Full Encryption: An end to end encryption mechanism in GaussDB

Jinwei Zhu
Kun Cheng
Huawei Technologies Co., Ltd
Beijing, China
zhujinwei@huawei.com
chengkun12@huawei.com

Jiayang Liu
Liang Guo
Huawei Technologies Co., Ltd
Shenzhen, China
liujiayang1@huawei.com
blue.guo@huawei.com

ABSTRACT

In this paper, we present a novel mechanism called Full Encryption (FE) in GaussDB. FE-in-GaussDB provides column-level encryption for sensitive data, and secures the asset from any malicious cloud administrator or information leakage attack. It ensures not only the security of operations on ciphertext data, but also the efficiency of query execution, by combining the advantages of cryptography algorithms (i.e. software mode) and Trusted Execution Environment (i.e. hardware mode). With this, FE-in-GaussDB supports full-scene query processing including the matching, the comparison and other rich computing functionalities. We demonstrate the prototype of FE-in-GaussDB and an experimental performance evaluation to prove its availability and effectiveness.

PVLDB Reference Format:
Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. Full Encryption: An end to end encryption mechanism in GaussDB. PVLDB, 14(12): 2811-2814, 2021.
doi:10.14778/3476311.3476351

1 INTRODUCTION

Cloud databases are facing more diversified and severer threats than ever due to the open environment and the blurry network boundary. Although most cloud database systems have adopted various protection mechanisms, such as the data encryption, the authentication, and the audit systems, etc., these defense mechanisms share one single assumption: the database users must trust the cloud infrastructure and the privileged users (e.g., administrator). However, those could also go rogue, making the user's sensitive data at great risk.

Under such a situation, several end-to-end defense mechanisms have been proposed and adopted in database systems, such as fully homomorphic encryption and property-preserving encryption [5]. Among them, the always encrypted Azure database [1] allows data owners to protect their assets at the column granularity, and utilize the Trusted Execution Environment (TEE) to safely and securely conduct complex operations against ciphertext data. Thus, even the privileged users or infrastructure providers cannot compromise the tenants' sensitive data. However, such a protection idea is yet challenging for the database design and implementation, and several researches have been proposed [1, 6–8].

In this paper, we present a novel security mechanism called Full Encryption in GaussDB (FE-in-GaussDB), providing the data confidentiality both on premise and in the cloud. Our solution combines the advantages of both software and hardware modes, and proposes a fusion strategy to freely switch among them. The software mode mainly focuses on the data encryption scheme and the index maintenance, while another one leverages the hardware TEE to securely conduct various operations against ciphertext data. With the above efforts, FE-in-GaussDB supports full-scene SQL query processing. Our main contributions are as follows.

- We explore a novel database design pattern which integrates the existing data encryption scheme together with the confidential computing by leveraging TEE.
- We carefully design and implement FE-in-GaussDB to make it transparent to the database users and applications.
- We show how FE-in-GaussDB achieves the balance between security and performance through the functional partitioning among the execution models.

Meanwhile, FE-in-GaussDB has been partially released into the openGauss community [4], which is an open-source database engine. In the following demonstration, Section 2 provides an overview of our design considerations, Section 3 describes the key design of FE-in-GaussDB, and Section 4 demonstrates the potential use cases and a comprehensive performance evaluation.

2 OVERVIEW

2.1 Threat Model

FE-in-GaussDB aims to protect the data confidentiality and integrity from the malicious access and leakage. We consider an adversary stronger than the 'honest-but-curious' [3] one. The adversary could view or tamper with the memory or disk information, as well as all external and internal communication data of database engine at any time. However, the adversary cannot view the memory data in TEE because of the isolation guarantee provided by the chip. In addition, the side channel attack, the access/behavior pattern attack, and the background knowledge attack are not considered in this work. Overall, our threat model is similar to the ones in other researches [1, 8]. The goal of FE-in-GaussDB is to provide the following guarantees:

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476351

- Correctness, if both the cloud server and the client faithfully follow the specification, the client gets the correct output for the input query, and the result can be verified by the client if necessary.
- Security, the database protects the data confidentiality, which leaves little chance for the adversary to get sensitive knowledge from the input, the storage or the output.

2.2 Combination of Software and Hardware Modes

FE-in-GaussDB combines both software and hardware security features to achieve the flexibility. The software mode includes the data encryption and index schemes for efficient equality and range comparison operations against ciphertext data, running in the Rich Execution Environment (REE, i.e. the non-secure world). The hardware mode leverages the hardware TEE (i.e. the secure world) provided by the chip to securely handle the decryption and complex computation on the ciphertext data, such as the string searching operation and the aggregation function. To achieve this, numerous efforts have been made, including:

- Modify the optimizer to determine whether the request is relevant to the cipher fields, and whether it should be processed within TEE;
- Build a two-level cipher index. The upper layer uses a label to represent different ranges. The lower layer leverages TEE to keep sequential storage;
- Both data definition language (DDL) operations and data manipulation language (DML) statements with equality query class are directly processed in database engine without accessing TEE;
- Both modes benefit from the same client encryption driver and client parser module.

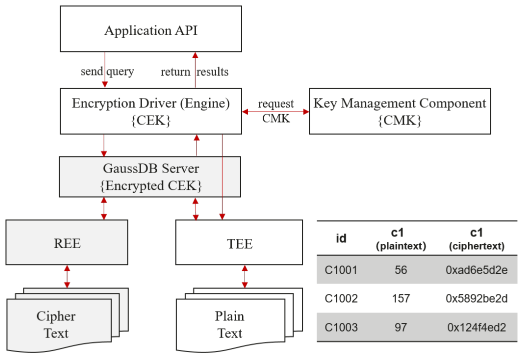
2.3 SecGear: A Virtual TEE

TEE is an isolated subsystem of the main computing platform, where code and sensitive data can be loaded and executed securely without being affected by the rest of the system. It has become a fundamental building block of security, which is also adopted by many database systems [1, 6–8]. However, since the TEE implementation varies among different Instruction Set Architectures (ISAs), all the above databases are tightly coupled with either Intel SGX or ARM TrustZone, which makes them less portable.

FE-in-GaussDB supports both SGX and TrustZone by leveraging secGear [2], which is an open-source libOS Software Development Kit (SDK). SecGear provides a uniformed set of interfaces for the trusted applications to access underlying hardware TEE resources on different platforms. Thus, FE-in-GaussDB, built on top of secGear, can be easily deployed on both Intel x86 servers and ARM aarch64 platforms. As far as we know, GaussDB is the first database system that can provide Full Encryption mechanism on both Intel SGX and ARM TrustZone.

3 ARCHITECTURE

As shown in Figure 1, the architecture of FE-in-GaussDB provides two execution modes, i.e. the software and hardware modes. The query processing in GaussDB chooses the proper execution mode based on the functional partitioning and the cost evaluation of both modes. The grayed components shown in Figure 1 is considered to be untrustworthy, while all other components are trustworthy because they either resides in the database users side or are designed to be trusted, e.g., the TEE implementation. The sensitive data is stored encrypted at column granularity in GaussDB server, together with the corresponding key encrypted.



Application API
send query
return results
Encryption Driver (Engine) {CEK}
request CMK
Key Management Component {CMK}
GaussDB Server {Encrypted CEK}
REE
TEE
Cipher Text
Plain Text

id	c1 (plaintext)	c1 (ciphertext)
C1001	56	0xad6e5d2e
C1002	157	0x5892be2d
C1003	97	0x124f4ed2

Figure 1: The adaptive architecture of FE-in-GaussDB.

With a careful design in GaussDB, our Full Encryption solution can protect the data all the time during transmission, computation and storage. FE-in-GaussDB provides the following features:

(1) a comprehensive set of DDL to manage key metadata, including Client Master Key (CMK) and Column Encryption Key (CEK);
(2) two data encryption approaches at the data column granularity, i.e. Deterministic (DET) encryption, and Randomized (RND) encryption, where DET means that the same plaintext results in same ciphertext, while RND outputs different ciphertext even with same plaintext.;
(3) the support for select, update, insert and delete queries with ciphertext data;

The whole architecture is transparent to the users and applications. By specifying the definitions of the columns that need to be encrypted, the client encryption driver encrypts the data automatically and sends them to server in ciphertext format. For parameterized query with plaintext parameters, we can get plaintext results from our execution engine. Specifically, upon each input query, the driver is designed to 1) parse the statement, 2) deduce the relation between the parameters and the encrypted column, 3) extract the encryption metadata including the encryption algorithm type and CEKs from the server, 4) encrypt the parameters to generate ciphertext query, which is sent to the database engine. Finally, the driver decrypts the query results returned from the database. The CEKs are stored in ciphertext format in GaussDB catalog. Upon each encryption/decryption requirement, the database returns ciphertext CEKs and the CMK metadata to the driver, including key ids, namespaces and so on. Since each user holds their own CMKs and CEKs, FE-in-GaussDB needs to authenticate the request to protect the users' keys from being tempered or leaked.

Although both x86 and ARM chips provide TEE, their design and implementation varies (e.g., the programming model and data

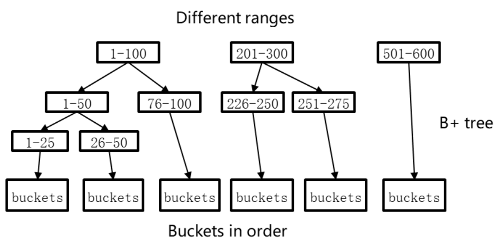


Figure 2: Ordered index of FE-in-GaussDB.

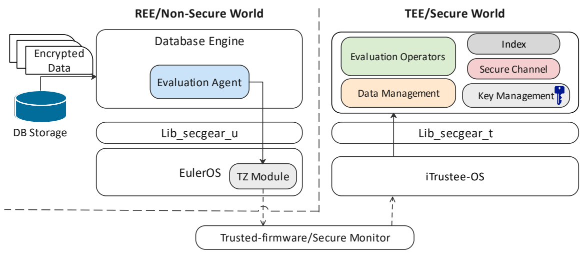


Figure 3: The hardware mode of FE-in-GaussDB.

access patterns), which causes troubles for the cross-platform development of trusted applications. Thus, the trusted functions of FE are built on secGear to retain the maintainability and portability. Once the hardware execution model is chosen, a trusted channel embedded in the SSL connection will be established between the driver and TEE to transmit the CEKs with the ECDH key exchange. Otherwise, if the software execution mode is preferred, FE will directly invoke cryptography algorithm without dealing with key information.

3.1 Key Management

The keys are the most important information in our solution. To protect the keys, FE-in-GaussDB has implemented a three-layer key management system(see Figure Figure 1), where each layer performs its own functions to secure the high-intensity key.

The bottom layer is the CEK. CEKs are used to encrypt different data of column attributes with random salt values, which guarantees the encryption isolation between various attributes. If one CEK is leaked, only the associated attribute could be affected.

The second layer is the CMK. Different users use their unique CMKs to encrypt their own CEKs, and these CMKs will never leave the users' trusted environments. Thus, even if one's data is maliciously accessed by others, they cannot decipher the ciphertext data encrypted by the data owner.

The top layer is the device key (DK). Different DKs are used to protect different CMKs. This greatly increases the difficulty of the key attack against devices.

3.2 Index

3.2.1 Equality index. An equality index is implemented on DET columns, which are in ciphertext format, by building a standard B+-Tree index. So the equality index has nothing to do with data plaintext and can support most of the index types, such as composite index and unique index. By using equality index, we can easily carry out equality operations, such as join, grouping and point lookups on DET columns without touching TEE, which has been open sourced in openGauss community. If a stronger security are required, we can build equality index within TEE on RND columns.

3.2.2 Ordered index. An ordered index can be implemented on both DET and RND columns by adopting B+-Tree index. This index can support sequential operations such as range queries and sorting. Furthermore, the data is split into different ranges, and the buckets are in order within each range, which improves the efficiency of searching without disclosing the overall order of data, as shown in Figure Figure 2. FE-in-GaussDB can also process range queries and sorting in TEE by arranging the buckets according to their corresponding plaintext order.

3.2.3 Confidentiality. We use AES-CBC algorithm to do DET or RND encryption on the sensitive data. What's more, we use the HMAC algorithm to ensure the integrity of the ciphertext, where the actual ciphertext is the concatenation of the DET or RND ciphertext and its HMAC result. Here, DET-AES-CBC mode is confidential under a weak condition.

Generally, there are two kinds of attacks on DET-AES-CBC. The first one is called the length extension attack. Since encryption is done in column-level, we can limit the number of blocks in most cases when refer to this column. The second one is remarked as inference attack. Because of deterministic, users can extract information via frequency attack or statistical analysis. By turning to RND or switching to TEE when complex queries are required, users can avoid disclose the information. Though DET or RND is called on sensitive data, a trusted channel between client and server is still needed.

3.3 Expression Evaluation within TEE

Most components of the database engine are insensitive to whether the data values are encrypted or not except for a subset of functions that directly computing data values [1, 8]. Such a set of functions are called "expression operators" in FE-in-GaussDB, which accept the direct input arguments or column data to execute their unique computation logic, such as mathematical functions, logical operations, comparison operations, string searching, data type conversion, aggregation functions (e.g., SUM, AVG) and hash functions, etc. Thus, those expression operators are implemented inside TEE as trusted code or applications, which are used to compute the expression outputs with the decrypted input values and column data.

Before the evaluation starts, one must securely send CEKs into the database server's TEE. It is done by the secure channel module though ECDH key exchange embedded in SSL connection with the client, during which the database engine serves as a gateway which has no knowledge about the keys. As shown in Figure 3, the complete evaluation process includes: 1) the database engine reads and packs all column data, user-input arguments and the demanded operation required by the query into TEE; 2) the data management module checks the input data and deciphers it if the corresponding plain data is not cached; 3) the data management module sends plain-text data into the required evaluation operator to get the output; 4) the data management module encrypts the

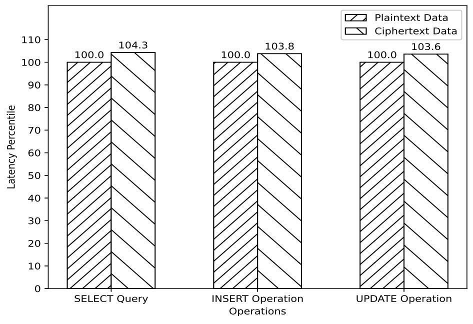


Figure 4: Performance evaluation result.

evaluation outputs and send them back to the database engine; 5) the database engine organizes the query result and sends it back.

4 DEMONSTRATION

In this section, we demonstrate our Full Encryption in three scenarios. The FE-in-GaussDB is deployed on a Taishan 2280 server with dual Kunpeng 920 CPUs and 256 GiB DDR4 ram.

4.1 Scenario 1 - Storing Data in the Cloud

Suppose a user has lots of sensitive data, and these data is called frequently. The user wants to reduce the cost by storing these data in cloud, while still keeps the data confidential. Transparent data encryption (TDE) is the traditional method adopted by cloud service, while it can only protect the data at rest. During runtime, there still have the possibility with information leakage. In contrast, by using FE-in-GaussDB, the user can store the data in ciphertext format in the cloud. It ensures that the data is encrypted during the transmission and calculation.

Though we want the data to be encrypted all the time, FE-in-GaussDB offers a comfortable performance degradation. To evaluate the performance precisely, we create two tables named tb1 and tb2. Each table contains the following 4 columns: column id with integer type, column c1 with integer type, column c2 with float type and column c3 with string type. The only difference is that the column c1 in tb1 is encrypted. Rather than using simple point query tests, we compare the latency of three common operations against ciphertext and plaintext: 1) an $INSERT$ statement which inserts 10,000 records into both tables separately; 2) a $SELECT$ query which requires all the records if c1 equals to the given value; 3) an $UPDATE$ operation against c1 if its data equals to a random value.

Each test is executed 10 times, and the average latency is calculated as the result. Finally, the normalized data is presented in Figure 4. The result shows that the FE-in-GaussDB introduces less than 5% overhead on the operation against ciphertext columns.

4.2 Scenario 2 - A Bank App Scenario

Personal data, such as bank account numbers, credit card numbers and personal phone numbers, is highly valuable and confidential to any bank customer. Recently, more and more bank applications provide a function called 'fast transfer' or 'transfer by phone number', which allows a remitter to transfer funds into a payee's account without entering the payee's account number. So the remitter enters the payee's linked phone number, and the transmission is done. In this scenario, the bank app invokes a query like: $SELECT\ account\_number\ FROM\ contacts\ WHERE\ name =$ '$Johnson'$ $AND\ phone\_number =$ '$341 - 234 - 5678$'. Directly call this query may raise a critical risk, since $account\_number$ and $phone\_number$ are very sensitive. While using FE-in-GaussDB, both $account\_number$ and $phone\_number$ can be stored as encrypted, and such an equality operation against the encrypted column can be handled without deciphering the $phone\_number$ data. Actually the driver uses the user's CEK to encrypt '13412345678' to get a ciphertext string, and then replaces the plaintext with this ciphertext. The converted query is finally sent to GaussDB, which may be deployed on data center and return results without touching the actual personal data. Moreover, the whole process is user-friendly to bank apps.

4.3 Scenario 3 - E-Business

Currently, more and more e-business companies are benefiting from the cloud services. For any on-line seller, their customers' mailing addresses are considered private and confidential. With FE-in-GaussDB, sellers can encrypt the addresses. When sellers want to learn the detailed contact information within a given city, they can issue a query as follows: $SELECT\ *$ $FROM\ custormer\_info\ WHERE\ address\ SIMILAR\ TO$ '$\%Dallas\%$'. After the optimizer in GaussDB make the decision and turn to hardware mode, the GaussDB system first establishes a trusted channel between the client and the TEE through ECDH key exchange embedded in the SSL connection. Then the converted query as explained in Sec. 4.2 is sent to GaussDB engine. The ciphertext data is sent into TEE and decrypted, then evaluated by the required operator. Finally, the evaluation result is encrypted in TEE, and it is sent back to the database engine in REE. Note that, if one directly executes the given query at the database server, the execution fails.

REFERENCES

[1] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. ACM, New York, NY, USA, 1511–1525.

[2] Chenmaodong and Huawei Co. Ltd. [n.d.]. *secGear*. Retrieved Feb. 22, 2021 from https://gitee.com/src-openeuler/secGear

[3] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, Alfred V. Aho (Ed.). ACM, 218–229.

[4] Huawei Co. Ltd. [n.d.]. *openGauss*. Retrieved Mar. 12, 2021 from https://opengauss.org/en/

[5] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 85–100.

[6] C. Priebe, K. Vaswani, and M. Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. 264–278.

[7] Pedro S. Ribeiro, Nuno Santos, and Nuno O. Duarte. 2018. DBStore: A TrustZone-backed Database Management System for Mobile Applications. In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications, ICETE 2018 - Volume 2: SECRYPT, Porto, Portugal, July 26-28, 2018*, Pierangela Samarati and Mohammad S. Obaidat (Eds.). SciTePress, 562–569.

[8] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388.