Authorized Update in Multi-User Homomorphic Encrypted Cloud Database

Tanusree Parbat, *Member, IEEE* and Ayantika Chatterjee, *Member, IEEE*

Abstract—Cloud Computing is a promising solution in distributed internet computing for IT and scientific research. However, data storage and computing in the cloud domain raise added questions in terms of security. Data encrypted with traditional encryption schemes may confirm confidentiality, but computation in cloud domain becomes infeasible. This paper focuses on designing an encrypted database considering homomorphic encryption (HE) as an underlying scheme so that query execution is carried out on encrypted version of the database without any need of intermediate decryption. Existing encrypted databases are either based on partial HE or deterministic HE to achieve practical performance; hence, they are either limited in terms of types of query execution or prone to known attacks. To mitigate these issues, we explore the practical challenges of a fully homomorphic encryption (FHE)-based database design. FHE theoretically promises to perform arbitrary operations on encrypted data. However, realizing any algorithm in homomorphic domain requires a circuit-based representation of that specific algorithm, which is a non-trivial task. In this work, we explore the practical challenges of FHE database design, mostly in the case of multi-user organizational scenarios, and propose a scheme for secure modification or conditional update of the encrypted database. Moreover, when an organization outsources a database to the cloud, a single FHE key is used to encrypt all columns of the database. However, all users(or employees) of the same organization should not have equal read and write access permission to the whole database. In this work, we propose an architecture to apply Attribute-Based Access Control (ABAC) on FHE databases with minimum overhead in terms of performance and storage. We propose required changes in registration, login, and user revocation phases for our scheme to perform conditional SQL query processing on FHE encrypted database (EDB). Our proposed framework is capable of performing end-to-end encrypted conditional UPDATE with suitable access control within 17 minutes on a multi-core processor platform for 769 rows and 9 columns of database size with 16-bit size of data. To the best of our knowledge, our proposed technique is the first one in literature to support arbitrary secure encrypted SQL query execution with suitable access control.

Index Terms—Cloud, encrypted database, FHE, SQL processing

1 Introduction

Cloud computing services like Software as a Service (SaaS) and Platform as a Service (PaaS) are helpful to enhance the scope of distributed internet computing [13]. However, recent reports of data breaches from Yahoo, JP Morgan Chase, Linkedin, eBay, Apple iCloud, and many other organizations [9] show that security is a serious bottleneck for adopting cloud to store and process critical data. To overcome such security breaches, critical data may be stored in encrypted form in the cloud [38] to maintain confidentiality. However, that adds an extra challenge in terms of processing such encrypted data. Each time data needs to be brought from the cloud to the client-side to process in an unencrypted domain, which in turn defeats the aim of cloud computing. Homomorphic Encryption (HE) is a promising solution, in this case, to support direct processing on encrypted data [20].

The ultimate version of HE is fully homomorphic encryption (FHE), which claims to support arbitrary operations on encrypted data with the aid of a noise-handling module or bootstrapping technique [1]. However, presence of such module makes FHE inherently slower for practical computations. For this reason, most of the homomorphic encrypted databases in literature [14], [26], [39] are based on partial homomorphic encryption (PHE), where need of this special noise-handling module alleviates the performance improvement. However, PHE-based databases can only support limited types of query execution. Hence, for complex query execution without any intermediate decryption, FHE support is still mandatory while designing an encrypted database.

Basic FHE SQL processing techniques, especially encrypted SELECT query execution have been discussed in [9]. The main contribution in this paper is to analyze the challenges of encrypted SQL UPDATE query for multi-user organizational cloud databases. For medical, financial, or similar organizations with confidential data, databases can be stored in FHE encrypted form for secure processing in the cloud domain. However, for processing those encrypted databases, required operators should be realized in their respective circuit-based representation, which is a non-trivial task. Hence, different query processing-related operators need to be handled separately. Moreover, in a public-key FHE-based framework, if an organization's database is encrypted with FHE public-key, multiple users can process FHE SELECT and UPDATE operations according to their requirements. Being the FHE scheme public, an adversary can trigger an UPDATE query encrypted by the public key to modify the database. Thus, illegitimate UPDATE query processing may lead to unwanted database

• *The authors are with the Indian Institute of Technology Kharagpur, Kharagpur 721302, India. E-mail: {tanusree.parbat, cayantika}@gmail.com.*

Manuscript received 1 April 2021; revised 9 July 2022; accepted 3 November 2022. Date of publication 25 November 2022; date of current version 21 June 2023.
(Corresponding author: Tanusree Parbat.)
Recommended for acceptance by W. Ku.
Digital Object Identifier no. 10.1109/TKDE.2022.3221148

modification. For a multi-user organization, it is difficult to control such malicious activity without any proper access control mechanism. In this scenario, our contributions in this paper are as follows:

1) We first highlight the challenges and security implications of implementing FHE-based SQL UPDATE. Existing FHE libraries only support bit-wise homomorphic operations like bit-wise AND, OR, etc. But practical implementation of traditional SQL query processing in FHE domain requires handling of complex operators and encrypted condition checking using those bit-wise TFHE gate operations. SQL or any practical algorithm-related operator development on top of these bit-wise gate operations is a non-trivial task considering our underlying processors are unencrypted. To elaborate the design challenge further, we take a simple example of loop execution handling, where both loop counter and loop terminating conditions are encrypted. In this case, encrypted comparison generates an encrypted outcome, which cannot be handled directly by existing unencrypted processors. Hence, dedicated encrypted building blocks are required for SQL processing operators. Existing research works [9], [36] have taken significant effort to realize general encrypted operational modules. However, in this work, we have reconstructed FHE condition check modules with an optimized form of FHE_Subtraction and proposed an overall circuit-based encrypted SQL UPDATE implementation on top of it. In this paper, we have restricted our discussion to FHE UPDATE query only.
2) Next we show in the public-key FHE assumption, any adversary only with the knowledge of public encryption key can perform illegitimate updates and that may lead to database corruption. For example "UPDATE *EDB* set COLUMN_NAME = 'Update_value' WHERE COLUMN_MANE='Enc_ Threshold"'; If adversary generates encrypted counterpart of the above query and transmits to the cloud, it can modify the partial or whole database based on selected condition. To prevent this kind of data modification attack, we incorporate a suitable access control with minimum FHE computation overhead. Since query processing on the FHE domain is inherently performance costly, the proposed idea here restricts the computation overhead in terms of a single FHE bit-level operation.
3) We also propose a secure authentication scheme including attributes with MAC address-based registration and login procedure to verify the query user before processing any query in the cloud.
4) Finally, we include a detailed security analysis of our scheme to show how this scheme maintains in-cloud data privacy for secure query processing and result generation.

The entire paper is organized as follows: In Section 2, we describe some preliminaries related to FHE scheme. Related works are discussed in Section 3. In Section 4, we introduce system and security model of our scheme. Challenges of FHE SQL query processing are described in Section 5. In Section 6, we present a technique to control FHE SQL UPDATE operation in a public-key environment. Then, we analyze performance and security of our proposed architecture in Section 7, followed by conclusion in Section 8.

2 Preliminaries

In this section, we articulate fundamental concepts of Homomorphic Encryption (HE) including partial HE (PHE), somewhat HE (SWHE), and fully HE scheme (FHE).

Let us consider an encrypted function $f : (x, \oplus) \rightarrow (y, \otimes)$, where x and y are two algebraic structures, operate on operators $\oplus$ and $\otimes$. An encryption scheme over $\oplus$ and $\otimes$ is called homomorphic, if it satisfies the following condition: $f(m_1) \oplus f(m_2) = f(m_1 \otimes m_2)$where m_1 and m_2 are sample messages which belong to all possible message set M.

PHE supports either additive or multiplicative homomorphic transformation [9]. For eample, RSA [40] and ElGamal cryptosystem [16] are multiplicatively homomorphic which means if we encrypt two messages m_1 and m_2 using encryption function ε, then by multiplicative PHE:
$\varepsilon(m_1) * \varepsilon(m_2) = \varepsilon(m_1 * m_2)$
Similarly, Goldwasser–Micali cryptosystem [22], Benaloh cryptosystem [3], and Paillier encryption scheme [34] are additive homomorphic.

SWHE [5] can handle a huge number of arbitrary additions but restricted multiplications. Leveled FHE (LHE) [6] offers a certain limit of arbitrary homomorphic operations. In this work, we focus on FHE-based design for arbitrary numbers of simple as well as complex SQL query processing without any requirement of intermediate decryption, which is otherwise not possible with PHE or SWHE schemes with limited homomorphic capability [20]. First plausible FHE construction was introduced by Gentry [20] to perform arbitrary homomorphic operations with encrypted multiplication and addition, based on lattice-based cryptosystem [27]. Gentry used bootstrapping method to reduce the noise from ciphertext and squashing method to reduce the complexity of decryption algorithm to retain homomorphic property. But this bootstrapping method is costly in terms of computation. Hence, performance improvement and cipher and/or key size reduction are the main objectives for next in-line works in this direction as detailed in [7], [8], [21], [27], [43], [46] etc.

In this work, we consider non-deterministic TFHE (Fast FullyHomomorphic Encryption over the Torus) [11] as the underlying FHE library. TFHE supports faster bit-wise homomorphism with bootstrapping in less than 0.1 seconds [10]. The main advantage of TFHE library is that there is no limitation on the number of TFHE gates or their configurations. That provides flexibility in case of general-purpose algorithm design. However, realization of general-purpose algorithms in a circuit-based representation is not straightforward, and different operators require different module development. Considering TFHE as an underlying library, FHE_Addition, FHE_Subtraction, FHE condition check modules (FHE_Is_Equal, FHE_Is_Greater/lesser) (as described in [9]) will be used in our proposed framework.

Our proposed framework uses the randomized FHE scheme which is considered to be IND-CPA secure for any probabilistic polynomial time (PPT) algorithm S if it holds following assumption good: $Pr[S(pk, Enc(pk, m_0)) = 1] - Pr[S(pk, Enc(pk, m_1)) = 1] \leq negl(\lambda)$ where negl(λ) is negligible functions over security parameter (λ), pk is public key and $m_0, m_1 \in M$ [message space].

3 Related Works

Few existing encrypted databases in literature are CryptDB [39], ZeroDB [15], Mylar, Arx, and Seabed [35] etc. In ZeroDB, all data processing occurs on the client-side. So multiple query round-trips happen between client and server until the desired data is retrieved from the server. On the other hand, CryptDB achieves encrypted processing by executing SQL queries on encrypted data under different encryption schemes; hence intermediate decryption is required. That leads to different attacks during the change of encryption schemes. Microsoft SQL Server 2016 and Microsoft Azure SQL database [17] with Always Encrypted (AE) require a trusted key to support equality comparison on deterministic encrypted data. Second version of AE with secure enclave [2] requires enclave-enabled keys to operate plaintext within enclave. However, Grubbs et al. have shown [24] that databases with deterministic encryption have some security issues, and they are susceptible to possible data leakage. In other encrypted databases, schemes are prone to several vulnerabilities due to the property-preserving characteristics [25]. In [32], authors proposed top-k query processing on the encrypted database in which objects of database relation are encrypted by homomorphic encryption followed by secure hashing. According to this paper, the server can perform only homomorphic equality check computation between two objects. Other condition checking computations are not possible due to hashing. Hence, this model is not practical when the encrypted database is deployed in cloud platforms. Boneh et al. [4] proposed an efficient third-party protocol exploiting additive and SWHE technique to support the conjunction queries execution. In this scheme, user obtains a set of matching records followed by equality hashing conditions. This scheme is not capable of handling multiple user queries together. In other proposed works [30], [41], authors presented efficient leveled FHE-based and SWHE-based private query processing schemes, respectively, which only address SELECT query execution in the cloud server. But UPDATE query-related challenges are not explored in these papers.

In literature, adequate efforts have been made to design RBAC-based encryption (RBE) scheme that integrates cryptographic techniques with role-based access control (RBAC) [47]. However, such encryption is suitable for secure storage but not for encrypted processing. In [26], hierarchical role-based multi-level access control model on paillier encrypted database breaks down if we assign too much permission to a role. In [44], authors have proposed trust and role-based hierarchy with multi-granular operational access rights that allow fine-grained access control for performing computations on a homomorphic encrypted file stored in the cloud. However, one major drawback in this scheme is that every user's request for confidential file operation is forwarded to the data owner after trust value of the user's role is granted. In [14], authors explain a policy to retrieve FHE encrypted message, where FHE key needs to be stored in cloud under some access control policy.

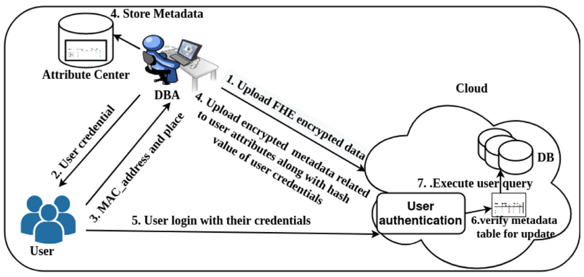

4. Store Metadata
Attribute Center
DBA
1. Upload FHE encrypted data
Cloud
4. Upload encrypted metadata related to user attributes along with hash value of user credentials
2. User credential
3. MAC_address and place
DB
7. .Execute user query
User authentication
6.verify metadata table for update
5. User login with their credentials
User

Fig. 1. System model.

However, traditional mechanisms for access management, such as Identity-Based Access Control (IBAC) and Role-Based Access Control (RBAC), have various constraints in dynamic users [23], [29]. Hence these are not feasible in a cloud environment due to lack of consistency and scalability in attribute management. The above discussion ensures the necessity of exploring an encrypted database where complex query can be processed without any intermediate decryption, and unauthorized access on FHE database need to be prevented. In the subsequent sections, we detail our proposed framework in this context. Our proposed scheme allows secure user revocation process inside the cloud, and it also works in large and dynamic scenarios where users and access privileges change from time to time. Without hampering the whole database system, our scheme smoothly handles data modification with minimal overhead.

4 Models and Assumptions

In this section, we explain system and security model of our proposed framework.

4.1 System Model

Proposed framework is comprised of three prime entities as shown in the Fig. 1: *Users*, Database Administrator (*DBA*) and *Cloud Server*. *Users* are employees of the organization, and their roles are assigned by DBA. *DBA* is in charge of database configuration and privilege management. Initially, DBA uploads the database $EDB = \{COLUMN_1, \ldots COLUMN_n\}$ with m data-row values in encrypted form to the cloud and sets up user access permission based on users' attributes obtained by registration and login process. After that, it stores this access permission in an access control list which is shared with the cloud in FHE encrypted form. Cloud stores this information in a metadata table. Whenever a user joins or leaves the organization, DBA is responsible to manage the database access permission. If user attributes are changed, DBA will take appropriate action on certain attributes without rearranging all user's attributes and setup. As there is no dependency between attributes of other users, changes should be minimal. Here, we assume that DBA is honest. On the other hand, *Cloud Server* provides storage, processing facilities and it is honest but curious. In this model, an added assumption is that the organizational databases can only be accessed from location-specific systems.

After successfully logging, users send query requests with encrypted predicates to the cloud. Cloud will process the users' requests according to the access permission with the help of metadata table. In that scenario, cloud first appends encrypted permission value with user requested encrypted

SQL query. If the user request is not permitted for a database update, the database will remain unchanged. According to our assumption, only UPDATE query without condition is not acceptable. In our subsequent sections, we shall elaborate on storing and processing this metadata table to maintain the confidentiality of efficient FHE query processing.

4.2 Security Model

Any privacy preserving database query should be free from access pattern disclosure or statistical leakage. Before providing security analysis of our scheme, we introduce the following assumptions and definitions:

1) *External adversaries* may be present in communication medium between legitimate users and semi-honest cloud. It is assumed that such adversaries do not have user credentials, but may gain access to FHE public key. They may attempt to access the users' information by eavesdropping on data communication. Adversaries may initiate various attacks by *impersonating legitimate users* to acquire secret parameters of other registered users. Later, they can bypass access permission and obtain encrypted data that result in a security violation.
2) *Concatenated query condition:* We consider a conditional query with the encrypted predicates to process an encrypted database securely. For that, we classify the query into conjunction ($\wedge$), disjunction ($\vee$), and threshold ($\leq, \geq, =, \neq$) based on the concatenation among the query conditions with predicated ciphertexts. Here, we consider a query Q with concatenated conditions $Q[C, l, p_c] \epsilon Q_c[n, \eta_1, \eta_2]$, where n is number of columns $C = (C_1, \ldots, C_n)$, η_1 is number of logical operation $l = (l_1, \ldots, l_{\eta_1})$ between conditions, η_2 is defined as number of encrypted predicates $p_c = (p_{c1}, \ldots, p_{c\eta_2})$.
3) *Indistinguishable query computation:* Database with query set security can be defined by nothing being disclosed beyond the textual expression with column names and logical operators in a SQL query condition because of IND-CPA secure FHE scheme.

Let Encrypted database $EDB = (C_1, \ldots, C_n)$ where column name C_i is unencrypted but value of C_i is encrypted. During transmission of a query from user to server, adversary $\widehat{A}$ can obtain textual expression and encrypted predicate value with column id (Id_Q). If $\widehat{A}$ maintains a query history $H_Q = (EDB, l, Id_Q, p_c)$, $\widehat{A}$ can get information about the column names and some encrypted predicates. Due to randomized FHE encryption function $Enc()$, same predicate value (p) generates different ciphers (say, p_{c1} and p_{c2}) in different time (say t_1 and t_2) i.e $[Dec(p_{c1}) = Dec(p_{c2}) = p]$ but $[p_{c1} \neq p_{c2}]$. Therefore, in two different time, if same predicate value is associated with same query Q over EDB, $\widehat{A}$ cannot differentiate between p_{c1} and p_{c2}. Using cryptogrphic terminology, we may rewrite it following probabilistic polynomial time (PPT) distinguishing algorithm S:

$$|Pr[S(p_{c1}) = 1] - Pr[S(p_{c2}) = 1]| \leq negl(\lambda),$$

where $negl(\lambda)$ is negligible function. Let adversary $\widehat{A}$ observes the query processing protocol during query execution, which performs circuit-based logical operation over homomorphic encrypted database and generates same size of resultant record $R_D(Q)$. From this observation, we get $\widehat{A}$'s view $A_\vee = (EDB, Q, l, Id_Q, p_c, R_D(Q))$. Now, consider $\widehat{A}$ sends a query with same query expression i.e., $Q = [EDB, Q, l, Id_Q, p_c]$ to query processing protocol and gets $R'_D(Q)$ for Q i.e., $A_{\vee'} = (EDB, Q, l, Id_Q, p_c, R'_D(Q))$. We can say that our query processing protocol achieves privacy for all queries over EDB if there exists a PPT algorithm S such that for all $A_{\vee'}$ over EDB

$$\{S(A_{\vee'})\} \overset{c}{\approx} S(A_\vee),$$

where $\overset{c}{\approx}$ indicates computational indistinguishable and $\widehat{A}$ will compute this information using the public knowledge only.

Definition 1 (Equivalent Cipher). *A cipheretext is said to be equivalent ciphertext E_c of an encrypted plaintext $Enc(m_1)$ if decryption of $Dec(E_c[Enc(m_1)])$ and $Dec(Enc(m_1))$ are equal but bit-wise $E_c[Enc(m_1)] \neq Enc(m_1)$.*

Definition 2 (Security against access pattern). *Let m_0 be a message, $Enc(m_0)$ be an encrypted message (cipher) and $E_c[Enc(m_0)]$ be an equivalent cipher of $Enc(m_0)$ after every computation process π. Then, we can say that computation process π will be secure against access pattern disclose if for any PPT algorithm S: $|Pr[S(E_c(Enc(m_0))) = 1] - Pr[S(Enc(m_0)) = 1] \leq negl(\lambda)$*

By these security definitions in semi-honest environment, if our proposed scheme does not disclose sensitive non-public information, it is considered to be secured. In the next section, we start with a simple implementation of FHE UPDATE without any access control and highlight the respective limitations. In the subsequent section, we explain our proposed framework for secure access control-based FHE UPDATE query processing.

5 Encrypted SQL Query Execution Framework Without Access Control

Structured query languages (SQL) in database management system (DBMS) are provided to ensure that users can efficiently retrieve and manipulate data. In our proposed framework, all row and column values are FHE encrypted.

5.1 Encrypted SQL SELECT Query

To implement SELECT statement, we follow the basic SELECT queries, supported by the standard (TPC-C) benchmark. Consider a database $EDB = (C_1, \ldots, C_n)$ with m rows and Val_{ij} is encrypted database value in column C_j, where i denotes any of m rows i.e., $1 \leq i \leq m$ and j indicates any of C columns i.e., $1 \leq j \leq n$. Let us consider a SELECT query with encrypted conditions *"SELECT * FROM EDB WHERE [condition$_1$] AND/OR [condition$_2$]...AND/OR [condition$_N$];"*. In this query, predicate value of $condition_1$, $condition_2$ ...$condition_N$ are FHE encrypted. For experimental purposes, one encrypted input condition has been taken, i.e., $condition_1$ means $C_j = Enc(predicate)$. To meet this condition in homomorphic domain, FHE condition check modules like FHE_Is_-Greater/lesser or FHE_Is_Equal are required and FHE "AND" and "OR" operators can combine multiple conditions together. Here, straightforward implementation steps of

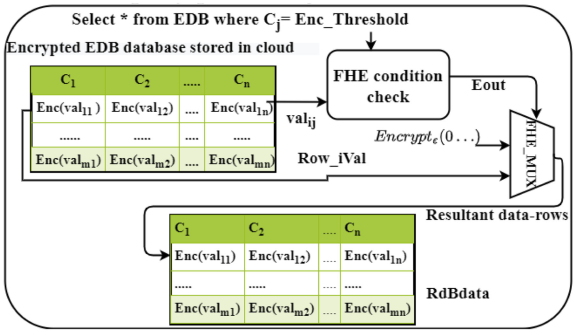


Fig. 2. Conditional encrypted SELECT operation.

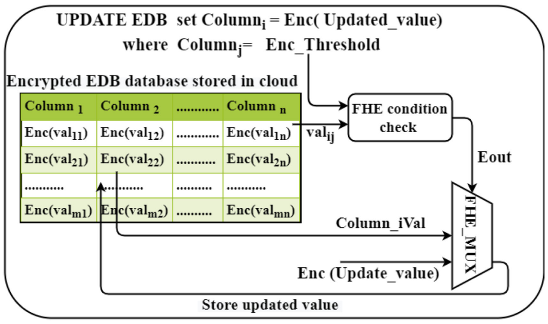

UPDATE EDB set Columni = Enc(Updated_value)
where Columnj= Enc_Threshold
Encrypted EDB database stored in cloud
FHE condition check
valij
Eout
Column_iVal
FHE_MUX
Enc (Update_value)
Store updated value

Fig. 3. Conditional encrypted UPDATE operation.

encrypted conditional SQL SELECT operation (shown in Fig. 2) are given below:

1) After receiving a SELECT query from user, cloud will perform FHE comparison operation between the query given encrypted predicate i.e., $Enc\_Threshold$ and every encrypted database value of i-th row and C_j column i.e., val_{ij} using FHE condition check module. In this experiment, we follow the equality check condition using FHE_Is_Equal module. This module generates an output $Eout$ as either $Enc(0)$ if $val_{ij} \neq Enc\_Threshold$ or $Enc(1)$ if $val_{ij} = Enc\_Threshold$. These comparison and equality check modules are developed based on FHE_Subtraction [9]. However, in this work, we propose an improved FHE_Subtractor using encrypted full subtractor, which reduces two encrypted XOR, AND, and OR gate operations in comparison to the subtraction module proposed in [9]. Based on this optimization, our overall circuit-based FHE SQL query processing is elevated in terms of performance.
2) This encrypted output $(Eout)$ is fed into an n-bit of FHE decision making module (FHE_MUX) as a select input, where n is the bit-size of column values. Our proposed design is flexible enough to choose any n as per the experimental requirement depending on the bit-size of database values. Later in result section, we will show how timing requirement of query execution is dependent on the choice of n (smaller n requires lesser execution time). The first input of FHE_MUX is n-bit of Enc(0...) and second input encrypted Row_iVal (i.e., i-th row-values, for which certain condition $(val_{ij} = Enc\_Threshold)$ has been checked) comes from database EDB. The main concept is that if the condition is satisfied for certain row-values (Row_iVal) i.e., $Eout = Enc(1)$, all/partial data row-values Row_iVal are multiplied by $Enc(1)$ to generate the equivalent cipher of valid row i.e., $E_c[Enc(non\text{-}zero)]$; otherwise, $Enc(0)$ is multiplied with the first FHE_MUX input n-bit of Enc(0...) to generate equivalent cipher of invalid row i.e., $E_c[Enc(zero)]$. In both the cases, generated resultant rows are stored in $RdBdata$ (resultant dataset).

5.2 Encrypted SQL UPDATE Query

Modifying and appending data to an existing database is achieved by 'UPDATE' query. In general, SQL UPDATE should be supported in the following form:

1) Modify some specific values from the EDB by the query: **"UPDATE EDB SET** $Column_i$ **=Update _value;"** where no conditional clause is associated. In this case, column values of the given query will be modified in the whole database without any condition checking.
2) Other type of UPDATE statement is: **UPDATE EDB SET** $Column_i$ $Column_i = Update\_value$ **WHERE** $[conditions_1]$, $AND/OR[condition_2], \ldots, [condition_n]$**;**. We shall mostly consider second type of conditional UPDATE queries in our subsequent discussion where $[condition_i]$ need to be checked with encrypted logical operators to generate an encrypted result.

In Fig. 3, implementation of encrypted conditional UPDATE operation is shown where a specific column $(Cloumn_i)$ value can be modified after satisfying some FHE conditions executed through FHE operation modules [9]. FHE encrypted UPDATE implementation with underlying FHE operators requires the following steps:

1) In order to update encrypted database EDB with m rows and n columns, we require FHE condition check module to perform encrypted logical comparison and n-bit of FHE_MUX to generate encrypted decision. Here we consider a simple UPDATE query: **UPDATE EDB SET** $Column_i = Enc(Update\_value)$ **WHERE** $Column_j$ = $Enc\_Threshold$**;**
2) Conditions are applied on encrypted values val_{ij} of ith row and jth column $(Column_j)$. Initially, encrypted values of $Column_j$ i.e val_{ij} are compared with query given encrypted predicate $Enc\_Threshold$ using FHE_Is_Equal module as this comparison is equality checking. If the output of FHE condition check module $Eout$ in Fig. 3 is $Enc(1)$ that means comparison condition $(val_{ij} = Enc\_Threshold)$ is satisfied for that certain i-th row; otherwise $val_{ij} \neq Enc\_Threshold$.
3) Finally, encrypted decision $Eout$ is given to the select input of FHE_MUX (shown in Fig. 3). First input of FHE_MUX comes from the ith row of the query given updated column $(Column_i)$ i.e., $Column\_iVal$ and user-defined newly updated value $Enc (Update\_value)$ is fed as a second input to FHE_MUX. If $Eout = Enc(1)$, EDB will be updated by equivalent cipher of $Enc(Update\_value)$ i.e., $E_c[Enc(Update\_value)]= Enc(Update\_value) \odot Enc(1)$; otherwise equivalent cipher of old column value $Column\_iVal$ will modify it.

5.3 Limitation of Proposed Encrypted UPDATE Query

It is evident that in a public-key FHE scheme, organizational users have a FHE public key to manage the encrypted database by transmitting encrypted FHE-query. The same key can be used to form an encrypted UPDATE query, and without any proper access control mechanism, any adversary with FHE encryption key can execute the UPDATE query. Hence, suitable access control with FHE UPDATE execution is an important requirement to protect FHE databases from insider/outsider attacks. If any single user account is compromised, that may lead to unauthorized data modification in the whole database. Hence, Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC) is the most natural solution incorporated with the SQL query execution. For multiuser organizations, only roles are not sufficient, and the policy needs to be extended based on attributes [47]. In conventional access control, the data owner possesses the right to specify access control policies during data upload to the cloud and can control critical data leakage issues in the untrusted environment. Attribute-Based Encryption [18], [23] offers desirable solutions to satisfy the data owner's requirements as proposed by Sahai and Waters in [42]. However, it does not provide the power of encrypted computation. Hence, we need to implement a framework to apply access control on FHE SQL query processing. Our proposed framework is also extendable to hybrid access control techniques merging RBAC and ABAC.

However, direct ABAC-related implementation on encrypted databases may incur a large number of encrypted decision-making, which can adversely affect performance. Again, attribute-based access information in plaintext form should not be outsourced to the cloud, which can be manipulated by malicious adversaries. Hence, ABAC-based encrypted UPDATE implementation should also focus on storage and computation overhead minimization. Such efficient storage and secure sharing of access control require some modifications in user registration, login, and user revocation steps. In the subsequent sections, we discuss these phases in detail.

TABLE 1
Variable Notation

Notations	Usage
DBA:	Database Administrator
UID:	User Identity
TPW:	One-time password
PW:	User Password
H(.):	One-way hash function where $H:\{0,1\}^{*}\longrightarrow\{0,1\}$
E$_p$:	An elliptic curve on finite field F_{2^m}
P:	Base point of elliptic curve
(Q_{pub}, q):	public key and private key of DBA, where $Q_{pub} = [q].P$
(Q_{pub1}, s):	public key and private key of cloud server, where $Q_{pub1} = [s].P$
r:	Random number chosen by DBA for every user
MAC_addr:	Media access control address (MAC address)
$\oplus$:	Bit-wise X-OR operator
$\|$:	Concatenation operator
Acc$_{List}$:	Access Control List
T$_i$:	Token message from cloud server
$\triangle T$:	Expected Time delay
T$_c$:	Current timestamp of cloud server
T$_u$:	Current timestamp of user
$\odot$:	Multiplication operator
$\overset{c}{\approx}$:	computational indistinguishability
π :	Query computation process
H_Q :	Query history
l:	Logical concatenated query operation
Id_Q:	Column id
p_c:	Encrypted predicate

6 Encrypted SQL Query Execution With Access Control

We mention the encrypted query processing steps of our proposed framework starting from user registration (all related variables are mentioned in Table 1):

1) When any new user joins the organization, user sends an access request to DBA for the encrypted cloud database.
2) DBA creates user login id and password for respective user and shares these with the user. Based on the users' roles and attributes, DBA creates an access control list for each user (following ABAC or hybrid RBAC-ABAC model). By access control lists (Acc_{List}) we mean an array of bits $ac_0, ac_1, \ldots, ac_{n-1}$, where n is the number of database columns and value of ac_i is FHE encrypted access permission for ac_i column of a particular login user (Table 2). DBA determines these values based on predetermined ABAC/RBAC-ABAC policies.

   In this framework, we consider the situation where the organization is uploading a single database, and all users/employees do not have same read/write (SELECT | UPDATE) permission on all the columns of the database. Access control list is to be prepared accordingly for UPDATE access. Read access (SELECT operation) is given to all the users after successful login. For any user, $ac_i = FHE(1)$ indicates write/ update access on ith column and $ac_i = FHE(0)$ indicates only read access for ith column of that particular database. An extension of this assumption is the situation where multiple databases are uploaded in the cloud by any organization, and users/employees have restricted UPDATE access permission to few databases among them. Our framework can easily handle this extension with a similar array of bits in access control list to define database-specific control. In that case, if $ac_i = FHE(1)$, that means user has write/ update access on ith database. If $ac_i = FHE(0)$, it indicates user only has read access for ith database. DBA creates this access control list in 'Registration phase'. In the user registration phase, DBA will send the Acc_{list} information to cloud in FHE encrypted form. Cloud will maintain a meta data table shown in Table 2 to store users' information with FHE encrypted Acc_{List}.
3) In the subsequent phases, user uses login information for database access ('login phase') and query processing inside the cloud.

TABLE 2
Access Control List

Login	Attributes	Encrypted Write Permission			
...	...	ac_{n-1}	ac_{n-2}	...	ac_0
$AID_1 \parallel \hat{r_1}$	$U\_attribute_1$	FHE(1)	FHE(0)	...	FHE(0)
$AID_2 \parallel \hat{r_2}$	$U\_attribute_2$	FHE(0)	FHE(1)	...	FHE(1)
$AID_3 \parallel \hat{r_3}$	$U\_attribute_3$	FHE(0)	FHE(0)	...	FHE(0)
...	...	...	...	...	...
$AID_n \parallel \hat{r_n}$	$U\_attribute_n$	FHE(0)	FHE(0)	...	FHE(1)

6.1 Registration Phase

It is important to note that cloud should have information about the mapping of access control with the corresponding user-id. However, DBA cannot share this user-id or access control information with the cloud in plaintext form. That may lead to subsequent data leakage or data modification during transmission. Access control information is shared in FHE encrypted form as that encrypted value will be used in SQL query processing in the FHE domain. However, FHE encryption leads to huge ciphertext expansion. Hence, traditional public-key encryption with low ciphertext expansion overhead is used for providing confidentiality to user-id related information during communication from DBA to cloud server. Here we assume that DBA and cloud choose elliptic curve cryptography (ECC) [31] based communication technique for initial information sharing and decide on their communication keys before actual communication. Here, we consider a $k-233$ elliptic curve(E) over prime finite field F_{2^m} defined by NIST [19], where $m = 233$ is determined by an equation : $y^2 + xy = x^3 + ax^2 + b$, where $a, b \,\epsilon\, F_{2^m}$ and $b \neq 0$. A set of all elliptic curve points E_p is defined as $\{(x, y) : x, y \,\epsilon\, F_{2^m} \;\&\; (x, y) \epsilon\, E_p\} \bigcup \{O\}$, where O is point at infinity, serving as identity element. Given an integer $k \,\epsilon\, F_{2^m}$ and a point $P \,\epsilon\, E_p)$ of order n, then point multiplication operation $[k].P \,\epsilon\, E_p$ is defined as $[k].P = \underbrace{P + P + \ldots + P}_{\text{k-times}}$. ECC parameters for initial communication can be decided as follows:

1) DBA chooses usual ECC parameters: a base point P of order n over elliptic curve equation [31] E_p and a secret key $q \in [1 \; to \; (n-1)]$ to calculate corresponding public key $Q_{pub} = [q].P$. Cloud server also selects secret key $s \in [1 \; to \; (n-1)]$ and calculates public key $Q_{pub1} = [s].P$. DBA also chooses one way hash function $H : \{0,1\}^* \longrightarrow \{0,1\}$ and publicly publishes $\{P, Q_{pub}, Q_{pub1}, H\}$.
2) While a user joins as an employee of the organization, DBA defines the designation (role), user-id (UID), and one-time password (TPW) for the user. Then it transmits $\{UID, TPW\}$ with FHE_key to the user securely through their registered mail-id.
3) After receiving these credentials, user completes login procedure and changes the one-time password (TPW) to password PW. Then, user sends $\{UID, MAC\_addr, place\}$ encrypted by DBA public key (Q_{pub}) to DBA. Here, MAC_addr is MAC address of the user's machine, and $place$ is the user's login location. These values are considered as user attributes. Several other attributes can be defined and added according to the policy requirements, which will not affect our design.
4) DBA will collect specific attributes like MAC address, Login_place (location) from the login information along with $\{UID, PW\}$ and computes: $\{H((UID \oplus r) \parallel role).P\}$, AID=$H(H(PW \parallel r) \parallel H(UID \oplus r)).P$, (r is a random number), $\hat{r} = H(MAC\_addr \parallel PW \parallel role) \oplus r$, $U\_attribute = [H(UID \oplus r) \parallel H(MAC\_addr \parallel place \parallel role \parallel r)].P$.

   Based on user's designation (role), DBA sets database modification permission known as access control list in FHE encrypted form and stores it in organization's attribute center along with $U\_attribute$ and $\{H((UID \oplus r) \parallel role).P\}$, encrypted by its public key (to avoid any login information leakage from DBA server). In case some user's attributes are changed, DBA can easily modify $U\_attribute$ based on $\{H((UID \oplus r) \parallel role).P\}$ in attribute center first and then cloud and also keep track of users. After that, DBA will transmit $[E_{Q_{pub1}}\{AID \parallel U\_attribute \parallel Acc_{List} \parallel \hat{r}\}]$ to the cloud, where AID, $U\_attribute$, a copy of FHE encrypted access control list (Acc_{List}), and $\hat{r}$ are encrypted by cloud's public key Q_{pub1} in order to authenticate user login requests in future.
5) On the other hand, after obtaining encrypted message from DBA, cloud decrypts $[E_{Q_{pub1}}\{AID \parallel U\_attribute \parallel Acc_{List} \parallel \hat{r}\}]$ using cloud's secret key and stores $\{AID, \hat{r}, U\_attribute\}$ along with FHE encrypted access control list for future use.

A pictorial representation of the registration process is shown in Fig. 4. It is important to note that user-specific FHE encrypted access permission will not be leaked to the cloud as FHE scheme is IND-CPA secure.

6.2 User Login and Authentication

During login phase, users login with user-id (UID), password (PW), and role information. After receiving $\{UID, PW, role\}$, cloud server performs the following computations:

1) Cloud gets MAC address and location from the login device information. In this framework, we assume that the organizational database can only be accessed by authorized users from location-specific systems. Attributes can be set accordingly. However, these attributes can be defined according to organizational policies, which will not affect our framework.

   Then, by processing login information, cloud calculates: $r' = H(MAC\_addr \parallel PW \parallel role) \oplus \hat{r}$ and $AID' = H(H(PW \parallel r') \parallel H(UID \oplus r')).P$ After that, cloud performs a MAC-address-based verification to check $AID_i \overset{?}{=} AID'_i$ for authenticating the login request. If $AID_i \neq AID'_i$, cloud rejects the login request and keeps a record of the MAC address from which that request has been made. These rejected MAC addresses are added to an intruder list. In future, the cloud can easily track an intruder from this intruder list before processing the users' query request and may terminate communication to prevent unauthorized user entry.

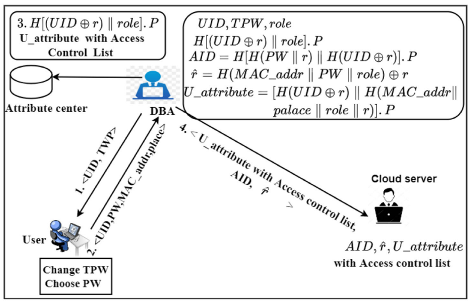
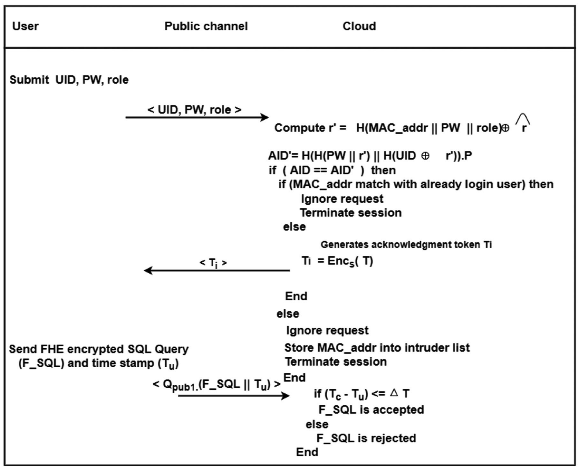

3. $H[(UID \oplus r) \parallel role].P$
U_attribute with Access Control List
Attribute center
DBA
$UID, TPW, role$
$H[(UID \oplus r) \parallel role].P$
$AID = H[H(PW \parallel r) \parallel H(UID \oplus r)].P$
$\hat{r} = H(MAC\_addr \parallel PW \parallel role) \oplus r$
$U\_attribute = [H(UID \oplus r) \parallel H(MAC\_addr \parallel palace \parallel role \parallel r)].P$
1. <UID, TWP>
2. <UID,PW,MAC_addr,place>
4. < U_attribute with Access control list, AID, $\hat{r}$ >
Cloud server
User
$AID, \hat{r}, U\_attribute$ with Access control list
Change TPW
Choose PW

Fig. 4. Registration phase.

2) Once login procedure is successful, the cloud sends a token message Ti as acknowledgment.
3) After receiving the acknowledgment from cloud, user executes the succeeding steps of sending FHE encrypted SQL query (F_SQL) along with the current timestamp (T_u) encrypted by cloud public key (Q_{pub1}).
4) After decrypting the received encrypted message, cloud obtains user's timestamp. It checks whether $[(T_c - T_u) \leq \triangle T]$, where T_c is current cloud time and $\triangle T$ is worst-case time delay for client to cloud transmission. If the condition is true, F_SQL will be processed further; otherwise message is rejected from cloud side. We have used $\triangle T$ to prevent the most known replay attack. If an eavesdropper snoops an encrypted user query during transmission through the network channel and transmits it later to cloud, cloud will reject that query after checking above condition. Thus, replay attacks can be prevented.

Fig. 5 shows pictorial representation of our authentication phase.


User
Public channel
Cloud
Submit UID, PW, role
< UID, PW, role >
< T_i >
Send FHE encrypted SQL Query (F_SQL) and time stamp (T_u)
< $Q_{pub1}.(F\_SQL \parallel T_u)$ >

```
Compute r' =  H(MAC_addr || PW || role)⊕ r̂
AID'= H(H(PW || r') || H(UID ⊕ r')).P
if ( AID == AID' ) then
  if (MAC_addr match with already login user) then
      Ignore request
      Terminate session
  else
       Generates acknowledgment token Ti
      Ti = Enc_s( T)
  End
else
  Ignore request
  Store MAC_addr into intruder list
  Terminate session
End
    if (T_c - T_u) <= △ T
      F_SQL is accepted
    else
      F_SQL is rejected
  End
```

Fig. 5. Login and authentication phase of our proposed scheme.

6.3 Access Control Based Encrypted SQL UPDATE

This section describes the proposed method (shown in Algorithm 1) for FHE update query processing based on access control information stored in the metadata table (Table 2). After receiving FHE encrypted SQL query from the user, cloud will take over the following steps:

1) It computes the received user attributes $(U\_attri_{received}) = [H(UID_{received} \oplus r') \parallel H(MAC\_addr\ received \parallel place_{received} \parallel role \parallel r')].P$ where UID $_{received}$, MAC_addr$_{received}$ and place$_{received}$ are login user-id, MAC address of user's login machine and login place respectively.
2) It finds the mapping relation with (U_attri$_{received}$) and already preserved U_attribute$_i$ in the metadata Table 2, where corresponding access control list with AID (user authorization) are available. If [U_attri$_{received}$ $\neq$ U_attribute$_i$], cloud rejects user's request to execute any query; otherwise it performs next step.
3) A query parser in the cloud gets activated, which analyses the encrypted SQL query to know whether it is a SQL UPDATE or SELECT query. If it is a SELECT type SQL query, cloud performs normal FHE SELECT query execution (shown in Fig. 2); otherwise, it accesses preserved Acc_{List} for database modification permission. If the UPDATE request for column c_i is ac_i, then cloud fetches FHE encrypted ac_i value from the metadata table and sends the same as an input (Enc_Uattri) to final encrypted SQL UPDATE execution block. If Enc_Uattri = FHE (1), then user has UPDATE access, and database is modified according to encrypted query execution; otherwise, database retains previous database entries in an equivalent form. In both cases, cloud handles users' update requests to execute the query without knowing Enc_Uattri in plaintext form.

Fig. 6 shows the implementation to perform encrypted SQL UPDATE with access control as mentioned in Algorithm 1. The access information bits from FHE encrypted access table are used as an input to SQL UPDATE query processing module and the requested SQL UPDATE query with encrypted access control permission is as follows: **UPDATE EDB SET** $Column_i = Column_i + Enc(Updated\_value)$ **WHERE** $val_{ij} = Enc_Threshold$ ANDEnc_Uattri;. Here, Enc_Uattri holds the access control information. If both the conditions (UPDATE related functional condition and attribute related access condition) get satisfied, then only row update will take place as shown in Fig. 6. The main advantages of our proposed framework are:

1) User addition and revocation can be easily handled.
2) Constant DBA intervention is not required for query processing. DBA can take part only during the registration phase when any new user joins and leaves the organization, or any change is required in the user permission or access policy. In case of user revocation, such attribute-related changes can be easily handled by metadata table, and it does not affect the overall framework.
3) Incorporation of access control adds a very low overhead in SQL query execution performance (only in terms of single bit-wise encrypted multiplication). Overhead related to communicating and storing the encrypted access table can be considered significant. However, storage is not a very significant concern in cloud domain, and ECC encrypted data communication overhead is also one time during the registration phase.

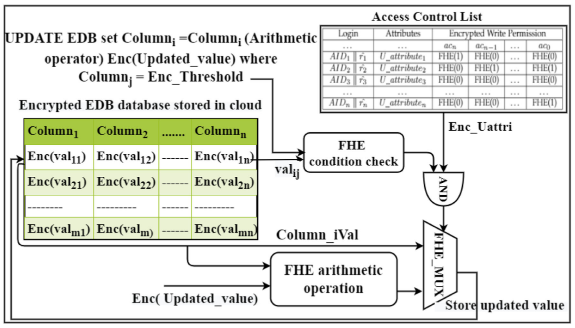

Access Control List
Login
Attributes
Encrypted Write Permission
ac_n
ac_{n-1}
ac_0
$AID_1 \| \hat{r}_1$
$U\_attribute_1$
FHE(1)
FHE(0)
$AID_2 \| \hat{r}_2$
$U\_attribute_2$
$AID_3 \| \hat{r}_3$
$U\_attribute_3$
$AID_n \| \hat{r}_n$
$U\_attribute_n$
UPDATE EDB set Column_i =Column_i (Arithmetic operator) Enc(Updated_value) where Column_j = Enc_Threshold
Encrypted EDB database stored in cloud
Column_1
Column_2
Column_n
$\text{Enc}(\text{val}_{11})$
$\text{Enc}(\text{val}_{12})$
$\text{Enc}(\text{val}_{1n})$
$\text{Enc}(\text{val}_{21})$
$\text{Enc}(\text{val}_{22})$
$\text{Enc}(\text{val}_{2n})$
$\text{Enc}(\text{val}_{m1})$
$\text{Enc}(\text{val}_{m})$
$\text{Enc}(\text{val}_{mn})$
val_{ij}
FHE condition check
Enc_Uattri
AND
Column_iVal
FHE_MUX
FHE arithmetic operation
Enc(Updated_value)
Store updated value

Fig. 6. Conditional Encrypted UPDATE operation with access control.

That does not affect the actual encrypted SQL query processing time.

Algorithm 1. Attribute Verification Algorithm

```
Input: $U\_attri_{received}$, $F\_SQL$
Output: F_SQL query execution
begin
    if ($U\_attri_{received} == U\_attribute$) then
        Query parser analyses the $F\_SQL$ query to distinguish SELECT & UPDATE.
        if (UPDATE query) then
            Step1: Access $Acc_{List}$.
            Step2: Collect encrypted update permission $ac_i$ in $Enc\_Uattri$ for column $C_i$.
            Step3: Sends corresponding $Enc\_Uattri$ to execution block.
        else
            Cloud performs normal execution on FHE SQL SELECT query.
        end
    else
        Deny access request.
    end
end
```

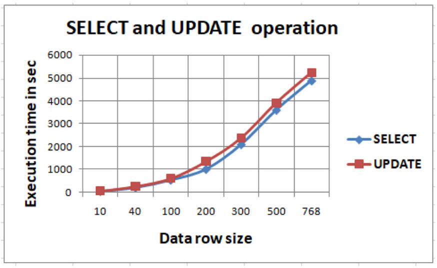
7 Performance Analysis

We have tested our proposed implementation on standard dataset *'Pima-Indians-Diabetes-Database' [28] with 769 rows, and 9 columns*, with each column values are of 16 bits. To measure the timing requirement for SQL UPDATE execution of the whole diabetes dataset, we have conducted experiments on 64 b AMD Ryzen $53550H$ clock speed 2.1 GHz, 8 GB RAM with Ubuntu 16.04.3 LTS from small data size with 10 tuples to large data size with 769 tuples of the dataset. We have ignored elliptic curve cryptography and hash function-related registration cost as they are one-time and does not contribute to each query computational overhead.

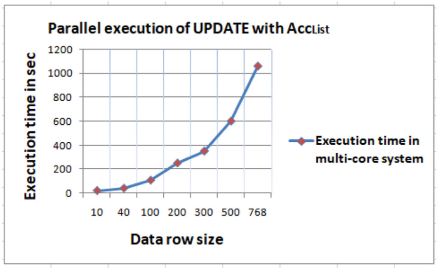
According to our proposed scheme, overhead of multiple conditions handling is minimal. However, execution time increases with the increase in number of database rows (as shown in Figs. 7a and in 7b) or with the increase of bit-size of database elements (as shown in Fig. 7c). Fig. 7a also shows that the execution timing requirement for UPDATE with access control and SELECT (where access control computation is not required) do not differ much, which indicates the access control overhead is minimal.

Due to the huge timing requirement of FHE operations, implementation of the proposed scheme becomes impractical without any aid of parallel processing. Hence, here we try to improve the performance with the support of the multi-processing concept as shown in Fig. 8. As our proposed implementation is row independent, we have taken a straightforward approach to divide the total cloud database rows (m) among the available (k) cores in the multi-core system i.e., $[m/k]$ row-blocks and encrypted SQL UPDATE query with access control can be executed on each copy of $[m/k]$ row-blocks into the multi-core-systems. Here, performance will vary with different values of k. After completing SQL UPDATE operation on $[m/k]$ rows, resultant data-rows from each core will be written back into the original database following consistency. Therefore, the required execution time for entire cloud data will be reduced to almost $[m/k]$ row-block execution, shown in Fig. 7b. We test our proposed framework in a multicore platform on 2$*$Intel Xeon SKL G-6148 with 384 CPU nodes, 40-cores per CPU node, clock speed 2.4 GHz each, and 192 GB RAM per node with Linux – CentOS 7.6 (Paramshakti HPC server: [37]) with dataset 'Pima-Indians-Diabetes-Database' [28]. This HPC platform allows maximum of 40-cores to execute a parallel program using OpenMP [33]. We have taken 8-cores for our experimental purpose. In our dataset, number of rows $m = 769$ and number of cores $k = 8$. According to our proposed solution, before SQL UPDATE processing, entire m rows are distributed into k core-systems after diving into $[m/k]$ row size-blocks. In this experiment, each core receives $[769/8 \approx 96]$ row size-blocks, and 8-cores will execute SQL UPDATE operations in parallel. Fig. 7b shows execution time for parallel processing in 8-core system. In HPC platform, total time to execute the whole dataset is approximately 17 minutes. Suppose, if we use total 40-cores for our experiment, execution time will be decreased to approximately 1 minutes, as each core will execute $[m/k = 19]$ or 20 rows at a time compared to a single processor implementation that requires approximately 87.26 minutes. In this paper, we have elaborated encrypted SQL UPDATE, where access control is an important issue. Our proposed framework is capable of supporting all sorts of encrypted query executions following standard (TPC-C) benchmarks [45] as shown in Table 4. However, encrypted JOIN and GROUP BY operations are costly in terms of performance. Hence, we consider optimized implementations of these encrypted SQL operators as our future direction of research.

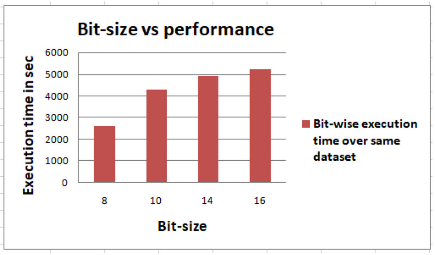
Table 3 mentions few reported encrypted databases in literature, and most of these encrypted databases claim to be better in terms of performance in comparison to FHE encrypted databases. However, we highlight that the previous schemes are either limited in terms of types of supported queries (partial HE-based schemes), prone to reported attacks [25] (deterministic HE schemes), or demand intermediate decryption (ZeroDB, CryptDB, etc.). In Table 3, we compare the efficiency of our proposed scheme with existing proposed methods in literature [14], [15], [26], [39], [44] and [2]. According to Table 3, implementations in [26], and [14] are based on Partial Homomorphic Encryption(PHE) and intermediate decryption is required to handle complex queries. In [44], proposed scheme is secure from data leakage, but in every processing time, data



(a) SELECT & UPDATE query execution Time with access control.



(b) Execution Time of UPDATE query with multiple conditions using multi-core-system.



(c) Bit-wise performance variation of UPDATE query execution

Fig. 7. Query execution time.

owner's intervention is required. Furthermore, CryptDB [39] and Always Encrypted based scheme [2] do not report user revocation process. But CryptDB and ZeroDB [15] require intermediate decryption. Due to deterministic encryption of Always Encrypted scheme, several intermediate attacks [25] are possible. It is clear from Table 3 that our proposed framework is free from such deficits and better in terms of security without any requirement of intermediate decryption and confirms arbitrary levels of homomorphic operations. However, to overcome the inherent performance bottleneck, parallel multi-core execution support can be adapted to achieve near to practical performance. In our future work, we target to improve the performance further with a suitable hardware-software co-processor design.

7.1 Security Analysis

Our proposed framework uses underlying FHE operators and is developed using existing TFHE library [11] and traditional ECC-based login procedure [19]. Randomized FHE scheme defined in TFHE library [10] provides semantic security, where a probabilistic polynomial-time adversary $\widehat{A}$ cannot distinguish two randomly chosen encrypted plaintexts m_0, and m_1 without secret encryption key with negligible advantage (mentioned in the Section 4.2). Hence our scheme is IND-CPA secure. Here, we analyze the security of our scheme against database-related attacks.

Theorem 1. *Assuming randomized FHE scheme IND-CPA secure, outsourced encrypted data processing achieves secured query computation against volume attack, access pattern leakage and statistical leakage based on Definitions 1 and 2 (mentioned in Section 4.2).*

Proof. Consider an adversary $\widehat{A}$, who receives an encrypted predicate value Enc_threshold after *WHERE* condition [i.e., Column $_j$=Enc_threshold] and tries to obtain information keeping track of query history (H_Q) during query transmission. At query processing state inside the cloud, if the query condition is true for certain i-th row i.e., Enc_threshold = val_{ij}, query computation process (π) generates equivalent encrypted row values $E_c[Enc(non\text{-}zero)] \approx (Row\_iVal) \odot Enc(1)$ by Definition 1. In case of unsatisfied condition, generated invalid row with different equivalent ciphertexts of n-bit of encrypted zero i.e., $E_c[Enc(zero)] \approx$ n-bit of Enc(0...) $\odot$ Enc(0) (shown in Fig. 9). In our experiment, we see that Dec($E_c[Enc(non\text{-}zero)]$) = Dec($Row\_iVal$) and Dec ($E_c[Enc(zero)]$) = Dec(n-bit of $Enc(0\ldots)$), where Dec() is decryption function but $E_c[Enc(non\text{-}zero)] \approx (Row\_iVal)$ and $E_c[Enc(zero)] \overset{c}{\approx}$ n-bit of $Enc(0)$. By this observation, our scheme achieves computational security against the following attack.

Security Against Volume Attack. Consider adversary $\widehat{A}$ divides total m database rows into m elementary ranges $[1,1],[1,2],\ldots,[1,m]$ to identify specific volume of query result among these ranges. Then, $\widehat{A}$ transmits each of these range queries Q to our query computation process (π) for getting volume information from RdBdata. Due to presence of valid equivalent row-data ($E_c[Enc\ (non-zero)]$) of Row_iVal and invalid equivalent row-data ($E_c[Enc(zero)]$) of $Enc(0)$ for SELECT operation (and equivalent row-data E_c[Enc(Update_value)] of Enc (Update_value) and equivalent cipher of old column values for UPDATE operation) in resultant database RdBdata, computation process π always generates RdBdata with row size m' which is equal to original database EDB size m i.e [m'= m] after every query processing (shown in Fig. 9). Hence, $\widehat{A}$ is unable to get the information how many rows are exactly affected for a certain query condition [as $\forall$ Q, m'=m].

Security Against Access Pattern Disclose and Statistical Leakage. If adversary $\widehat{A}$ sends a query Q with same textual expression and same encrypted predicates multiple times from its track of H $_Q$=(EDB, l, Id_Q, p_c, RdBdata) to π, every time at the end of query computation, generated resultant RdBdata' $\overset{c}{\approx}$ RdBdata as $E_c[Enc(non-zero)]' \overset{c}{\approx} E_c[Enc(non-zero)]$ and $E_c[Enc(zero)]' \overset{c}{\approx} E_c[Enc(zero)]$ (in case of UPDATE, $E_c[Enc(Update\_value)]'_c \approx E_c[Enc\ (Update\_value)]$ and $E_c[Column\_iVal]' \approx E_c[Column\_iVal]$) for every i th row of resultant dataset by the assumption

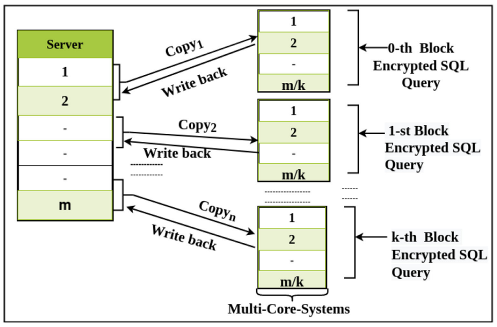

Server
1
2
m
Copy1
Write back
Copy2
Write back
Copyn
Write back
1
2
m/k
0-th Block Encrypted SQL Query
1-st Block Encrypted SQL Query
k-th Block Encrypted SQL Query
Multi-Core-Systems
Fig. 8. Block-wise data distribution for UPDATE query execution.

TABLE 3
Security Comparison of Our Scheme With the Related Schemes

Scheme	Query type	Encrypted scheme	Support attribute revocation	Underlying scheme	Intermediate decryption
Crypt DB [39]	SQL operation of TPC-C benchmark	Crypt DB	X	Partial HE, deterministic	✓
Zero DB [15]	Not Reported TPC-C benchmark	Zero DB	X	Non-deterministic	✓
Microsoft SQL server & Azur SQL database with Always Encrypted [2]	Few SQL operation support TPC-C benchmark	Always Encrypted	X	Deterministic, prone to known attacks [25]	X
Kamlesh Kumar Hingwe et al. [26]	Not Reported TPC-C benchmark	Pailler encryption	Role-based	Partial HE non-deterministic	✓
K. Sethi et al. [44]	Not Reported TPC-C benchmark	FHE	Role-based	Full HE, non-deterministic	Data owner interferance
Yong Ding et al. [14]	Not Reported TPC-C benchmark	CP-ABE and HE	✓	Partial HE, non-deterministic	✓
Boneh et al. [4]	Not Reported TPC-C benchmark	Somewhat HE	X	Additive HE, non-deterministic	X
kim et al. [30]	Not Reported TPC-C benchmark	Leveled FHE	X	non-deterministic	X
Saha et al. [41]	Not Reported TPC-C benchmark	Somewhat HE	X	non-deterministic	X
Our Proposed scheme	SQL operation of TPC-C benchmark	FHE	✓	FHE, non-deterministic	X

TABLE 4
Comparison With Other Schemes Based on Supported Operations

Encrypted Database	combining encrypted query operation	$+,*,\geq,\leq$	Equality comparison	JOIN	ORDER BY	LIKE	UPDATE	GROUP BY
CryptDB	X	✓	✓	✓	Supported but order reveal	✓	✓	✓
ZeroDB	$\longleftarrow$ Processing on client side $\longrightarrow$							X
Always Encrypted (Deterministic)	$\longleftarrow$ Deterministic encryption based processing $\longrightarrow$							
Always Encrypted with secure enclave	$\leftarrow$ Column encryption keys required for above operation within enclave $\longrightarrow$							
Boneh et al. scheme	✓	X	✓	Not reported	Not reported	Not reported	Not reported	Not reported
kim et al. scheme	✓	✓	✓	Not reported	Not reported	Not reported	Not reported	Not reported
Saha et al.	✓	limited Multiplications	✓	Not reported	✓	Not reported	Not reported	✓
Our scheme	✓	✓	✓	✓	✓	✓	✓	✓

of FHE-based scheme IND-CPA secure. Therefore, by statistical analysis of query history H_Q and resultant row-data for a known query, adversary $\widehat{A}$ will not guess any non-public information efficiently because of advantage of $\widehat{A}$ in this game is computed as follows: Pr[$\widehat{A}$(ek, m$_b$, Enc(m$_b$))=1]=e where $b \leftarrow \{0,1\}$ and keyGen(λ)$\longrightarrow$ (pk, sk, ek) [pk, sk, and ek are public, private and evaluation key respectively]; Then advantage $\widehat{A}$=negl(λ) $\geq$ $|\underset{\frown}{e} - \frac{1}{2}|$; For the negligible advantage negl(λ) of adversary $\widehat{A}$ in case of query computation process, our scheme assures security against access pattern leakage and statistical leakage. According to Definitions 1 and 2, for any PPT algorithm neither server nor $\widehat{A}$ will be able to correlate an equivalent cipher with its corresponding ciphertext observing memory address without having authentic encryption key. □

Security Against SQL Injection. Consider a situation, where database EDB=($C_1, \ldots, C_n$) and only values of C_i are encrypted but there is no scope of user input validation and no further restriction on database access. In that case, adversary $\widehat{A}$ is able to inject unauthorized data input (from its query history H_Q) in a UPDATE query statement and successfully modify the database. In our scheme, a user requires two times verification to execute a query - 1) after sending valid login information and 2) before processing SQL query. During the login phase, after receiving login request, cloud computes r' = H(MAC_addr || PW || role)$\oplus \hat{r}$

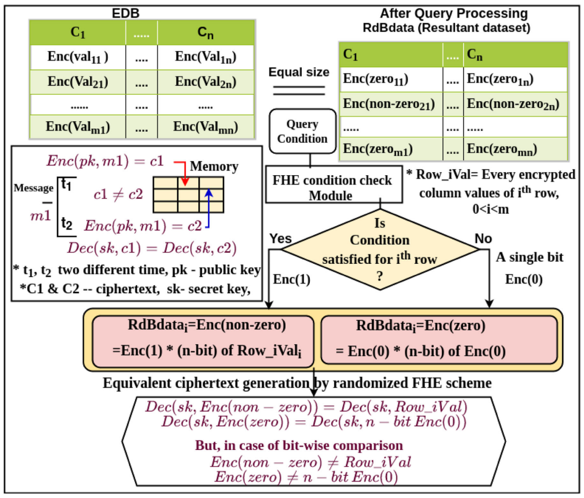

EDB
C1 ..... Cn
Enc(val11) .... Enc(Val1n)
Enc(Val21) .... Enc(Val2n)
Enc(Valm1) .... Enc(Valmn)
Equal size
Query Condition
After Query Processing
RdBdata (Resultant dataset)
Enc(zero11) .... Enc(zero1n)
Enc(non-zero21) .... Enc(non-zero2n)
Enc(zerom1) .... Enc(zeromn)
Enc(pk, m1) = c1
Memory
Message t1
c1 ≠ c2
m1
t2 Enc(pk, m1) = c2
Dec(sk, c1) = Dec(sk, c2)
* t1, t2 two different time, pk - public key
*C1 & C2 -- ciphertext, sk- secret key,
FHE condition check Module
* Row_iVal= Every encrypted column values of ith row, 0<i<m
Is Condition satisfied for ith row ?
Yes
No
Enc(1)
A single bit Enc(0)
RdBdatai=Enc(non-zero) =Enc(1) * (n-bit) of Row_iVali
RdBdatai=Enc(zero) = Enc(0) * (n-bit) of Enc(0)
Equivalent ciphertext generation by randomized FHE scheme
Dec(sk, Enc(non − zero)) = Dec(sk, Row_iVal)
Dec(sk, Enc(zero)) = Dec(sk, n − bit Enc(0))
But, in case of bit-wise comparison
Enc(non − zero) ≠ Row_iVal
Enc(zero) ≠ n − bit Enc(0)

Fig. 9. Equivalent ciphertext generation.

and AID′ = H(H(PW ∥ r′) ∥ H(UID ⊕ r′)).P and checks AID′$\overset{?}{=}$AID. If any of the received information from user (such as UID, PW, MAC_addr, and role) are mismatched, login request is failed. After successful login, login user sends a FHE SQL query with timestamp encrypted by Q_{pub1} to cloud. According to our assumption, UPDATE query without condition is not acceptable and that condition is FHE encrypted. Due to second times user verification followed by (U_attri $_{received}$) $\overset{?}{=}$ U_attribute$_i$ in the cloud and FHE encrypted SQL conditions, our scheme avoids unauthorized input injection.

Security Against Spoofing Attack and Insider Attack. An attacker can pretend to be a legitimate user knowing any valid userid-password pair and send database access requests to the cloud server. In an insider attack scenario, an organization's unauthorized employees can also obtain userid-password pair; and try to access the database. In both the cases, after receiving login request, cloud computes r′ = H(MAC_addr ∥ PW ∥ role) ⊕ $\hat{r}$, AID′ = H(H(PW ∥ r′) ∥ H(UID ⊕ r′)).P and verifies AID′$\overset{?}{=}$AID. In this case, the login device's MAC address(MAC_addr) is important, which coming with the login request. If MAC address is different, login request will not be successful. After receiving encrypted query Q $_{pub1}$.(F_SQL ∥ T$_u$) from logged user, cloud again computes (U_attri$_{received}$) = [H(UID$_{received}$ ⊕ r′)∥ H(MAC_addr $_{received}$ ∥ place$_{received}$ ∥ role ∥ r′)].P and verify whether (U_attri $_{received}$) $\overset{?}{=}$ U_attribute $_i$. If this condition is true and query is a SELECT query, query computation process π generates RdBdata. If this condition is true and query is a UPDATE query, user needs to meet the encrypted access control list (Acc$_{List}$) to resist unauthorized data modification activity from insider and outsider adversary $\widehat{A}$. If update permission Enc_Uattri ≠ FHE(1) or encrypted predicate Enc_Threshold in query condition is not matched with any val$_{ij}$ of given j th column, EDB will not be modified. So, our scheme can prevent server spoofing attacks or insider attacks, assuming that legitimate users' login devices and userid-password pairs are not compromised together.

Security Against Replay Attack. After successful login, cloud gets a query (F_SQL) concatenated with transmitting timestamp (T $_u$) i.e Q$_{pub1}$.(F_SQL ∥ T$_u$) from the logged user at a time T$_c$. If [T$_c$ - T$_u$] ≤ △ T [where, △T is an expected time delay defined by DBA], F_SQL will be accepted. Otherwise, cloud will discard the user query. This timestamp helps to prevent replay attacks in our scheme. In this scheme, users' information is encrypted by ECC form and stored with hashing technique in the cloud to secure the authentication process. Moreover, multiple user attributes have been used for validating the user authenticity; if any of these attributes get compromised, our framework is still be secured against malicious activity of any adversary until all attributes together are getting compromised. To achieve better security requirement, EDB column name (C_i) can be kept in hash form in our proposed framework. We need to include a hash equality check during column matching to verify the condition. Our framework assures 88-bits security level for quantum model and more than 128-bits security level for a traditional system because of TFHE library [11]. For the login procedure part based on ECC security assumption, it is true that this scheme in the given format is not secure considering post quantum assumptions. However, this part can be replaced by isogeny-based key exchange protocol like SIKE [12]. Since, usage of this ECC concept is only in the login part, it will not affect the performance of actual encrypted processing part.

8 Conclusion

In this work, we propose a FHE-based framework for encrypted database designing which assures secure SQL processing with an efficient access control policy. Our proposed solution prevents not only external attacks but also resists illegal write-back options by internal users. The proposed design is based on randomized encryption. Hence, it is free from several attack possibilities compared to existing encrypted databases in literature. Moreover, it supports multiple encrypted conditions checking in access control-based SQL query processing with minimum operational overhead. In this work, it is essential to note that we have mainly focused on incorporating access control techniques and FHE query processing in the same framework. Efficient selection of attributes for different use cases, efficient storage of metadata table, and other techniques of performance improvement will be explored in our future work.

References

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–35, 2018.
[2] P. Antonopoulos et al., "Azure SQL database always encrypted," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1511–1525.
[3] J. Benaloh, "Dense probabilistic encryption," in *Proc. Workshop Sel. Areas Cryptogr.*, 1994, pp. 120–128.
[4] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, "Private database queries using somewhat homomorphic encryption," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2013, pp. 102–118.
[5] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Proc. Theory Cryptogr. Conf.*, 2005, pp. 325–341.
[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, 2014.
[7] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-LWE and security for key dependent messages," in *Proc. Annu. Cryptol. Conf.*, 2011, pp. 505–524.
[8] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014.

[9] A. Chatterjee and I. Sengupta, "Translating algorithms to handle fully homomorphic encrypted data on the cloud," *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, pp. 287–300, First Quarter 2018.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2016, pp. 3–33.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[12] L. De Feo, "Mathematics of isogeny based cryptography," Dec. 2017, *arXiv:1711.04062*.

[13] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, "Cloud computing: Distributed internet computing for it and scientific research," *IEEE Internet Comput.*, vol. 13, no. 5, pp. 10–13, Sep./Oct. 2009.

[14] Y. Ding and X. Li, "Policy based on homomorphic encryption and retrieval scheme in cloud computing," in *Proc. IEEE Int. Conf. Comput. Sci. Eng. Int. Conf. Embedded Ubiquitous Comput.*, 2017, pp. 568–571.

[15] M. Egorov and M. Wilkison, "ZeroDB white paper," 2016, *arXiv:1602.07168*.

[16] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. IT-31, no. 4, pp. 469–472, Jul. 1985.

[17] Always encrypted. Accessed: Jan. 5, 2021. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-ver15

[18] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 196–205.

[19] R. for discrete logarithm-based cryptography: Elliptic curve domain parameters. Accessed: Feb. 19, 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800–186-draft.pdf

[20] C. Gentry, *A Fully Homomorphic Encryption Scheme*. Stanford, CA, USA: Stanford Univ., 2009.

[21] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2011, pp. 129–148.

[22] S. Goldwasser and S. Micali, "Probabilistic encryption and how to play mental poker, keeping secret all partial information," in *Proc. 14th Annu. ACM Symp. Theory Comput.*, 1982, pp. 365–377.

[23] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proc. 13th ACM Conf. Comput. Commun. Secur.*, 2006, pp. 89–98.

[24] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov, "Breaking web applications built on top of encrypted data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1353–1364.

[25] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why your encrypted database is not secure," in *Proc. 16th Workshop Hot Topics Operating Syst.*, 2017, pp. 162–168.

[26] K. K. Hingwe and S. M. S. Bhanu, "Hierarchical role-based access control with homomorphic encryption for database as a service," in *Proc. Int. Conf. ICT Sustain. Develop.*, 2016, pp. 437–448.

[27] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Proc. Int. Algorithmic Number Theory Symp.*, 1998, pp. 267–288.

[28] Pima indians diabetes database. Accessed Jan. 20, 2021. [Online]. Available: https://www.kaggle.com/uciml/pima-indians-diabetes-database

[29] A. A. E. Kalam et al., "Organization based access control," in *Proc. IEEE 4th Int. Workshop Policies Distrib. Syst. Netw.*, 2003, pp. 120–131.

[30] M. Kim, H. T. Lee, S. Ling, and H. Wang, "On the efficiency of FHE-based private queries," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 2, pp. 357–363, Mar./Apr. 2018.

[31] N. Kobliz, "Elliptic curve cryptography," *Math. Comput.*, vol. 48, pp. 203–209, 1987.

[32] X. Meng, H. Zhu, and G. Kollios, "Top-k query processing on encrypted databases with strong security guarantees," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 353–364.

[33] OpenMP. Accessed: Jan. 5, 2021. [Online]. Available: http://www.cse.iitm.ac.in/krishna/courses/2018/even-cs6868/openmp-2.pdf

[34] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 1999, pp. 223–238.

[35] A. Papadimitriou et al., "Big data analytics over encrypted datasets with seabed," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 587–602.

[36] T. Parbat and A. Chatterjee, "Encrypted SQL arithmetic functions processing for secure cloud database," in *Proc. 11th Int. Conf. Secur. Privacy Appl. Cryptography Eng.*, 2021, pp. 207–225.

[37] High performance computing (HPC). Accessed: Feb. 19, 2021. [Online]. Available: http://www.hpc.iitkgp.ac.in/

[38] S. A. Pitchay, W. A. A. Alhiagem, F. Ridzuan, and M. M. Saudi, "A proposed system concept on enhancing the encryption and decryption method for cloud computing," in *Proc. 17th UKSim-AMSS Int. Conf. Modelling Simul.*, 2015, pp. 201–205.

[39] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 85–100.

[40] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[41] T. K. Saha, M. Rathee, and T. Koshiba, "Efficient private database queries using ring-LWE somewhat homomorphic encryption," *J. Inf. Secur. Appl.*, vol. 49, 2019, Art. no. 102406.

[42] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2005, pp. 457–473.

[43] P. Scholl and N. P. Smart, "Improved key generation for gentry's fully homomorphic encryption scheme," in *Proc. IMA Int. Conf. Cryptogr. Coding*, 2011, pp. 10–22.

[44] K. Sethi, A. Chopra, P. Bera, and B. K. Tripathy, "Integration of role based access control with homomorphic cryptosystem for secure and controlled access of data in cloud," in *Proc. 10th Int. Conf. Secur. Inf. Netw.*, 2017, pp. 194–199.

[45] TPC-C. Accessed: Jan. 11, 2021. [Online]. Available: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[46] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2010, pp. 24–43.

[47] L. Zhou, V. Varadharajan, and M. Hitchens, "Achieving secure role-based access control on encrypted data in cloud storage," *IEEE Trans. Inf. Forensics Secur.*, vol. 8, no. 12, pp. 1947–1960, Dec. 2013.

Tanusree Parbat (Member, IEEE) received the BTech degree from WBUT and MTech degree from Jadavpur University, India, and is currently working toward the PhD degree in secure cloud database under homomorphic domain from Advanced Technology Development Centre, IIT Kharagpur, India. Her research interests include secure cloud computing.

Ayantika Chatterjee (Member, IEEE) is currently an Assistant Professor with Advanced Technology Development Centre, IIT Kharagpur, India. Her research interests include cryptography and embedded security.