# A New Solution to Protect Encryption Keys when Encrypting Database at the Application Level

Karim El bouchti[1], Soumia Ziti[2], Fouzia Omary[3], Nassim Kharmoum[4]
Department of Computer Science, IPPS Team
Faculty of Sciences, Mohammed V University in Rabat
Morocco

***Abstract*—Encrypting databases at the application level (client level) is one of the most effective ways to secure data. This strategy of data security has the advantage of resisting attacks performed by the database administrators. Although the data and encryption keys will be necessarily stored in the clear on the client level, which implies a problem of trust viz-a-viz the client since it is not always a trusted site. The client can attack encryption keys at any time. In this work, we will propose an original solution that protects encryption keys against internal attacks when implementing database encryption at the application level. The principle of our solution is to transform the encryption keys defined in the application files into other keys considered as the real keys, for encryption and decryption of the database, by using the protection functions stored within the database server. Our proposed solution is considered as an effective way to secure keys, especially if the server is a trusted site. The solution implementation results displayed better protection of encryption keys and an efficient process of data encryption /decryption. In fact, any malicious attempt performed by the client to hold encryption keys from the application level cannot be succeeded since the real values of keys are not defined on it.**

***Keywords*—Database encryption; encryption key protection model; database encryption keys protection; data security**

## I. Introduction

Today, Database (DB) security is considered as one of the significant challenges in the computer world. Recently, it has been the subject of several debates and studies by experts and researchers in the data security field. The main purpose is to protect sensitive data against unavailability, leakage, and modification face to attackers' threats [9], [21], [22].

With the rapid development of technology, the attack scenarios on the DBs have become easy to realize. Currently, a penetration to a DB using SQL injection techniques becomes fast and quite simple thanks to specialized tools [10]. Different threats may come from various sources, some from trusted DB users, others from external ones, and some attacks are performed by the DB administrators [1], [2], [3]. Data theft is a dangerous attack. However, attacks compromising data integrity can generate heavier consequences; they are hardly detectable as the theft of data. For this reason, the implementation of three security levels is necessary: 1 / - Physical security, 2 / - Operating system security, 3 / - Database Management System (DBMS) security [15], [18].

The implementation of these security levels is useful, especially the access control mechanism implemented at the DBMS level. It is considered as the first defense line against unauthorized access [14], [23]. However, this mechanism is not sufficient, it protects only against attacks coming from outside the information system, and some limits can be generated as it has been explained in [11], [16]. Besides the access control mechanism, a security approach based on DB encryption can play an important role; it can be performed on three levels: DB level, hard disk level, and application level [7].

Encrypting DB at the application level consists of delegating encryption and decryption to applications that are connected to it. The DB server processes and manipulates only encrypted data, and the encryption keys are not implemented on the server. This solution provides strong protection of the encryption keys from threats performed by the administrator. However, the keys must necessarily be stored either on the applications or in a place where the applications are managed (application server for example). This approach is limited as the application user (or the application server administrator) are not always trusted sites, they can attack encryption keys at any time, and decrypt all sensitive data without leaving any traces [16].

It is mentioned in [8] that DBs confront several internal threats, especially those coming from legitimate users of the system. These threats can be realized in collaboration with a malicious administrator, another legitimate user, or with a malicious attacker outside the information system [1], [12]. Excessive privilege abuse, legitimate privilege abuse, and elevation of privilege are some models of internal attacks that compromise encryption keys. A legitimate DB user can exploit the privileges one's to attack encryption keys directly. The user can also exploit uncorrected DB vulnerabilities or DB configuration errors to access illegally to the DB encryption keys location. A mistrust application administrator may reveal the encryption keys to an external attacker in order to attack the DB outside the perimeter of the information system [8], [13]. Trusted users are considered a serious threat to the security of encryption keys when encryption is performed at the application level. Otherwise, the internal attacks are considered dangerous not only on DBs but on Big Data platforms like Hadoop. For instance, the authors of [24] argued that an effective attack launched from the compromised node could degrade the data processing performance of the cluster. Then, they exposed an effective schema that might mitigate the risk of this attack and keep the cluster running efficiently. Also, the

authors of [25] have developed a solution called ROVER, which is an efficient and verifiable Erasure Coding based Storage (ECS) for Big Data platforms. They showed that this solution implementation has good robustness and effectiveness against attacks from compromised nodes.

In order to protect encryption keys within applications, several solutions have been proposed by researchers. Among the most important works, we highlight the work proposed by Ding et al. [19]. They have proposed a new data encryption model implemented at the application level that ensures the confidentiality of sensitive data. It is based on a new method using keys chain to protect encryption keys. The authors of [20] have proposed a special concept called "Encryption as a service" to encrypt DB. Its main goal is to outsource the encryption system outside applications as an encryption service provider unit independently of applications users and the DB server. The protection of encryption keys in this concept is done using a Master key. Bouganim et al. have presented in [17] a solution named "Client HSM". It consists of integrating the module "HW Security Module" at the level of each DB user. This module holds and protects encryption keys to eliminate their exposure.

Despite the efforts made by researchers to improve the protection of encryption keys at the application level, they still need to be developed and improved. The solutions proposed in the literature are not sufficient as well as each solution has its limits, as we have mentioned in our previous works [3], [5], [8]. In addition, the solutions mostly presented are specific in the case when encryption is done at DBMS.

In this context, the present work aims to propose a new solution to protect encryption keys within applications against internal attacks when adopting encryption at the application level. Our solution is original and provides better protection for keys without any requirement of equipment or material. It is based on the outsourcing of the protection of encryption keys defined on the client side to a DB server. The principle is to transform user's queries holding user's encryption keys to new queries that remotely call functions stored on the DB server in order to convert these keys to real keys for encryption and decryption.

Our paper will be presented as follows: Section II consists of giving general overview of DB encryption at the application level. Section III describes our proposed solution and explains its implementation; we discuss in this section the test results provided by the implementation. Finally, the article ends with a conclusion in Section IV.

## II. Database Encryption Preliminaries

### A. Database Encryption at the Application Level

Encrypting DB at the application level (client side encryption) means that the process of encrypting/decrypting data is done locally on the application before transmitting data to the DB server. This principle is similar to an externalization of the DB in a cloud storage service [6], [16]. The keys in this concept are managed by the application on the client. It holds all the encryption keys, and there is no transmission of data or keys in the clear to the DB server [4] (see Fig. 1).

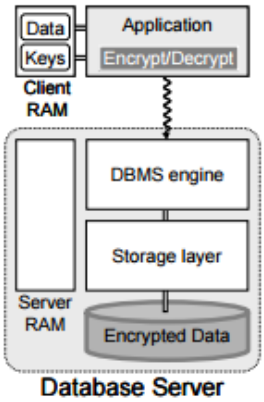


Fig. 1. Database Encryption at Application Level [17].

DB encryption at the application level has many advantages. It resists against attacks coming from administrators. However, it has several disadvantages such as, data and keys are necessarily stored in clear mode on the client, which makes a problematic trust face to the DB users, or even the applications administrator. Both of them cannot always be trusted sites [4], [16].

### B. Internal Attack on Encryption at the Application Level

As it has been highlighted by the authors of [4], [5], [8], [16], the DB legitimate operators (including operators that manage applications connected to DB) are considered the main threat compromising the encryption keys when DB is encrypted at the application level. The values of keys and their locations are the vulnerability of this concept. In order to clarify the concept of internal attack when encrypting DB at the application level, we have chosen the attack on DB in both the 2-tier and 3-tier architectures.

In 2-tier client/server architectures, we have 2 essential components [26] including: the client machine (first level) and the DB server (second level). In this client-server model, the DB is encrypted at the client level. The client program accesses DB directly and the encryption keys are in clear on the clients, either in the codes or in the application's configuration files. Thus, they are totally submitted to each user that executes the programs. The users can easily get the encryption keys and attack the DB.

In the 3-tier client-server architectures, we have 3 essential components [26]: the client machine (first level), the application server (second level), and the DB server (third level). In this model, often the keys are stored in clear on the application server and protected using a password in a well-defined server location. Likewise, they are completely submitted to the application server administrator. A simple attack scenario in the 3-tier architecture is a conspiracy attack conducted by a user and the application server administrator. A malicious administrator can reveal all the encryption keys to another user in order to attack DB and decrypt its sensitive data completely without leaving any traces.

## III. Proposed Solution

The principle of the proposed solution is to outsource the client's encryption key protection to the DB server. In fact,

before sending the user's query, which contains an encryption key (or multiple encryption keys) to the DB server, we transform it into a new query that converts that key (those keys) to the real encryption/decryption key of the DB. For this purpose, we propose and implement two types of solutions. The first one concerns the protection of encryption keys when inserting data into a single sensitive column, while the second concerns the protection of keys when inserting data into multiple sensitive columns.

Let's consider Table I which has the structure (A) and let us suppose the attribute "Salary" as a sensitive column to be encrypted during insertion and decrypted during the consultation of Table I.

Table I (code, Last_name, First_name, Salary) (A)

*A. Solution1: Inserting in a Single Sensitive Column*

Let's consider a user who executes an insert query having the following form:

Insert into Tab1 values ('0001', 'elbouchti', 'karim', AES192 ('Mycolor012345678has@ml@p', 2000)); (B)

The user inserts in Tab1 a data line, the column "Salary" will be encrypted with the algorithm AES192 by using the key K="Mycolor012345678has@ml@p". Before sending this query to the DB server, it will be transformed to another query having the syntax below:

Insert into Table I values ('0001','elbouchti','karim', AES192(Func ('Mycolor012345678has@ml@p'), 2000)); (C)

Func() is a function called when the user executes an insert query on a DB table, which has a single sensitive column. The description of Func() is not defined in the files of the user's application or the application server. Its main role is to transform the encryption key value defined in the query that comes out of the user's application into a new encryption key whose value is the real encryption/decryption key of the column (see Fig. 2).

We propose to implement the function Func() in the DB server, it will be one of the objects created inside it. Its model follows the algorithm below:

Func (X)

{K = E (X, H (Nom_table));

Retour (K);}

With:

- E: A symmetric encryption algorithm.
- H: A hash function.
- X: The key defined in the insert statement (user's application files).
- K: the real key to encrypt/decrypt the data of the column.
- Table_name: The table name defined in the insert statement.

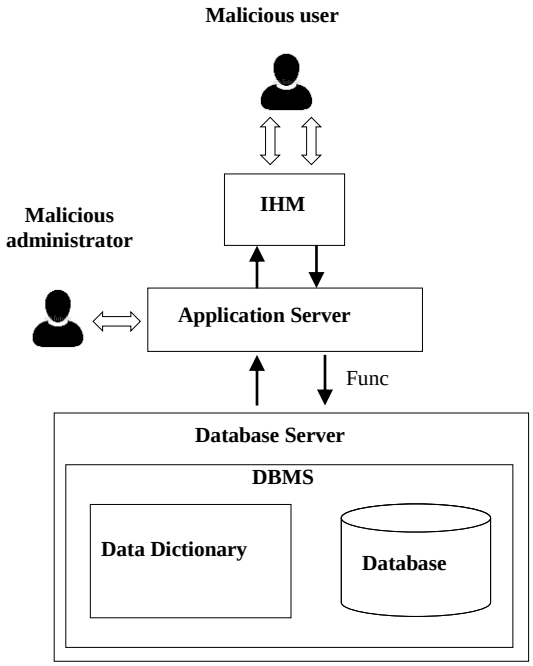


Fig. 2. The Call of Func() Function.

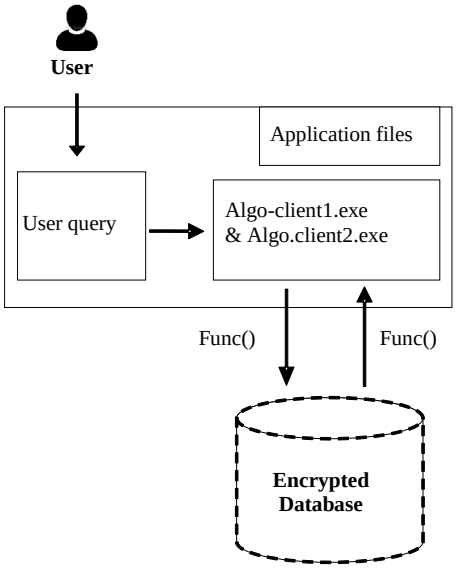


Fig. 3. Process of the Query Converting.

The conversion of query (B) into the query (C) is performed by a transformation algorithm (Algorithm 1 and Algorithm 2 defined below in the two implementations of our solution). These algorithms must appear in executable mode among the files of the application as shown in Fig. 3.

*B. Implementation of the Solution 1*

The implementation of solution 1 has been performed as follows:

- We have developed an application and connect it to a DB named "DB_karim" developed under ORACLE10G.

- We have chosen the AES256 as an encryption algorithm and the MD5 as a hash function to develop the function Func().

When a user executes an insert query, it will be transformed by the algorithm "Algo-client1" into a new query that calls the function Func() of the DB. The code of the "Algo-client1" is defined as follows:

**Algorithm 1:** Query transformation in solution1.

**Algo-client1**

```
Input: executed_query
Output: new_query
Begin
Decompose (executed_query)
    Func(key)
    Generate (new_query)
    Execute (new_query)
End
```

The Procedure Decompose() allows the decomposition of the user's query and search exactly the key to substitute by the real encryption key.

The function Func() takes the encryption key defined in the user's query and delivers the real encryption key.

The Function Generate() reformulates the new query with the real encryption key and subsequently send and execute this query using Execute().

The test of "Algo-client1" was performed on the "agent" table which has the following structure:

agent (code, first_name, last_name, dose) (D)

In this table, we have defined the column "dose" as a sensitive column. It will be protected by an encryption using the AES256 algorithm and the key k = '@mysonmyson@123'. Table I and Table II show the results obtained before and after the encryption

### *C. Solution 2: Insert in Multiple Sensitive Columns*

The principle of protecting encryption keys in a query when inserting data in multiple sensitive columns is similar to solution 1. The main goal is to protect the encryption key of each column distinctly from the other column keys.

Let's consider a user that executes an insert query having the following form:

Insert into Table I values ('0001', AES192('Mycolor442266775hrt@HH@T','elbouchti'), AES192('Mycolormybeautyy@nn@x','karim'),AES192('Mycolor012345678has@ml@p', 2000); (E)

Before sending that query, it will be transformed into the query (F) having the syntax:

Insert into Table I values ('0001', AES192(Func_Mc('Mycolor442266775hrt@ HH @ T','elbouchti')), AES192(Func_Mc('Mycolormybeautyy@ nn @ x','karim')), AES192(Func_Mc('Mycolor012345678has @ ml @ p')), 2000); (F)

TABLE. I. THE TABLE "AGENT" BEFORE ENCRYPTION

| code | first-name | last_name | dose |
|---|---|---|---|
| 1000 | Karim | El bouchti | 10 |
| 1001 | hamid | Afane | 10 |
| 1002 | adil | Faris | 15 |
| 1003 | amina | Lemnawar | 04 |
| 1004 | tayebi | El bouchti | 06 |
| 1005 | amina | Aghbal | 09 |

TABLE. II. THE TABLE "AGENT" AFTER ENCRYPTION

| code | first-name | last_name | dose |
|---|---|---|---|
| 1000 | Karim | El bouchti | 3C0354B2295D6898C 5BB2731CB6ADCB9 |
| 1001 | hamid | afane | 3C0354B2295D6898C 5BB2731CB6ADCB9 |
| 1002 | adil | faris | ADE96B60B117A764 F5870665DF4F3D7D |
| 1003 | amina | lemnawar | 88659CA5130C28136 F6AD7B37F74E531 |
| 1004 | tayebi | Elbouchti | 29152E11B0AEA7CD 69F4327E6AD4730D |
| 1005 | amina | aghbal | C1EDB798F41A5C19 F18B2A3E35D6ED92 |

In this case, the Func_Mc() function is defined and stored in the DB server. It substitutes the keys defined in the query (E) that comes out from the user's application into the real keys of encryption/decryption.We suggest implementing Func_Mc() in the DB server among its objects. It should follow the algorithm below:

Func_Mc (X)

{K = E (X, H ((Nom_table) || (Nom_col)) || (Nom_DB) || Sum (id_col, Id_tab)));

Retour (K) ;}

With:

- E: A symmetric encryption algorithm.
- H: A hash function.
- X: The key defined in the insert statement (user's application files)
- K: the real key to encrypt/decrypt the data of the column.
- Nom_col: The column name defined in the insert statement.
- Nom_DB: The name of DB.
- Sum (id_col, Id_tab): the sum of the column and table identifiers.

### *D. Implementation of the Solution 2*

The implementation of the solution 2 was performed using the same tools deployed in the solution1. We have chosen the AES256 as an encryption algorithm and the MD5 as a hash function to develop the function Func_Mc().

The algorithm "Algo-client2" presented below, transforms the user's query into a new one which calls Func_Mc() before its execution on the DB server.

**Algorithm 2**: Query transformation in solution 2.

```
Algo-client2
Input: executed_query
Output: new_query
Begin
Decompose (executed_query)
    Func_Mc(key)
    Generate (new_query)
    Execute (new_query)
End
```

We have tested the algorithm "Algo-client2" on the table "travailleur" which has the structure (G) below:

travailleur (Matricule, dose_prof, dose_sup, dose_neut, dose_int) (G)

We have considered all the "travailleur" columns as sensitive columns. Thus, these columns will be protected by an encryption according to the elements of the Table III and Table IV:

Table V and Table VI show the results obtained when inserting data in "travailleur" table before and after the encryption.

### *E. Results and Discussions*

The results obtained when implementing our two solutions allow us to deduce that both solutions provide better protection of encryption keys, no malicious act performed by the client to hold encryption keys can succeed. The real values of keys are not defined on the client.

Outsourcing the protection of the encryption keys defined on the client side to a DB server is an effective way to secure keys, especially if the server is a trusted site. Our solutions are well adapted to this concept; they are conditioned by a high level of trust at DB server and its operators. Shmueli et al. have mentioned in [11] that the level of trust in the DB server is a fundamental criterion of a DB encryption solution. The partial trust scenario is one of the three levels of trust mentioned in their manuscript. It means that the DB server, together with its memory and the DBMS software are trusted, but the secondary storage that it uses is not. Thus, we choose to integrate our solution into this concept.

TABLE. III. THE ENCRYPTION KEYS ASSOCIATED TO THE "TRAVAILLEUR" COLUMNS

| Sensitive column | Algorithm | Encryption key defined in the user query |
|---|---|---|
| Matricule | AES256 | &@I@encrypt@my@ |
| dose_prof | AES256 | &@I@encrypt@he@ |
| dose_sup | AES256 | &@I@encrypt@sh@ |
| dose_neut | AES256 | &@I@encrypt@it@ |
| dose_int | AES256 | &@I@encrypt@we@ |

TABLE. IV. THE ENCRYPTION KEY GENERATED BY FUNC_MC()

| Sensitive column | Encryption key defined in the user query | The encryption key generated by Func_Mc() |
|---|---|---|
| Matricule | &@I@encrypt@my@ | B50C849772BA517C 27C52327D2CF06B9 |
| dose_prof | &@I@encrypt@he@ | E03D47BBED9EA464 AB0BD0DD6B9DF1DA |
| dose_sup | &@I@encrypt@sh@ | F2AB5228D454FF263 044C4D43D92AD50 |
| dose_neut | &@I@encrypt@it@ | 16307E27C51A178C B731B6F8BDC29CC8 |
| dose_int | &@I@encrypt@we@ | E817629DA45424B34 CFCDE4B4B79ADC2 |

TABLE. V. THE TABLE "TRAVAILLEUR" BEFORE ENCRYPTION

| Matricule | dose_prof | dose_sup | dose_neut | dose_int |
|---|---|---|---|---|
| A00001 | 1,2 | 10,2 | 1,2 | 0 |
| A00002 | 2,4 | 5,47 | 0,2 | 0 |
| A00003 | 6,5 | 7,21 | 0,8 | 0 |
| A00004 | 2,8 | 3,55 | 0 | 0 |
| A00005 | 1,1 | 4,7 | 0,22 | 0 |
| A00006 | 6,5 | 8,6 | 0,14 | 0 |

TABLE. VI. THE TABLE "TRAVAILLEUR" AFTER ENCRYPTION

| Matricule | dose_prof | dose_sup | dose_neut | dose_int |
|---|---|---|---|---|
| 372E7E32A13F7DB9F7FCB 9FBECAEBD3D | FAA8DB86C358211C261058 A7AEB13CC4 | 48F105C92DBA3DDC5913 C82A9B810BF0 | 0547D9BDA3BA07B0C1C9 9EC1805247C1 | 8B954F14B94DFEBB85FE 222B16AE527F |
| 767D2CF95410ECFB0F34C EC0A1B6F466 | 29C9A2C18ABD74B321FB59 523B85CFFE | 0C480ADBEF9CCB14C439 77C5174F035C | F41E9DED97E2112BE602E B10E6888A09 | 8B954F14B94DFEBB85FE 222B16AE527F |
| 62709892E338E435543A204 719A83198 | BB5FB6DFCDD9E1EAC436 ABCA446A2DDA | F55467102D30CDD54E24F D5D18A4453B | BBD0714D33B5DD58E39D 952E338E8BBB | 8B954F14B94DFEBB85FE 222B16AE527F |
| 04EA3E21C176BFCD71EE3 BA7DDB8CBA3 | 786033AD9863E08F31D8936 E5A06E716 | 5B2FCAC0F7418386B5AF B265002D09E9 | 94AA0C35085D49DFD6CC 2F4CB5606CDF | 8B954F14B94DFEBB85FE 222B16AE527F |
| 4197B8E915413B5F6BD3B1 D93950731C | 3FA4CA64C8873E2E01A0C4 3D242403D6 | 6DF8E9AB7A60CA0B85B1 FDFEBF1B2343 | 7CF257E8293174EF6980A6 7E9E80645E | 8B954F14B94DFEBB85FE 222B16AE527F |
| 586F84DB7D48EB80B2BFB 156528030C4 | BB5FB6DFCDD9E1EAC436 ABCA446A2DDA | DAA03A0B62D6A9B9C902 788A9515A2D6 | F575A63E49A188328CAB3 2A1AF9A342D | 8B954F14B94DFEBB85FE 222B16AE527F |

The proposed solutions are well adapted for 2 and 3 tier architectures. For each architecture, the key protection functions (Func () and Func_Mc ()) are implemented inside the DB server. When we choose to manage the DB inside a Cloud (DBAAS), the functions (Func () and Func_Mc ()) will be included among the outsourced DB objects.

Otherwise, these solutions can be compromised by a specific attack, which remains the retro-engineering attack. An attacker can reveal the source code from the executables of the solution files (Algo-client1 and Algo-client2) by using special tools. The attacker can get the return of the functions and can determine how these functions provide the new keys. In this case, we suggest putting all the code of the user's queries in stored procedures, which will also be placed on the DB server.

## IV. Conclusion

Databases are the favorite targets of attackers due to the values of the information they contain and their volume. They are vulnerable to several typologies of attacks, especially the internal attacks. When implementing an encryption strategy at the application level to secure DB, the internal attack threats become more dangerous for the sensitive DB, and the probability of attacking the encryption keys is considerable. The attacker can use them to decrypt all sensitive data without leaving any traces

In this work, we have proposed and implemented two solutions that protect encryption keys inside applications. The principle of these solutions is to transform the encryption keys defined at the user's query to other keys considered the real keys for encryption and decryption. This transformation is performed using functions stored within DB server. Our solution is considered efficient and simple when implementation, and it is more adapted to a trusted DB server. In forthcoming works, we aim to adapt our solutions to cover more complicated attacks made by the internal attackers on sensitive DB.

## References

[1] I. Homoliak, J.Guarnizo, Y.Elovici, M.Ochoa, “Insight into insiders and it: A survey of insider threat taxonomies, analysis, modeling, and countermeasures” ACM Computing Surveys, New York, NY, USA, vol.52, no.2, pp.30, April 2019.

[2] M. Zabihimayvan, D. Doran, “Fuzzy Rough Set Feature Selection to Enhance Phishing Attack Detection“, IEEE International C onference on Fuzzy Systems,2019, in press.

[3] K. El Bouchti, S. Ziti, F. Omary, N. Kharmoum, “A New Database Encryption Model Based on Encryption Classes” Journal of Computer Science, vol.15,no.6, pp.844-854, June 2019.

[4] K. El bouchti, N. Kharmoum, S. Ziti, F. Omary, “A new approach to prevent internal attacks on Database encryption keys” Proceedings of the International Conference Scientific Days Applied Sciences, Larache, Morocco,pp.15-16, February 2019.

[5] K. El bouchti, S.ZITI, F.OMARY, “A new approach to protect encryption keys in Database Management System”, Proceedings of the International Conference Modern Intelligent Systems Concepts, Rabat, Morocco, pp.12-13, December 2018.

[6] C.Priebe, K.Vaswani, M.Costa,M, “Enclavedb: A secure database using sgx“, IEEE Symposium on Security and Privacy (SP), USA, pp.264-278, May 2018.

[7] Y.Elovici, R.Vaisenberg, E.Shmueli. (2018). U.S. Patent No. 9,934,388. Washington, DC: U.S. Patent and Trademark Office.

[8] K.El bouchti, S.Ziti, Y.Ghazali, N.Kharmoum, “Sécurité des Bases de Données : Menaces principales et solutions de chiffrement existantes“, Proceedings of the JDSIRT Conference Information Systems, Networks Telecommunications, Meckness, Morocco, pp.13 ,November 2018.

[9] Hashim, Hassan B, “Challenges and Security Vulnerabilities to Impact on Database Systems“, Al-Mustansiriyah Journal of Science,vol. 29, no.2, pp.117-125, 2018.

[10] Zainab.S.Alwan, Manal.F.Younis, “Detection and Prevention of SQL Injection Attack: A Survey“, International Journal of Computer Science and Mobile Computing, vol. 6, no 8, pp. 5-17, 2017.

[11] E.Shmueli, R.Vaisenberg, E.Gudes, Y. Elovici, “Implementing a database encryption solution, design and implementation issues“, Computers & security, vol.44, pp.33-50, 2014.

[12] B.H.Chen, P.Y.Cheung, P.Y.Cheung, Y.K.Kwok, “Cypherdb: A novel architecture for outsourcing secure database processing“, IEEE Transactions on Cloud Computing, vol.6, no.2, pp.372-386, 2018.

[13] Deepicata. N. Soni, “Database Security: Threats and Security Techniques”, International Journal of Advanced Research in Computer Science and Software Engineering, vol.5, no.5, pp.621-624, 2015.

[14] J.D. Bokefode, S.A. Ubale, S.S. Apte, D.G. Modani, “Analysis of dac mac rbac access control based models for security. Analysis“,International Journal of Computer Applications,vol. 104, no 5, pp.6-13, 2014.

[15] I.Basharat, F.Azam, A. Muzaffar,”Database Security and Encryption: A Survey Study”, International Journal of Computer Applications, vol. 47,no.12,pp.28-34, June 2012.

[16] S. Jacob, “Protection cryptographique des bases de données: conception et cryptanalyse,“ Ph.DThesis, Université Pierre et Marie Curie-Paris VI, 2012.

[17] L.Bouganim, Y.Guo, “Database encryption”, In Encyclopedia of Cryptography and Security, Springer US, pp. 307-312, 2012.

[18] E.Shmueli, R.Vaisenberg,Y.Elovici, C.Glezer, “ Database encryption: an overview of contemporary challenges and design considerations,” ACM SIGMOD Record, vol.38, no.3,pp. 29-34, 2010.

[19] Y.Ding, K.Klein, “Model-driven application-level encryption for the privacy of e-health data”, International Conference in Availability, Reliability, and Security, IEEE, Krakov, Poland, pp. 341-346, February 2010.

[20] L.Liu, J.Gai, “A new lightweight database encryption scheme transparent to applications”, 6th IEEE International Conference on Industrial Informatics, Daejeon, South Korea, pp.135-140, July 2008.

[21] S.Sesay, Z. Yang, J.Chen, D. Xu, “A secure database encryption scheme”, Procceding of the Consumer Communications and Networking Conference, Second IEEE, Las Vegas, USA, pp.49-53, January 2005.

[22] Mattsson, U. T, “ A practical implementation of transparent encryption and separation of duties in enterprise databases: protection against external and internal attacks on databases”, Proceeding of the International Conference on E-Commerce Technology, Munich, Germany, pp. 559-565, July 2005.

[23] Y.Elovici,R.Waisenberg, E.Shmueli, E.Gudes, “A structure preserving database encryption scheme”, In Workshop on Secure Data Management, Berlin, Germany, pp.28-40, 2004.

[24] J.Wang, T.Wang, Z.Yang, Y.Mao, N.Mi, & B.Sheng, “Seina: A stealthy and effective internal attack in hadoop systems”, In International Conference on Computing, Networking and Communications, IEEE, Santa Clara, USA, pp.525-530, January 2017.

[25] T.Wang, N.S.Nguyen, N, J.Wang, T.Li, X.Zhang, N.Mi, & B.Sheng, “Rover: Robust and verifiable erasure code for hadoop distributed file systems”, The 27th International Conference on Computer Communication and Networks, Hangzhou, China, pp.1-9, July 2018.

[26] M.Kambalyal, 3-tier architecture. Retrieved On, vol.2, p.34.