100); Kc6 varchar2(100); BEGIN

/*Generating Km and seeking Kc for each column from MYTABLE_ENCRYPT_OBJET*/

Kc1 varchar2(100); Kc2 varchar2(100); Kc3 varchar2(100); Kc4 varchar2(100); Kc5 varchar2(100); Kc6 varchar2(100); END"
```
Kc1 := Search_Encrypt_Key('idf_agent');
Kc2 := Search_Encrypt_Key('name_agt');
Kc3 := Search_Encrypt_Key('Dose_interne_agt');
Kc4 := Search_Encrypt_Key('Dose_superficielle_agt');
Kc5 := Search_Encrypt_Key('Dose_profonde_agt');
Kc6 := Search_Encrypt_Key('Catégorie_agt');

/* Inserting in table "agent"*/

INSERT INTO agent VALUES (Encrypt_AES256(Kc1, :new.idf_agent), Encrypt_AES256(Kc2, :new.name_agt), Encrypt_AES256(Kc3, :new.Dose_interne_agt), Encrypt_AES256(Kc4, :new.Dose_superficielle_agt), Encrypt_AES256(Kc5, :new.Dose_profonde_agt), Encrypt_AES256(Kc6, :new.Catégorie_agt));
End;

3.2. Implementation of the model (2)

To implement model (2), we define encryption on the "agent" table level using the following SQL syntax:

Create table agent encrypt using AES256 (idf_agent varchar2(100), name_agt varchar2(100), Dose_interne_agt varchar2(100), Dose_superficielle_agt varchar2(100), Dose_profonde_agt varchar2(100), Catégorie_agt varchar2(100));

The "Algo3" algorithm supports the execution of this instruction. It creates the "agent" table and defines encryption on all its data using the algorithm AES256. In this case, the data are encrypted / decrypted with a single key KT, which will be protected by a single master key Km(T). The "Algo3" algorithm generates and stores KT and Km(T) also in MYTABLE_ENCRYPT_OBJET and TEST_MANAGEMENT according to the models defined below:

/* The model of the encryption key used in the model (B)*/
KT = H (Table_name)
/* The Km generation model of a table */
Km (T) = H (Table_name || Database_name)

Algorithm 3: Process managed by Algo3.

<table>
<thead>
<tr>
<th>Input: Sensitive_table_query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Created_sensitive_table_query</td>
</tr>
<tr>
<td>Begin</td>
</tr>
</tbody>
</table>

Decompose (Sensitive_table_query);
Sensitive_table_name := Extract (Sensitive_table_query);
KT := KT_Generator (Sensitive_table_name);
Km(T) := Km_Generator (Sensitive_table_name, Database_name);
Insert into TEST_MANAGEMENT values (Sensitive_table_name, Used_algo, KT, Km(T));
Insert into MYTABLE_ENCRYPT_OBJET values (Sensitive_table_name, Used_algo, Encrypt_AES256(Km(T), KT));
Execute (Sensitive_table_query);
End;

The execution of the "Algo3" algorithm generates the following records:

The "Algo4" algorithm represents the functioning of the model (B). It supports the data encryption inserted by a user. The tables 8 and 9 show the result of inserting four rows in the "agent" table before and after the encryption.

Algorithm 4: The DB encryption using model (2) and (B).

Algo4
CREATE OR REPLACE TRIGGER Insert_Model_2
BEFORE INSERT ON agent
FOR EACH ROW
DECLARE
KT varchar2(100);
BEGIN
/* Generating Km and seeking KT for each column from MYTABLE_ENCRYPT_OBJET*/
KT := Search_Encrypt_Key('agent');
/* Inserting in table "agent"*/

Table 6: Records created in the TEST_MANAGEMENT table in the model (2).

<table>
<thead>
<tr>
<th>C1_OBJECT</th>
<th>C2_ALGO</th>
<th>C3_KEY</th>
<th>C4_MASTERKEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent</td>
<td>AES256</td>
<td>B33AED8F3134996703DC39F9A7C95783</td>
<td>697C371A913425CF202D15F143D2DAF0</td>
</tr>
</tbody>
</table>

Table 7: Records created in the MYTABLE_ENCRYPT_OBJET table in the model (2).

<table>
<thead>
<tr>
<th>K_OBJECT_NAME</th>
<th>K_ENCRYPT_ALGO</th>
<th>K_KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent</td>
<td>AES256</td>
<td>73607902547466848431FC41910AB5770A6962367465EC1CB9526A864367916929D05016D02B96D9B55511854D3AB13</td>
</tr>
</tbody>
</table>
3.3. Results and discussion

This section introduces data discussion and analysis based on the findings obtained by implementing the two Km models. They are summarized as follows:

- The proposed solutions are more practical than the conventional ones, primarily the Wallet, HSM, and the security server, where their disadvantages have explained and revealed in [15]. Our solutions optimize perfectly additional costs to protect the keys either in terms of hardware acquisition (case of HSM and security server) or in human resources (the administrator of the security server).

- The Wallet solution used in Oracle TDE generates Km, which protects encryption keys, and stores it within the Wallet. Here, it is necessary to create the Wallet and its password as well as a secured location (such as backup systems) to store the password whenever the Km is newly created. This operation is mandatory before starting the process of data encryption/decryption inside DBMS [11]. It is worth to mention that the backup system is a critic component of the Wallet concept. In this vein, the protection concept of encryption keys based on the proposed models (1) and (2) is similar to the Wallet solution in terms of Km generation within the DBMS. However, with our concept, the Km creation does not require any Wallet creation to store Km or secured location to store the password.

- The proposed solution does not require any protection management of the encryption keys by a security administrator, DB administrator or another trusted collaborator, as discussed in [15]. Hence, none of them knew about the generation of Km or its location. The probability of attacking keys is almost impossible, even if the attacker arrives to consult table MYTABLE_ENCRYPT_OBJET stored in the DB dictionary.

- The proposed solution does not define the place where storing Km. Km generation is performed automatically while defining encryption on a sensitive object (column or table), precisely during the creation by the DB administrator. Obtaining a value of Km by attackers (administrators, internal or external attackers) is almost an impossible operation.

- It is important to notice that the new concept we have proposed and implemented enhances the security of encryption keys. In fact, compared to the Oracle TDE Column Encryption solution that uses a single master key to protect all the column keys, the model (1) generates several Km to protect each column key. The number of Km generated is equal to the number of sensitive columns. For example, if a DB contains 20 sensitive columns to encrypt, we need 20 keys to encrypt data and 20 Km to protect them.

- Our solution can work with any DB encryption models, obviously with those implementing encryption granularities at the level of columns and tables. It is well adapted to free license DBMSs.

- In the model (1) implementation, we focused on to protect 6 encryption keys of the following columns (idf_agent, name_agt,Dose_interne_agt,Dose_superficielle_agt,Dose_profonde_agt, Catégorie_agt). Each column key is protected by encrypting it with its own Km generated by the model (1). This protection was tested by the implementation of the model (A). Table 4 shows the encryption test results of the 6 columns of the “agent” table deploying the model (1). Likewise, in the model (2) implementation, one encryption key of the table "agent" has protected with a single Km generated using this model. Then, this protection was tested using the model (B). Table

---

### Table 8: The "agent" table before encryption.

<table>
<thead>
<tr>
<th>idf_agent</th>
<th>name_agt</th>
<th>Dose_interne_agt</th>
<th>Dose_superficielle_agt</th>
<th>Dose_profonde_agt</th>
<th>Catégorie_agt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Azzaoui</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>A</td>
</tr>
<tr>
<td>1001</td>
<td>Rachidi</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>1002</td>
<td>Kharmoum</td>
<td>16</td>
<td>05</td>
<td>14</td>
<td>A</td>
</tr>
<tr>
<td>1003</td>
<td>Sajid</td>
<td>14</td>
<td>66</td>
<td>65</td>
<td>B</td>
</tr>
</tbody>
</table>

### Table 9: The "agent" table after encryption.

<table>
<thead>
<tr>
<th>idf_agent</th>
<th>name_agt</th>
<th>Dose_interne_agt</th>
<th>Dose_superficielle_agt</th>
<th>Dose_profonde_agt</th>
<th>Catégorie_agt</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F843D72B4CF6AF3B</td>
<td>03377847D26334CAD6</td>
<td>7012CB55124F226FA2</td>
<td>E530E0D8133F14</td>
<td>7012CB55124F226FA2</td>
<td>91B5BD09938E4CE4</td>
</tr>
<tr>
<td>FEDD0FA73A46DB</td>
<td>65F815335C87F9</td>
<td>E530E0D8133F14</td>
<td>7012CB55124F226FA2</td>
<td>E530E0D8133F14</td>
<td>DF76529F6740B8A</td>
</tr>
<tr>
<td>2D80DCE15D394B765</td>
<td>0B71021170A841DAE</td>
<td>0C04B78B2A594ED2</td>
<td>D6CE2FCD9EE91FD</td>
<td>52516FBB145DB600</td>
<td>CF92FB85E0E373795</td>
</tr>
<tr>
<td>94AD5E18F3045BEE</td>
<td>0CB4641C08076D76</td>
<td>2D6CF94F7153C168</td>
<td>P9C4D57D66ECF1</td>
<td>CF92FB85E0E373795</td>
<td>DF76529F6740B8A</td>
</tr>
<tr>
<td>4CDDEE4F428E61343</td>
<td>DA45B069E894C7BE4</td>
<td>0F8913B9B06C772A68</td>
<td>8E9489B13A1124</td>
<td>5CD91E8240D88EAD</td>
<td>CC2CCB3C2994CE129</td>
</tr>
<tr>
<td>AD6C823E1860A4</td>
<td>C2B69E232E14B1A4</td>
<td>7072CB7673A84C1</td>
<td>59B9C9B66F478A5</td>
<td>CC2CCB3C2994CE129</td>
<td>91B5BD09938E4CE4</td>
</tr>
<tr>
<td>CA8028E0ADBDD105C</td>
<td>EE6099C500334279227</td>
<td>98B7E13A8752956</td>
<td>EC2F5721D81D9C9E6</td>
<td>EC2F5721D81D9C9E6</td>
<td>DF76529F6740B8A</td>
</tr>
<tr>
<td>E371803D3F575E</td>
<td>AFA5C27E199FB</td>
<td>59B9C9B66F478A5</td>
<td>2187B81CFCDF6B2B</td>
<td>4C6EA7B949ED52</td>
<td>72621DFB38C674D9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

www.astesj.com
8 represents the result obtained of encrypting table "agent" by deploying the model (2).

- The results presented in tables 2 and 3 shows the keys generation when the "agent" table is created by the administrator. In table 2, the columns C3_KEY and C4_MASTERKEY represents, respectively, the encryption key generated of each column and its associated Km. In table 3, the column K_KEY, belongs to MYTABLE_ENCRYPT_OBJET table, represents the encryption of each column key of the "agent" using its associate Km.
- In table 6, the columns C3_KEY and C4_MASTERKEY represents, respectively, the key generated to encrypt the entire table and its Km. The value of the column K_KEY in table 7 represents the encryption of the encryption key using its associate Km. Both tables show the keys generation when the "agent" table is created by the administrator.
- As described earlier, tables 4, 5, 8, and 9 show the result of inserting rows in the "agent" table before and after the encryption. The encryption results showed that our solution works perfectly, either when encrypting or decrypting data. Actually, the encryption process via model (A) or (B) requires the generation of Km through model (1) or (2), respectively. Each value of generated Km is used to extract, from K_KEY column, the real encryption /decryption key.
- The process of data decryption follows the same operations.

Finally, both proposed model works inside DBMS are summarized according to the algorithm flowchart described below:

Let’s consider a sensitive table A

![Algorithm flowchart specifying the both proposed models](https://www.astesj.com)

**Conclusion:**

Besides the conventional mechanisms deployed to secure sensitive DB (network protection, authentication, and access control), data encryption at DBMS level is a strong way that reinforces the defense in depth of the sensitive data. This process is strongly linked to the protection of the encryption keys on which they depend on two main factors: the location where the keys are stored and users who have access to them.

Our contribution in this article is to implement two solutions that secure encryption keys within the DBMS. These solutions are original and well adapted to any encryption model inside a DBMS. The solution’s purpose is to protect DB keys by their encryption using a master key generated when defining encryption on a table or column. In forthcoming works, we aim to develop a solution that covers the protection of encryption keys when encryption is done on Tablespaces.

**Conflict of Interest**

This manuscript has not been published and is not under consideration for publication elsewhere. We have no conflicts of interest to disclose.

**Acknowledgment**

I would like to express my appreciation to all my professors, whom they helped and guided me to realize this work.

**References**


