Database Encryption

Nicolas Anciaux, Luc Bouganim, Yanli Guo

Database Encryption

Nicolas Anciaux[1], Luc Bouganim[1] and Yanli Guo[1]

(1) Inria Saclay Île-de-France, 91120 Palaiseau, France

Nicolas Anciaux
Email: Nicolas.Anciaux@inria.fr

Luc Bouganim
Email: Luc.Bouganim@inria.fr

Yanli Guo
Email: yanli.guo@inria.fr

Related Concepts

Hardware Security Module

Definition

Database encryption refers to the use of encryption techniques to transform a plain text database into a (partially) encrypted one, 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 data confidentiality, enforcing access control policies defined in 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 a disk. Another source of threats comes from the fact that many databases are today outsourced to Database Service Providers (DSPs). Then, data owners have no other choice than trusting DSPs 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 with 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 received much attention since the early 2000s from the database community [2, 3, 4, 5] 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 through the firewall and bypass access control policies, they still need 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 the encryption be performed, 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 to 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.).

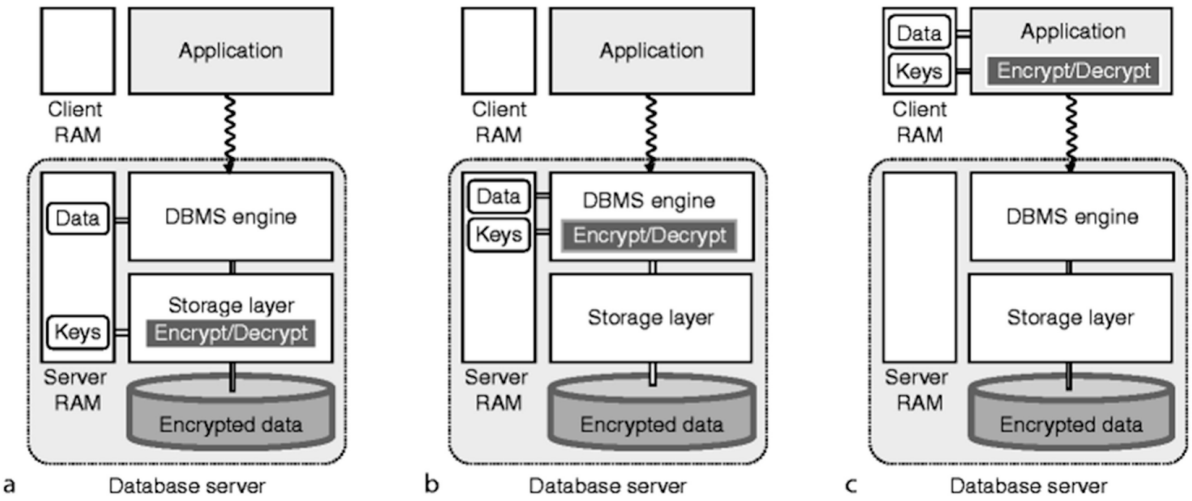
Database-level encryption allows for securing the data as it is inserted into or retrieved from the database. The encryption strategy can thus be part of the database design and can be related to data sensitivity and/or user privileges. Selective encryption is possible and can be done at various granularities, such as tables, columns, and rows. It can even be related to some logical conditions (e.g., encrypt salaries greater than 10 K€/month). Depending on the level of integration of the encryption feature and the DBMS, the encryption process may incur some changes to applications. Moreover, it may cause DBMS performance degradation since encryption generally forbids the use of indexes on encrypted data. Indeed, unless using specific encryption algorithms or modes 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 limited protection against the server administrator or any intruder usurping the administrator's identity. Indeed, attackers could spy on 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 [2, 4, 5], 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 is 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 Fig. 1.



Database Encryption. Fig. 1

Three options for database encryption level: (**a**) Storage-level encryption; (**b**) database-level encryption; and (**c**) application-level encryption

Encryption Algorithm and Mode of Operation

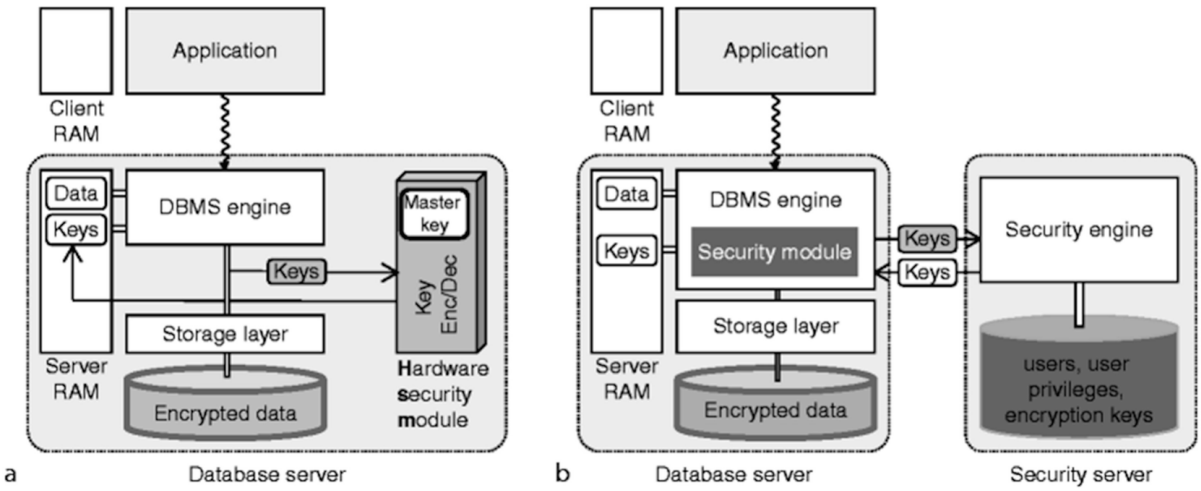
Independently of the encryption strategy, the security of the encrypted data depends on the encryption algorithm and mode of operation, the encryption key size, and its protection. Even having adopted strong algorithms, such as advanced encryption standard (AES), the cipher text could still disclose plain text information if an inappropriate mode is chosen. For example, if an encryption algorithm is implemented in electronic codebook mode (ECB), identical plaintext blocks are encrypted into identical ciphertext blocks, thus disclosing repetitive patterns. In a database context, repetitive patterns are common as many records could have the same attribute values, so much care should be taken when choosing the encryption mode. Moreover, simple solutions that may work in other contexts (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 the previous example, performing an exclusive OR between the old and the new version of encrypted data will disclose the exclusive OR between the old and the new version of plain text data). All the specificities 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, and a 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 algorithms and modes 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.

To overcome this problem, specialized tamper-resistant cryptographic chipsets, called hardware security modules (HSM), can be used to provide secure storage for encryption keys [6, 7, 8]. Generally, the encryption keys are stored on the server and 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 removed from the server memory as soon as the cryptographic operations are performed, as shown in Fig. 2a.



Database Encryption. Fig. 2

Key management approaches (**a**) HSM Approach (**b**) Security Server Approach

An alternative solution is to move security-related tasks to distinct software running on a (physically) distinct server, called a **security server**, as shown in Fig. 2b. The security server then manages users, roles, privileges, encryption policies, and encryption keys (potentially relying on an 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 the user or to the 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 collusion between the DBA and the SA.

Open Problems and Future Directions

Encryption of data **at rest** is now commonly provided at various levels by most commercial Database Management Systems (DBMS), such as Oracle [6, 7] and SQL Server [8]. However, an outstanding issue pertains to securing data **in use**. Indeed, while HSMs can be attached to DBMS for managing encryption keys, they lack the computational power to process data, and hence during execution, both keys and data are visible (in plaintext) in the DBMS's memory.

The past decade has witnessed the rise of **Trusted Execution Environments** (TEE) inclusion in numerous processor models (smartphones, computers, and servers), examples being ARM Trustzone [9] and Intel Software Guard Extensions (Intel SGX) [10]. These environments establish a tamper-resistant data processing environment [11], relying on two fundamental mechanisms:

- ***Isolation.*** The code running inside the TEE is isolated from the rest of the environment (including the operating system and hypervisor). This isolated environment is referred to as **enclave** or secure world, where external programs cannot tamper with the executing code. The data processed using the isolated code is safeguarded against eavesdropping and malicious modifications, stemming from both physical and software attacks. This protection is achieved through encryption and integrity tracking of the RAM used by the code [12].

- ***Attestation***. While executing within the TEE, the isolated code can provide proof of its identity, certifying that it operates within a genuine TEE. Its identity is established through a cryptographic hash of the memory pages containing its code [13]. This proof is signed by a secret key loaded into the CPU during manufacturing (or subsequently through remote means). This mechanism ensures the verification of computation results originating from the expected code.

Enabling DBMSs to exploit these TEE security properties and hence prevent clear-text manipulation of sensitive column values outside the TEE remains challenging [**Erreur ! Source du renvoi introuvable.**]. Research prototypes embed subpart/simplified DBMS engines in TEE, but modern TEEs (such as Intel SGX) still have memory and I/O limitations that make it difficult to develop a fully viable industrial solution. Additional challenges arise when combining DBMS and TEE (see [15] for a recent tutorial). For example, **side-channel attacks** can exploit interactions between the TEE and the untrusted environment, which leads to developing oblivious techniques to hide data access patterns inside the TEE (see e.g., [16]). Additionally, the stateless nature of TEE execution requires innovative methods to synchronize enclave-based concurrent database updates with persistent storage and logs (see [17]). Scalability might also involve multiple TEEs, demanding techniques to obfuscate information during distributed execution (see [18]).

Recommended Reading

1. Wikipedia (2023) List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches
2. Hacigümüs H, Iyer B, Li C, Mehrotra S (2002) Providing database as a service. In: International conference on data engineering (ICDE). IEEE Computer Society, Washington, DC, pp. 29–39
3. Agrawal R, Kiernan J, Srikant R, Yirong Xu (2002) Hippocratic databases. In: Proceedings of the 28th international conference on very large data bases. Morgan Kaufmann, pp. 143–154
4. Damiani E, De Capitani Vimercati S, Jajodia S, Paraboschi S, Samarati P (2003) Balancing confidentiality and efficiency in untrusted relational DBMS. In: Proceedings of the 10th ACM conference on computer and communications security. ACM, New York, pp. 93–102
5. Bouganim L, Pucheral P (2002) Chip-secured data access: confidential data on untrusted servers. In: Proceedings of the 28th international conference on very large data bases. Morgan Kaufmann, pp. 131–142
6. Oracle Key Vault, Root of Trust HSM Configuration Guide, Release 21.2, 2021. https://docs.oracle.com/en/database/oracle/key-vault/21.2/okvhm/oracle-key-vault-root-trust-hsm-configuration-guide.pdf
7. Oracle security team (2021) Securing the Oracle Database, fourth edition. https://download.oracle.com/database/oracle-database-security-primer.pdf
8. Microsoft (2023) SQL Server technical documentation, SQL Server security best practices. https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-server-security-best-practices?view=sql-server-ver16
9. Pinto S, Santos N (2019) Demystifying arm trustzone: a comprehensive survey. ACM Comput Surv, 51(6), pp. 1-36
10. Costan V, Devadas S (2016) Intel SGX Explained. Cryptology ePrint Archive
11. Sabt M, Achemlal M, Bouabdallah A (2015) Trusted Execution Environment: What It Is, and What It Is Not. IEEE Trustcom/BigDataSE/Ispa, pp. 57-64
12. Gueron S (2016) Memory encryption for general-purpose processors. IEEE Security & Privacy, 14(6), pp. 54-62

13. Brickell E, Li J (2011) Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. International Journal of Information Privacy, Security and Integrity, 1(1), pp. 1-33
14. Microsoft (2023) Always Encrypted with secure enclaves. https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves?view=sql-server-ver16
15. Anciaux N, Bouganim L, Pucheral P, Popa IS, Scerri G (2019) Personal database security and trusted execution environments: a tutorial at the crossroads. PVLDB, 12(12), pp. 1994–1997 - Slides available at https://team.inria.fr/petrus/TutorialVLDB2019/
16. Eskandarian S, Zaharia M (2019) ObliDB: oblivious query processing for secure databases. PVLDB, 13(2), pp. 169-183
17. Priebe C, Vaswani K, Costa M (2018) EnclaveDB: A secure database using SGX. IEEE Symposium on Security and Privacy, pp. 264-278
18. Schuster F, Costa M, Fournet C, Gkantsidis C, Peinado M, Mainar-Ruiz G, Russinovich M (2015) VC3: trustworthy data analytics in the cloud using SGX. IEEE symposium on security and privacy, pp. 38-54