National Technical University of Athens
School of Electrical and Computer Engineering

Database Security & Cryptography

Ζωή Παρασκευοπούλου 03108152
zoe.paraskevopoulou@gmail.com

Νίκος Γιανναράκης 03108054
nick.giannarakis@gmail.com

March 14, 2013

Contents

1 Introduction

Databases usually store sensitive data, such as medical records, credit card numbers and government top secret information. For this reason the database administrators should ensure proper protection of sensitive data. To achieve this, an unambiguous and consistent security policy must be followed, employing a number of mechanisms in order to create different layers of security. These mechanisms can vary from security features of the operating system (authentication, logging) to firewalls that secure the network against hackers. However such mechanisms only offer protection against an intruder, not an internal threat such as corrupted employees with access to the data. Moreover possible security leaks could make feasible for someone to overcome these mechanisms, by gaining access to the system and thus accessing all the sensitive data directly by accessing the corresponding files on disk.

To be more concise, database security can be described by the following properties [3]:

Confidentiality Information should not be accessible to unauthorized users.

Integrity Data cannot be corrupted, only authorized users should be able to modify the data.

Availability Authorized users should be able to access the data reliably at all times.

In order to further protect sensitive data stored in the database, a DBMS usually provides essential built-in security features such as access control policies. Access control provides a way of granting or restricting privileges for users or groups on database objects. However it's still possible for a malicious user to infer sensitive information even if he has insufficient privileges as it will be explained later.

When all fails the last line of defense for a DBMS is data encryption. Even if a malicious user manages to circumvent all the above mechanisms he will still need to decrypt the data. Data encryption can provide very strong security for data at rest but there are many factors to be considered before encrypting your database.

- Should the data be encrypted by the DBMS or by the application that created them?
- Who can decrypt the data?
- What are the performance impacts on the DBMS?

- Do all data need to be encrypted?

All of the above above points depict serious decisions one should consider during the design phase of a database and the applications using it.

2 Access Control

It is common that different users of a database have access to different objects of the database and so a DBMS must provide mechanisms for regulating access on it's objects. There are two main methods of access control typically provided by a DBMS [3][6]: **discretionary access control** and **mandatory access control**. While the first type of access control is characterized by the fact that a user can be granted a privilege from another user, possibly along with the ability of passing that privilege to other users, the latter is based on universal system policies which cannot be modified by single users.

2.1 Discretionary access control

Discretionary access control is based on access privileges. A user may be granted the privilege to access a certain object with a certain action (i.e. read the object, modify the object). Furthermore a user may have the privilege of passing a certain access permission to another user, with or without the "passing privilege". The exact definition of what is a user, an object and an action is DBMS dependent. For instance it is common for users to be organized in groups of users with the same privileges. Regarding objects, different levels of granularity are usually supported since for example an object can represent the whole database, a certain table or even a specific field within it. For the purposes of this section an object will refer to a table or a view.

2.1.1 SQL

Discretionary access control is supported in SQL by the commands *GRANT* and *REVOKE*. The two commands have the following syntax:

```
GRANT privileges ON object TO users [WITH GRANT OPTION]

REVOKE [GRANT OPTION FOR] privileges ON object FROM users {RESTRICT | CASCADE}
```

The *GRANT* command grants users with privileges to execute a set of actions on specific objects. A series of privileges can be defined, including *SELECT, INSERT* and *DELETE* privileges. A user may also be granted the privilege to grant a certain privilege to another user. A user that creates a table in the database has all privileges on it along with the granting privilege. On the other hand a user that creates a view must have *SELECT* privileges on all the involved tables and thus he is always granted the *SELECT* privilege for the specific view. In addition the user can grant the *SELECT* privilege for that view to other users only if he has *SELECT* privilege with granting privilege on all the involved tables. Only the owner of the database schema can execute the *CREATE, ALTER* and *DROP* commands and the privilege of executing that commands cannot be granted or revoked. The authorized privileges are stored in a table in the DBMS in the form of a privilege descriptor that includes the following elements [6]:

- The identifier of the grantor of the privilege, a special keyword *system* is used for the creator of an object.
- The identifier of the grantee of the privilege
- The action that the privilege allows
- The object on which the privilege is granted
- An attribute to indicate if the grantee can grant the privilege to others

A series of *GRANT* commands can be described by a graph known as *privilege dependency graph* where nodes represent users (privilege descriptors) and the directed edges represent the dependency between two nodes. The *REVOKE* command allows users to revoke privileges they previously granted. There are two available options for the *REVOKE* command: *CASCADE* or *RESTRICT*. When the first option is specified then the privilege is revoked from the named user and also from all the users who has been granted that privilege exclusively from the named user. This holds even if a user has been granted a privilege implicitly from the named user, for example consider the following scenario: User A has a certain privilege with grant option he grants it to user B along with the granting ability. Suppose then that user B grants this privilege to user C and users B and C has not been granted this privilege from some other user. Revoking the privilege from user A will result in revoking the privilege from both users B and C. With the latter option specified, the *REVOKE* command will be rejected if revoking the privileges from the named users results in implicitly revoking the privileges from other users.

2.2 Views

SQL also supports content-based access restrictions using *Views*. A view is a virtual table that's created by a table or another view and usually contains a subset of the original data. A view can also hold aggregated data from a table or another view. As mentioned before, to create a view, a user must be granted $SELECT$ access to all the tables directly referenced by the view. In order to be able to grant access to the view to other users too, the user must have the grant option on all tables involved in the view. However if a user grants the privilege to the view, possibly with the grant option, to another user then the second user doesn't need the privilege on the tables referenced by the view in order to access the view or grant acces to another user. Views therefore can be used for information hidding, providing a convenient way for selective access on subsets of tables which are not accessible directly [6].

2.3 Mandatory access control

Discretionary access control mechanisms are an effective way to restrict user access to data but they pose certain weaknesses. For example a malicious user who has been granted the $SELECT$ privilege without the grant option could still circumvent the discretionary access control system by making a copy of the table and as an owner of the new table be able to grant $SELECT$ privileges on the new table to other users. Another way to exploit discretionary access control systems is through trojan horses. For instance a malicious user could read a table without the required privileges by creating a table and granting $INSERT$ privileges to a user who can read the table. Then modify certain applications which the victim uses in order to copy the other user's table to the one created by the malicious user. No violation of the rules set by the DAC mechanism occurred yet the system was violated.

Mandatory access control can prevent such attacks. A popular mandatory access control model is the Bell-LaPadula [3] model which consists of objects (i.e. tables, views, records, columns), subjects (i.e. users, applications), security labels on objects and clearance for certain security label on subjects. Security labels are organized according to a partial order from the safest level to the last safe (such as TOP SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED). The Bell-LaPadula model defines two rules on all accesses to database objects:

1. The Simple Security Property - a subject with clearance to a given security level may not read an object at a higher security level.

2. The * (star) Property - a subject with clearance to a given security level must not write to any object at a lower security level.

Suppose that a mandatory access control system like the Bell-LaPadula is enforced along with Discretionary access control, in that case a trojan horse attack like the one explained above will fail. Assume a table A classified as SECRET. Suppose user Alice with clearance for SECRET and a user Eve with a lower clearance namely CONFIDENTIAL. In that case Eve can only create objects of label CONFIDENTIAL and thus create a table B with label CONFIDENTIAL at most. Therefore it is not allowed for Alice to write B because the star property is violated and Eve's attack fails.

In a multilevel system like that, consider a table with tuples each classified with a security label. Users with different clearance levels may see different versions of the table which must be kept coherent and consistent without violating the MAC rules. The major problem that arises in such a case is called *polyinstantiation*. A table is considered polyinstantiated when it contains two or more tuples with the same primary key. Polyinstantiation can occur with two different ways:

- A user with low level clearance attempts to insert a tuple which happens to have the same primary key with a higher level security tuple. Forbidding the insertion wouldn't be possible because a subject with low clearance could infer information about a higher security level object (i.e. the primary key of the existing tuple) leading to implicit violation of MAC rules. Although overwriting in place the old tuple may seem a secure solution, it could lead to serious integrity problems. The only acceptable solution is to allow the insertion, keeping both tuples with the same primary key.

- A user with high level clearance attempts to insert a tuple which has the same primary key with a lower security level tuple. Since overwriting the old tuple would explicitly violate MAC rules (note that a subject modifies an object with lower security level) the available options is either to keep only the old tuple, which could lead to severe denial-of-service problems, or to keep both the old and the new data resulting in polyinstantiation.

The design of a DBMS must take into consideration the problem of potential inconsistencies due to polyinstantiation. A number of different workarounds have been proposed by researchers none of which is suitable for all applications [12].

As we can see access control policies provide some security but they cannot be considered an adequate measure of protection for the data at stake. A malicious user could still find other means to bypass the access control policies mentioned

above. For example an intruder with access to the file system could mine the database footprint on disk. With our minds set on a layered security system, we will use other mechanisms, namely cryptography, to address such issues and complement mechanisms like access control policies, firewalls, etc.

2.4 Inference attacks

A malicious user with limited access to the database may still be able to infer information about data which are inaccessible to him. A common reason for this problem is the fact that the attacker may have external knowledge about the stored data which along with the data he has access to, allows him to infer information about the rest of the data.

This problem is mostly present in statistical databases. A statistical database is a database that only executes statistical queries. For example in a database with information regarding students, one could query about the average age of all students, the maximum GPA of all students, etc. Suppose now that a malicious user Eve knows Alice has the highest GPA in the database. Eve could learn Alice's GPA by repeatedly sending queries such as *"How many students have GPA above X"* for incremental values of X until the answer is 1. Even worse, once Eve knows the GPA of Alice she could make queries such as *"What is the maximum age of all students with GPA* $> X$*"* and in that way violate the database security.

It is obvious that defending against such attacks is a complicated task. In fact the best one can do is limit the number of queries a user may execute. Another not always feasible solution is to audit logs of queries for suspicious activity as the one described above. A typical example of a real-world problem of this kind are Census bureaus, which are responsible for collecting and releasing information about all citizens without revealing information about individuals [10].

3 Encryption

Protecting sensitive data - especially data at rest - from a malicious user such as an intruder or a corrupted employee can be achieved using database encryption. Even if a malicious user can bypass access control policies or gain access to the file system he would still need the appropriate keys to decrypt the data. The above example raises two major requirements involving data encryption. [1]

1. A secure encryption technology in order to protect sensitive data.

2. A trustworthy key generation and management scheme.

There are numerous design and implementation decisions to be considered before implementing a database encryption scheme. One should answer questions such as where the encryption takes place, in the storage level, in the database level or in the application level? How can the number of users with access to the decryption keys be minimized? Should the keys be stored in a separate location from the data? Does partial encryption of the database provide adequate security? What is the performance impact of the different database encryption strategies?

3.1 Encryption principles

Encryption serves as a mean to make sensitive data unreadable to unauthorized users. In order to encrypt data, random keys are used. The same keys are used in order to decrypt the data. Decryption of the data without access to the appropriate keys should be extremely difficult. The security of the encrypted data depends on the encryption algorithm used, the key size and the implementation of the encryption algorithm. Bad algorithm choice, small key size or a mistake in implementation can render the system insecure. In addition to security, the encryption algorithm, the key size and the implementation affects performance. For example, many databases use the DES algorithm which is considered insecure. A more secure solution is a variant of DES, the 3DES which is also popular in database products, but is 3 times slower than DES [1]. As mentioned in order to decrypt the data one only needs the key used to encrypt them, therefore inadequate key generation (predictable) or management could render the encryption useless.

3.2 Encryption impacts

Encryption of data usually imposes performance penalties which can be severe if wise planning of an encryption strategy is absent. Encryption can increase data size due to padding on the original data which is required by most block-cipher encryption algorithms such as DES, 3DES and AES. Moreover the encryption and decryption of data will certainly cause some performance degradation due to the nature of cryptographic computations, depending on the amount of data encrypted, the encryption algorithm and the key size. An additional source of performance degradation is encryption of indexed fields. Look-ups on encrypted data may require decryption of each field examined which is a major overhead [1]. However, in some cases indexing and look-ups can be done on the encrypted data,

which won't affect performance. This requires that the look-up terms will also be encrypted on every look-up. [5].

3.3 Encryption Strategies

In order to minimize the performance and space requirements of data encryption, one should carefully consider which data needs to be encrypted. One may only need to encrypt part of the data, for example in the case of a credit card number only the last four digits needs to be encrypted. Furthermore certain values are not suitable for encryption, for instance small integers and boolean values, because they can be inferred with high probability. Once we have assessed the data to be protected, we should decide where the encryption will take place depending on our security and performance requirements.

3.3.1 Encryption inside the DBMS

In this encryption strategy data will be encrypted as soon as they are stored, meaning that they will be transferred through the network in plain-text form. The data are decrypted on the database server at runtime, therefore the decryption keys must be either transferred or stored in the database which provides insufficient security against a corrupted database administrator or a malicious user who managed to authenticate himself as one. One of the advantages of this encryption strategy is that it is transparent to the applications thus no changes to the applications are necessary. Thus implementing encryption inside the DBMS is simple, however there are security and performance issues to be considered. For example some DBMS offer limited encryption options, such as slow (i.e. 3DES) or insecure (i.e. DES) encryption algorithms or lack the option of selective data encryption. Furthermore since the data are transferred in plain-text form, they are at risk during transit. Finally since all encryption and decryption operations take place at the database server there is increased load on the server. Encryption inside the DBMS may take the two following forms.

- **Storage-level encryption**
  Storage-level encryption, encrypts data in the storage subsystem therefore it is well suited for encrypting entire files and directories and protecting data at rest. However the storage subsystem has no knowledge of the database scheme and it's users thus the encryption strategy cannot be related to the user privileges. Furthermore, the choice of data to be encrypted is limited

to file granularity which may lead to increased overhead due to unnecessary data encryption. Finally, one should note that there may be unencrypted copies of sensitive data in log or temporary files.

- **Database-level encryption**
  Database-level encryption offers more flexibility regarding encryption granularity as the encryption strategy can be related to the database scheme. Data can be encrypted at table, row or column level. For example one can encrypt only certain fields of a table such as the password or social security number field, or certain rows based on a logical condition such as encrypting only salaries above 10K$/month. Database-level encryption can degrade performance as it complicates indexing of encrypted data. Indeed one needs to use special encryption algorithms such as order preserving encryption in order to perform look-ups on encrypted indexes.

3.3.2 Encryption outside the DBMS

If encryption of the data during transfer is required then a more suitable solution is to encrypt the data outside the DBMS at application level. Thus the data are transferred as cipher-texts and are stored and retrieved from the DBMS in encrypted form. This increases security during transit and also addresses the problem of excessive loads on the database server due to the encryption and decryption operations, since all encryption and decryption operations are performed on the application level. However all applications must be modified in order to support encryption and decryption capabilities. In order to get the most out of this solution it is recommended to use it along with an Encryption Server which provides encryption and decryption services to the applications in a solid and consistent way. Furthermore this solution separates the encryption keys from the encrypted data, since the keys never leave the Encryption Server, making the system more secure as an attacker must gain access to both the database and Encryption Server. In order to guarantee the security of the above strategy, it is essential that a strong authentication strategy is used between the applications and the Encryption Server ensuring that only authorized users can decrypt sensitive data. Moreover the Encryption Server should be secured against attacks by proper event logging and auditing. In conclusion this solution offers flexibility regarding the encryption algorithm which can reduce performance overheads and increase security. In addition to this, this solution is highly-scalable in regard to the number of users and encrypted databases (i.e. one can add more databases without modifying the

Encryption Server). Figure 1 summarizes the different encryption strategies as detailed in section 3.3.

Encryption Level	Storage-Level	Database-Level	Application-Level
Granularity	Coarse (page)	Fine (cell)	Fine (cell)
Choice of encryption algorithm	Yes	Limited to DBMS offers	Yes
DBMS support (indexes, etc.)	Limited	Yes	No
Server Load	High	High	Low
Implementation simplicity	Simple	Moderate	Complicated
Transparent to applications	Yes	Yes	No

Figure 1: Comparison of different encryption strategies

3.4 Key Management

Key management refers to the process of cryptographic key generation and storage. Even the most sophisticated encryption algorithms are as secure as the management of the keys. Therefore one needs to consider carefully where the keys are stored and who has access to them. Other important aspects of key management are how many encryption keys are needed, how access to the keys will be restricted to authorized users only and how often should the keys change. For instance, keeping the number of the keys low makes implementation simpler but more data is vulnerable when one key is stolen. In addition to that, the more systems that can access the keys the higher probability for a key to leak. Regarding key storage a simple approach is to store the keys in a restricted database table or a file. This approach has the drawback that all database administrators can access the keys and decrypt the sensitive data without leaving any trace.

As mentioned above a method to avoid this problem is to separate the encrypted data from the encryption key. In order to achieve this one may use specialized tamper-resistant cryptographic chipsets called hardware security module (HSM). HSM can provide a secure way to store encryption keys. Usually the HSM stores a master key used to encrypt the keys which are stored in the database server [2]. Therefore database administrators cannot directly access the encryption keys.

An additional mean to secure key management is to use a security server maybe along with a HSM, which manages security related tasks such as users, roles, privileges, encryption policies and encryption keys. As in the case of an encryption server, there is added security due to the fact that the database administrator and the security administrator must co-operate in order to decrypt the data.

3.5 Encryption Modes

In order to secure a database, cryptography is required not only for encryption of sensitive data, but for user authentication and data integrity as well. There are three categories of encryption algorithms, symmetric encryption used for encrypting data, asymmetric encryption also known as public key cryptography and used in order to securely exchange a value and hashing algorithms which among other things are used for integrity control.

3.5.1 Symmetric encryption

Symmetric encryption algorithms use the same key for encryption and decryption hence the term symmetric. Symmetric-key algorithms are mainly used to encrypt data for storage. It is obvious that in order to use a symmetric-key algorithms to exchange encrypted data between two parties, the two parties must have first established a shared key in a secure manner, this is usually achieved through public-key cryptography as we will detail in the next section. There are two types of symmetric-key ciphers:

1. Stream ciphers - Stream ciphers encrypt a message one bit at a time.

2. Block ciphers - Block ciphers encrypts a block of bits on each iteration. The plain text may require padding so that it is a multiple of the block size.

Stream ciphers are faster than block ciphers but they require the use of unique key for every message encryption which makes key management a tedious task. Therefore, due to the amount of data in databases, block-ciphers are preferred for database encryption.

Data stored in a database may have a long lifetime therefore one should ensure that a secure and future-proof encryption algorithm is used, regardless of the encryption strategy and the key management chosen. Key size is an important security factor for encryption algorithms and the National Institute of Standards and Technology (NIST) recommends a key-size of at least 128-bits for symmetric-key ciphers. Insufficient key size could expose the encrypted data to brute-force

attacks. For example the DES utilizes 56 bits key size which even though seemed sufficient at the time DES was designed, the increase in computational power of modern computers made brute-force attacks feasible in less than 24 hours. Recommended alternatives to the outdated DES algorithm include the AES (Advanced Encryption Standard) which was chosen by NIST in order to replace the DES, the 3DES and the RC5.

Block ciphers can operate in various modes, however not all operation modes are suitable for encrypting a database. For example if a block-cipher is used in electronic codebook mode (ECB), identical plain-text blocks will be encrypted to identical cipher text blocks thus allowing for patterns to appear since it is common for databases to have identical values. A more suitable mode of operation for database encryption is the cipher block chaining (CBC) mode. In CBC mode, encryption of each block depends on the previous encrypted block therefore the final cipher-text cannot be predicted.

3.5.2 Asymmetric encryption

Asymmetric encryption also known as public-key encryption utilizes two keys, a public key known to everyone and a private key known to it's owner only. The two keys are linked with mathematical properties,a message encrypted with the public key can only be efficiently decrypted using the corresponding private key.

Asymmetric encryption schemes are mainly used so that two or more parties can establish a shared key in a secure manner - which can be used in a symmetric cipher afterwards - or for digital signature algorithms used to provide authenticity and integrity of the data exchanged. Popular public-key algorithms are the RSA key-exchange algorithm and the Digital Signature Algorithm.

3.5.3 Hashing

A hash function is an irreversible function that will output a fixed size of bytes associated with the input. Hashing algorithms are mainly used for data integrity. For example, one may hash a block of data and produce a unique output. The data can then be transferred along with their hash value. In order for the recipient to verify the integrity of the received data, he needs to hash the received data and compare the hash values. A popular use of hashing algorithms in database systems is for storing passwords as the operation is irreversible, someone with access to the database cannot leak the password.

3.5.4 Database specific algorithms

Despite the performance impacts that typical encryption algorithms impose on a database their use in DBMS is common. In order to tackle these problems researchers focused on encryption schemes with database encryption in mind. Some results of this research include but are not limited to the following:

Privacy homomorphic encryption allows computations to be performed on the cipher text and obtain a result that matches computations performed on the plain text. Examples of such schemes that allows the execution of aggregation queries on encrypted data can be found in [7] and [8], however both solutions have been found to bear security holes.

Order Preserving Encryption In [4] an order preserving encryption scheme is presented. It allows building indexes on cipher-text and for direct comparisons on encrypted data. Thus equality, range queries as well as queries such as MAX, MIN and COUNT can be done directly on encrypted data. Furthermore updates on values don't break the scheme. However this scheme has also been shown to be insecure.

Fast Comparison Encryption This encryption scheme allows for fast comparison between encrypted data. Encryption and decryption is done byte-by-byte starting by the most significant byte, therefore decryption of the two values compared can be implemented with "early stopping". That is, the decryption will stop when a difference between the two values has been found [9].

3.6 Application in DBMS

Modern DBMS implement encryption features and provide primitives in order to implement an effective encryption strategy. Encryption can be enabled either through optional packages or with SQL statements. To limit encryption overhead, selective encryption is generally done at the column level. Some of the encryption features modern DBMS such as PostgreSQL, Microsoft SQL, Oracle Database, etc. pack are:

- Password Storage Encryption
  User passwords are stored in hashed form, therefore a malicious user with access to the database cannot determine the actual password.

- Selective Encryption
  The database administrator can opt to encrypt only specific fields of selected tables in order to reduce encryption overheads and protect only sensitive data.

- Data Partition Encryption
  Data partition allows encrypting the entire partition and decrypting it when the file system is mounted. This provides protection in case the disk is stolen however when the file system is mounted the files are in plain-text form.

- Password encryption over the network
  Passwords are encrypted twice on the client before sent to the server using the MD5 hashing algorithm and a random salt generated by the server at the time the connection was established. Salting prevents another connection to connect to the database server using the same encrypted password at a later time.

- Data encryption over the network
  In order to protect data on transit, SSL connections can be used.

Moreover, Microsoft SQL Server 2008 introduced Transparent Data Encryption (TDE). Transparent Data Encryption performs encryption and decryption of the physical files instead of the data but within the database system. The database is protected by a single key known as Database Encryption Key (DEK) which can be managed through a HSM. Therefore modifications to applications accessing the database are not required as the encryption is transparent to the application level. Yet if a malicious user obtained a copy of the database files he would still need to decrypt the files.

Oracle also introduced TDE in Oracle 10g although it offers different features than Microsoft's TDE. Oracle's TDE simplifies the management of encryption keys used to encrypt data at different granularity levels. A master key, which could be managed through a HSM, is used in order to encrypt keys that are used to encrypt part of the data. A table key also known as a column key is used to encrypt one or more columns in a given table, alternatively one can encrypt at tablespace level (a tablespace represents data for a set of tables and indexes) using a tablespace key [11].

Alternatively one may opt for third-party solutions that connect with the DBMS to provide encryption solutions.

4 Conclusion

Data stored in databases are usually of particular importance (i.e. financial records, health records, passwords, etc.) therefore protecting them is crucial for financial, privacy, as well as legal reasons. It is obvious that one can't rely on a single security mechanism to protect his data, there should be mechanisms protecting the data at different levels which includes but are not limited to, a firewall to protect the network, access policies control for restricting access to the database, authentication methods (e.g. passwords and digital signatures), integrity checks, event logging and frequent auditing for suspicious activities and of course data encryption. From a cryptography perspective database encryption is still an open problem and not much progress has been made to allow for more secure and efficient implementations of encryption in modern DBMS.

References

[1] RSA Security company, *Securing Data at Rest: Developing a Database Encryption Strategy, white paper, 2002.*

[2] Luc Bouganim, Yanli Guo, *Database Encryption*, 2009.

[3] Raghu Ramakrishnan, Johannes Gehrke *Database Management Systems*, *McGraw-Hill Education*, 2002.

[4] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, Yirong Xu *Order Preserving Encryption for Numeric Data*, 2004.

[5] Ulf T. Mattsson *Database Encryption - How to Balance Security with Performance*, *Protegrity Corp.*

[6] Sabrina De Capitani di Vimercati, Pierangela Samarati, Sushil Jajodia *Database Security**, 2004.

[7] Hakan Hacigumus, Balakrishna R. Iyer, and Sharad Mehrotra, *Effcient execution of aggregation queries over encrypted relational databases*, *DASFAA*, 2004.

[8] Sun S. Chung and Gultekin Ozsoyoglu, *Anti-tamper databases: Processing aggregate queries over encrypted databases*, *Proceedings of the 22nd International Conference on Data Engineering Workshops*, Washington, 2006.

[9] T. Ge and S. Zdonik, *Fast, secure encryption for indexing in a column-oriented DBMS.*, In International Conference on Data Engineering - ICDE 2007, IEEE, 2007.

[10] Dorothy E. Denning, *Cryptography and Data Security*, *Addison-Wesley*, 1982.

[11] *Oracle Advanced Security Transparent Data Encryption Best Practices*, Oracle, 2012.

[12] Marshall D. Abrams, Sushil Jajodia,Harold J. Podell eds. *Information Security: An Integrated Collection of Essays*, Essay 21,*IEEE Computer Society Press*, Los Alamos, CA, 1195