Database encryption

Luc Bouganim, Yanli Guo

Database Encryption

Luc Bouganim
INRIA Rocquencourt
Le Chesnay, FRANCE
Luc.Bouganim@inria.fr

Yanli GUO
INRIA Rocquencourt
Le Chesnay, FRANCE
yanli.guo@inria.fr

Related concepts and keywords

Database security, Data confidentiality, Hardware Security Module

Definition

Database encryption refers to the use of encryption techniques to transform a plain text database into a (partially) encrypted database, thus making it unreadable to anyone except those who possess the knowledge of the encryption key(s).

Theory

Database security encompasses three main properties: confidentiality, integrity and availability. Roughly speaking, the confidentiality property enforces predefined restrictions while accessing the protected data, thus preventing disclosure to unauthorized persons. The integrity property guarantees that the data cannot be corrupted in an invisible way. Finally, the availability property ensures timely and reliable access to the database.

To preserve the data confidentiality, enforcing access control policies defined on the database management system (DBMS) is a prevailing method. An access control policy, that is to say a set of authorizations, can take different forms depending on the underlying data model (e.g., relational, XML), and the way by which authorizations are administered, following either a Discretionary access control (DAC), Role Based Access Control (RBAC) or Mandatory Access Control (MAC).

Whatever the access control model, the authorizations enforced by the database server can be bypassed in a number of ways. For example, an intruder can infiltrate the information system and try to mine the database footprint on disk. Another source of threats comes from the fact that many databases are today outsourced to Database Service Providers (DSP). Then, data owners have no other choice than trusting DSP's arguing that their systems are fully secured and their employees are beyond any suspicion, an assumption frequently denied by facts [1]. Finally, a database administrator (DBA) has enough privileges to tamper the access control definition and to spy on the DBMS behavior.

With the spirit of an old and important principle called defense in depth (i.e., layering defenses such that attackers must get through layer after layer of defense), the resort to cryptographic techniques to complement and reinforce the access control has recently received much attention from the database community [1][2]. The purpose of database encryption is to ensure the database opacity by keeping the information hidden to any unauthorized persons (e.g., intruders). Even if

attackers get though the firewall and bypass access control policies, they still need the encryption keys to decrypt data.

Encryption can provide strong security for data at rest, but developing a database encryption strategy must take many factors into consideration. For example, where should be performed the encryption, in the storage layer, in the database or in the application where the data has been produced? How much data should be encrypted to provide adequate security? What should be the encryption algorithm and mode of operation? Who should have access to the encryption keys? How to minimize the impact of database encryption on performance?

Encryption Level

Storage-level encryption amounts to encrypt data in the storage subsystem and thus protects the data at rest (e.g., from storage media theft). It is well suited for encrypting files or entire directories in an operating system context. From a database perspective, storage-level encryption has the advantage to be transparent, thus avoiding any changes to existing applications. On the other side, since the storage subsystem has no knowledge of database objects and structure, the encryption strategy cannot be related with user privileges (e.g., using distinct encryption keys for distinct users), nor to data sensitivity. Thus, selective encryption – i.e., encrypting only portions of the database in order to decrease the encryption overhead – is limited to the file granularity. Moreover, selectively encrypting files is risky since one should ensure that no replica of sensitive data remains unencrypted (e.g., in log files, temporary files, etc).

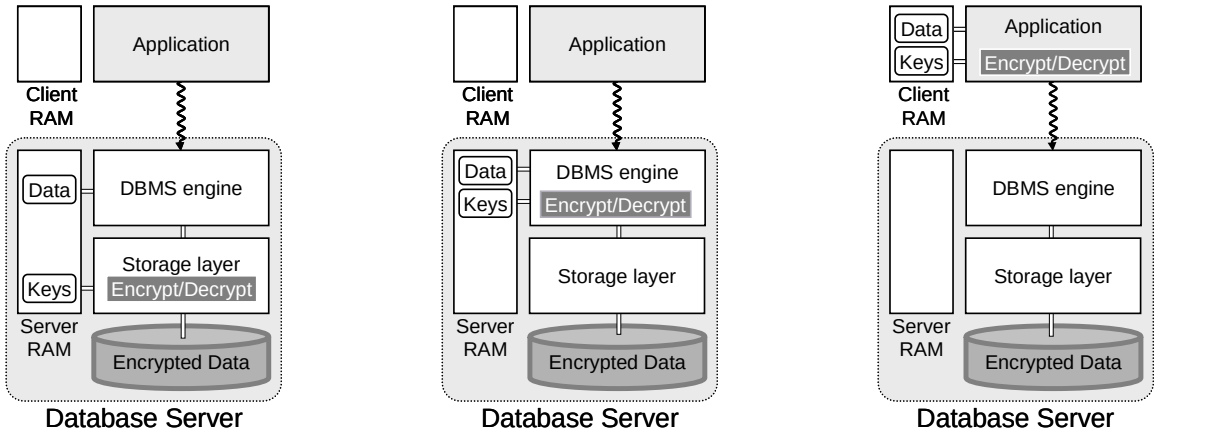
Database-level encryption allows securing the data as it is inserted to, or retrieved from the database. The encryption strategy can thus be part of the database design and can be related with data sensitivity and/or user privileges. Selective encryption is possible and can be done at various granularities, such as tables, columns, rows. It can even be related with some logical conditions (e.g., encrypt salaries greater than 10K€/month). Depending on the level of integration of the encryption feature and the DBMS, the encryption process may incur some change to applications. Moreover, it may cause DBMS performance degradation since encryption generally forbids the use of index on encrypted data. Indeed, unless using specific encryption algorithms or mode of operation (e.g., order preserving encryption, ECB mode of operation preserving equality, see below), indexing encrypted data is useless.

For both strategies, data is decrypted on the database server at runtime. Thus, the encryption keys must be transmitted or kept with the encrypted data on the server side, thereby providing a limited protection against the server administrator or any intruder usurping the administrator identity. Indeed, attackers could spy the memory and discover encryption keys or plain text data.

Application-level encryption moves the encryption/decryption process to the applications that generate the data. Encryption is thus performed within the application that introduces the data into the system, the data is sent encrypted, thus naturally stored and retrieved encrypted [1][3][4], to be finally decrypted within the application. This approach has the benefit to separate encryption keys from the

encrypted data stored in the database since the keys never have to leave the application side. However, applications need to be modified to adopt this solution. In addition, depending on the encryption granularity, the application may have to retrieve a larger set of data than the one granted to the actual user, thus opening a security breach. Indeed, the user (or any attacker gaining access to the machine where the application runs) may hack the application to access unauthorized data. Finally, such a strategy induces performance overheads (index on encrypted data are useless) and forbids the use of some advanced database functionalities on the encrypted data, like stored procedures (i.e., code stored in the DBMS which can be shared and invoked by several applications) and triggers (i.e., code fired when some data in the database are modified). In terms of granularity and key management, application-level encryption offers the highest flexibility since the encryption granularity and the encryption keys can be chosen depending on application logic.

The three strategies described above are pictured in Figure 1.



a. Storage-level encryption **b. Database-level encryption** **c. Application-level encryption**

Figure 1. Three options for database encryption level

Encryption Algorithm and Mode of Operation

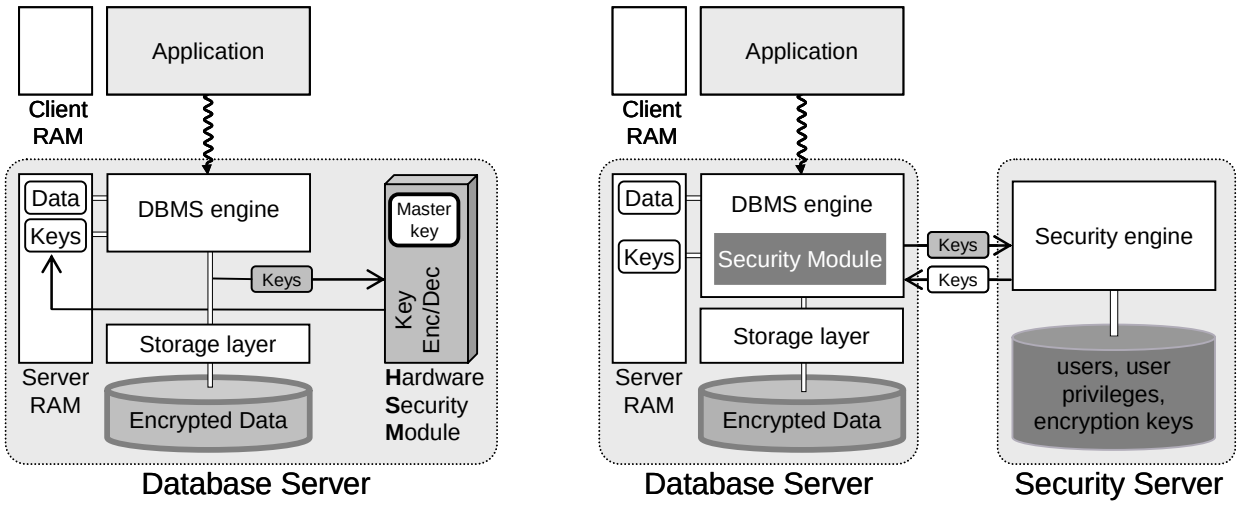
Independently of the encryption strategy, the security of the encrypted data depends on the encryption algorithm, the encryption key size and its protection. Even having adopted strong algorithms, such as AES, the cipher text could still disclose plain text information if an inappropriate mode is chosen. For example, if encryption algorithm is implemented in electronic codebook mode (ECB), identical plaintext blocks are encrypted into identical cipher text blocks, thus disclosing repetitive patterns. In database context, repetitive pattern are common as many records could have same attribute values, so much care should be taken when choosing the encryption mode. Moreover, simple solutions that may work in other context (e.g., using counter mode with an initialization vector based on the data address) may fail in the database one since data can be updated (with previous example, performing an exclusive OR between old and new version of encrypted data will disclose the exclusive OR between old and new version of plain text data). All specificity of the database context should be taken into account to guide the choice of an adequate encryption algorithm and mode of operation: repetitive

patterns, updates, huge volume of encrypted data. Moreover, the protection should be strong enough since the data may be valid for a very long time (several years). Thus, state-of-the-art encryption algorithm and mode of operation (without any concession) should be used.

Key Management

Key management refers to the way cryptographic keys are generated and managed throughout their life. Because cryptography is based on keys that encrypt and decrypt data, the database protection solution is only as good as the protection of the keys. The location of encryption keys and their access restrictions are thus particularly important. Since the problem is quite independent of the encryption level, the following text assumes database-level encryption.

For database level encryption, an easy solution is to store the keys in a restricted database table or file, potentially encrypted by a master key (itself stored somewhere on the database server). But all administrators with privileged access could also access these keys and decrypt any data within the system without ever being detected.



a. HSM Approach **b. Security Server Approach**

Figure 2. Key Management Approaches

To overcome this problem, specialized tamper-resistant cryptographic chipsets, called **hardware security module (HSM),** can be used to provide secure storage for encryption keys [14][16]. Generally, the encryption keys are stored on the server encrypted by a master key which is stored in the HSM. At encryption/decryption time, encrypted keys are dynamically decrypted by the HSM (using the master key) and remove from the server memory as soon as the cryptographic operations are performed, as shown in Figure 2.a.

An alternative solution is to move security-related tasks to distinct software running on a (physically) distinct server, called **security server**, as shown in Figure 2.b. The security server then manages users, roles, privileges, encryption policies

and encryption keys (potentially relying on a HSM). Within the DBMS, a security module communicates with the security server in order to authenticate users, check privileges and encrypt or decrypt data. Encryption keys can then be linked to user or to user's privileges. A clear distinction is also made between the role of the DBA, administering the database resources, and the role of the SA (Security Administrator), administering security parameters. The gain in confidence comes from the fact that an attack requires a conspiracy between DBA and SA.

While adding a security server and/or HSM minimizes the exposure of the encryption keys, it does not fully protect the database. Indeed encryption keys, as well as decrypted data still appear (briefly) in the database server memory and can be the target of attackers.

Applications

Since several years, most DBMS manufacturers provide native encryption capabilities that enable application developers to include additional measures of data security through selective encryption of stored data. Such native capabilities take the form of encryption toolkits or packages (Oracle8i/9i [15]), functions that can be embedded in SQL statements (IBM DB2 [5]), or extensions of SQL (Sybase [18] and SQL Server 2005 [14]). To limit performance overhead, selective encryption can be generally done at the column level but may involve changing the database schema to accommodate binary data resulting from the encryption process [14].

SQL Server 2008 [14] introduces transparent data encryption (TDE) which is actually very similar to storage-level encryption. The whole database is protected by a single key (DEK for Database Encryption Key), itself protected by more complex means, including the possibility to use HSM. TDE performs all of the cryptographic operations at the I/O level, but within the database system, and removes any need for application developers to create custom code to encrypt and decrypt data.

TDE (same name as SQL Server but different functionalities) has been introduced in Oracle10g/11g, greatly enlarging the possibilities of using cryptography within the DBMS [16]. Encryption keys can now be managed by a HSM or be stored in an external file named wallet which is encrypted using an administratively defined password. Selective encryption can be done at the column granularity or larger (tablespace, i.e., set of data files corresponding to one or several tables and indexes). To avoid the analysis of encrypted data, Oracle proposes to include in the encryption process a *Salt*, a random 16 bytes string stored with each encrypted attribute value. An interesting, but rather dangerous, feature is the possibility to use encryption mode that preserve equality (typically a CBC mode with a constant initialization vector), thus allowing, for instance, to use indexes for equality predicates encrypting the searched value.

The database-level encryption with security server approach mentioned above is proposed by IBM DB2 with the Data Encryption Expert (DEE [5]) and by third-party vendors like Protegrity [6], RSA BSAFE [17] and SafeNet [19] (appliance-based solution). The third-party vendors' products can adapt to most DBMS engine (Oracle, IBM DB2, SQL Server and Sybase).

Open Problems and future directions

Encryption Scheme

While all existing commercial database products adopt classical encryption algorithms for database encryption, specific encryption schemes have attracted much attention in the academic field, specifically in the *Database as a Service* paradigm. In this paradigm, database service providers offer its customers seamless mechanisms to create, store, and access their databases at the host site [1]. In this context, the database server may manage encrypted data without having access to the encryption keys (similar to application-level encryption).

Privacy homomorphic (PH) encryption is a form of encryption where one can perform some specific algebraic operations on the plaintext by performing (possibly different) algebraic operations on the cipher text. The first application of PH to aggregation queries in relational databases is exploited in [7], but this homomorphic encryption function is insecure against cipher text-only attacks. In [8], it supports complex aggregate queries and nested queries, but this scheme may reveal information about the input distribution, which can be exploited. Order preserving encryption scheme (OPES) [9] allows building directly indexes on cipher text. OPES can handle, without decryption, any interesting SQL query types. Unfortunately, OPES has been shown insecure in [10] and their authors introduced the fast comparison encryption (FCE) scheme for the database-level encryption strategy. FCE can be used for fast comparison through partial decryption technique. It encrypts plaintext byte by byte allowing fast comparison starting from the most significant byte and stopping as soon as a difference is found.

An alternative proposal is to use classical encryption algorithms and to store additional auxiliary fuzzy information, next to the cipher text in order to allow partial query processing on encrypted data [1][3]. Such auxiliary information shouldn't reveal plain text content, thus a trade-off exists between security and efficiency: increasing the precision of auxiliary information increases the performance since more processing can be done on encrypted data, but it also increases the risk of data disclosure.

New database encryption strategies

Currently existing architecture including database encryption are not fully satisfactory since, as mentioned above, encryption keys appears in plain text in the RAM of the server or of the client machine where the application runs. HSM acts as a safe storage to minimize the risk diminishing the keys exposure during its lifetime. Research is being conducted to make a better use of HSM, avoiding exposing encryption keys during the whole process. Two architectures can be considered: **server-HSM** when the HSM is shared by all users and is located on the server; **client-HSM** when the HSM is dedicated to a single user and is located near the user, potentially on the client machine. These two architectures are pictured in figure 3.

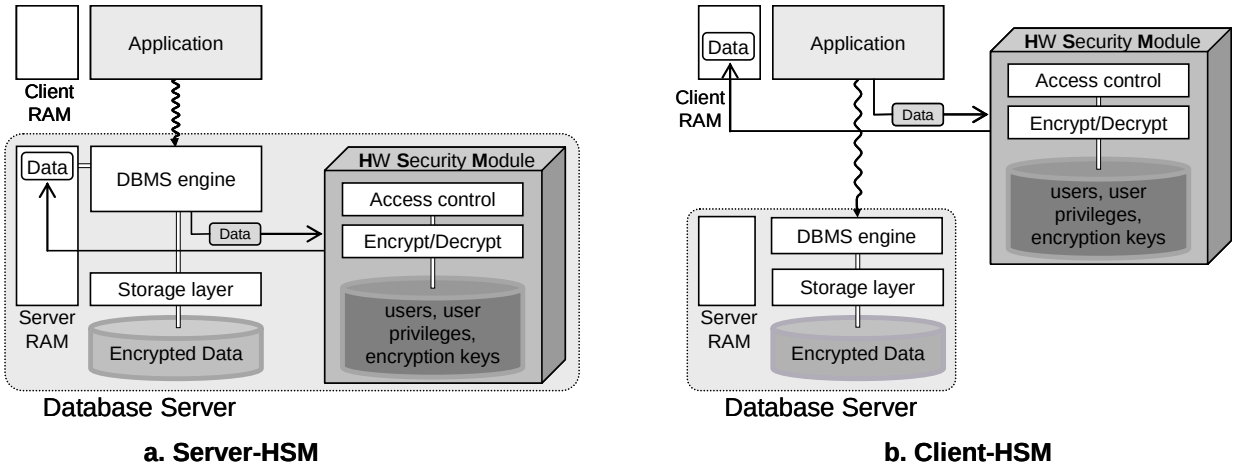


Figure 3. HSM based new database encryption strategies

Logically, the server-HSM is nothing more than a database-level encryption with a security-server embedded in the HSM. The HSM now manages users, privileges, encryption policies and keys. It has the same advantages as the database-level encryption with security-server approach but does not expose encryption keys at any moment (since encryption/decryption is done within the HSM). Moreover, the security server cannot be tampered since it is fully embedded in the tamper-resistant HSM. With this approach, the only data that appears in plain-text is the query results that are delivered to the users. The main difficulty of this approach is its complexity, since a complex piece of software must be embedded in a HSM with restricted computation resources (due to security constraints).

While the client-HSM approach seems very similar to the server-HSM one and brings the same benefit in terms of security, it poses several new challenges. Indeed, the HSM is now dedicated to a single user and is potentially far from the server thus making difficult any tight cooperation between the database server and the HSM. Thus, the database server must work on encrypted-data, and provide to the HSM a super-set of the query results, then decrypted and filtered in the HSM. Despite these difficulties, since the HSM is dedicated to a single user, the embedded code is simpler and less resource demanding, making this approach practical [4].

Other Security Issues

Encrypting the data only guarantees data confidentiality, but gives no assurance on data integrity, i.e., on the fact that the data has not been illegally forged or modified (authenticity) or replaced by older versions (freshness). In addition, if the database server is untrusted, (e.g., it may have been tampered by attackers), one should check the query results correctness (results corresponds to the query specification) and completeness (no query result is missing).

Cryptographic techniques and more specifically cryptographic hash functions are important component for building integrity checking techniques. Typically, Message Authentication Code (MAC) can be used to ensure data authenticity and, when combined with Merkle Hash Tree (MHT), can bring proofs of correctness and

completeness [11] . Ensuring freshness is more complex since an element of trust is needed to keep information about the current version of each data.

Using cryptographic techniques “as-is” to provide the aforementioned guarantees has a large negative impact on the database size (e.g., a 20 bytes MAC is added to each encrypted attribute value in Oracle 11g TDE to ensure data authenticity) and on the database performance, thus motivating many on-going research on that topics. For instance, in the Database as a Service (DAS) context [1], as MHT imposes severe concurrency constraints that slow down data updates, a new signature based authentication method has been introduced recently, for checking the authenticity, completeness and freshness of query answers [12]. In addition, some probabilistic approaches are also used to provide integrity assurance for outsourced database [13].

Recommended reading

[1] Hacigümüs H., Iyer B., Li C., Mehrotra S., Providing Database as a Service, International Conference on Data Engineering (ICDE), 2002, pp. 29-39.

[2] Rakesh Agrawal , Jerry Kiernan , Ramakrishnan Srikant , Yirong Xu, Hippocratic databases, Proceedings of the 28th international conference on Very Large Data Bases, 2002, pp.143-154.

[3] Ernesto Damiani, S. De Capitani Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati, Balancing confidentiality and efficiency in untrusted relational dbms, Proceedings of the 10th ACM conference on Computer and communications security, ACM, 2003, pp. 93-102.

[4] Luc Bouganim and Philippe Pucheral, Chip-secured data access: confidential data on untrusted servers, Proceedings of the 28th international conference on Very Large Data Bases. 2002, pp. 131-142.

[5] IBM corporation, IBM Database Encryption Expert: Securing data in DB2, 2007.

[6] Mattsson U., Transparent Encryption and Separation of Duties for Enterprise Databases: A practical implementation for Field Level Privacy in Databases, Protegrity Technical Paper, 2004, http://www.protegrity.com/whitepapers.

[7] Hakan Hacigumus, Balakrishna R. Iyer, and Sharad Mehrotra, Effcient execution of aggregation queries over encrypted relational databases, DASFAA, 2004, pp. 125-136.

[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, pp. 98-107.

[9] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu, Order preserving encryption for numeric data, Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM, 2004, pp. 563-574.

[10] Tingjian Ge and S. Zdonik, Fast, secure encryption for indexing in a column-oriented dbms, IEEE 23rd International Conference on data engineering, 2007, pp. 676-685.

[11] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin, Dynamic authenticated index structures for outsourced databases, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, 2006, pp. 121-132.

[12] HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidis, Scalable Verification for Outsourced Dynamic Databases, Proceedings of the 35th international conference on Very Large Data Bases. 2009, pp. 802-813.

[13] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng Meng, Integrity auditing of outsourced data, Proceedings of the 33rd international conference on Very large data bases, 2007, pp. 782-793.

[14] Sung Hsueh, Database Encryption in SQL Server 2008 Enterprise Edition, SQL Server Technical Article, 2008. http://msdn.microsoft.com/en-us/library/cc278098.aspx.

[15] Oracle Corporation, Database Encryption in Oracle9i, technique white paper, 2001.

[16] Oracle Corporation, Oracle Advanced Security Transparent Data Encryption Best Practices, White Paper, 2009.

[17] RSA Security company, Securing Data at Rest: Developing a Database Encryption Strategy, whiter paper, 2002.

[18] Sybase Inc, Sybase Adaptive Server Enterprise Encryption Option: Protecting Sensitive Data, 2008. http://www.sybase.com.

[19] Safenet, database encryption, 2009. http://www.safenet-inc.com/products/database_encryption/index.asp.