Received April 6, 2021, accepted May 5, 2021, date of publication May 20, 2021, date of current version June 4, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3082139

An Efficient Secure System for Fetching Data From the Outsourced Encrypted Databases

SULTAN ALMAKDI[1], BRAJENDRA PANDA[2], (Senior Member, IEEE), MOHAMMED S. ALSHEHRI[1], (Graduate Student Member, IEEE), AND ABDULWAHAB ALAZEB[1,2], (Graduate Student Member, IEEE)

[1]Department of Computer Science, College of Computer Science and Information systems, Najran University, Najran 55461, Saudi Arabia
[2]Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR 72701, USA

Corresponding author: Sultan Almakdi (saalmakdi@nu.edu.sa)

ABSTRACT Recently, database users have begun to use cloud database services to outsource their databases. The reason for this is the high computation speed and the huge storage capacity that cloud owners provide at low prices. However, despite the attractiveness of the cloud computing environment to database users, privacy issues remain a cause for concern for database owners since data access is out of their control. Encryption is the only way of assuaging users' fears surrounding data privacy, but executing Structured Query Language (SQL) queries over encrypted data is a challenging task, especially if the data are encrypted by a randomized encryption algorithm. Many researchers have addressed the privacy issues by encrypting the data using deterministic, onion layer, or homomorphic encryption. Nevertheless, even with these systems, the encrypted data can still be subjected to attack. In this research, we first propose an indexing scheme to encode the original table's tuples into bit vectors (BVs) prior to the encryption. The resulting index is then used to narrow the range of retrieved encrypted records from the cloud to a small set of records that are candidates for the user's query. Based on the indexing scheme, we then design a system to execute SQL queries over the encrypted data. The data are encrypted by a single randomized encryption algorithm, namely the Advanced Encryption Standard-Cipher-Block Chaining (AES-CBC). In the proposed scheme, we store the index values (BVs) at user's side, and we extend the system to support most of relational algebra operators, such as select, join, etc. Implementation and evaluation of the proposed system reveals that it is practical and efficient at reducing both the computation and space overhead when compared with state-of-the-art systems like CryptDB.

INDEX TERMS Cybersecurity, privacy-preserving, encrypted databases, SQL queries, cloud computing.

I. INTRODUCTION

In the contemporary electronic era, both individuals and organizations need scalable data storage and high-performance computing units to process and store their data. Historically, only large organizations/companies have been able to own such units, as they were not affordable for most individuals and small companies. With the rise of cloud computing, however, this problem has been solved, as users can now rent storage and computational units as needed at an affordable price. Most cloud providers provide databases as a service, which allow individual users and companies to outsource their data and access them at any time, from any location. According to the report in [1], the compound annual growth rate (CAGR) of cloud database market is anticipated to be 46.78% in 2023, Figure.1. However, given that privacy breaches are one of the most common threats in the cloud computing environment, many people have expressed concerns about privacy when outsourcing sensitive data. For instance, untrustworthy cloud service providers might steal personal customer information, such as email addresses, mailing addresses, and phone numbers, and sell that information to third parties, who can then use it to send irritating advertisements to users via email, mail, and telephone.

More importantly, attackers who target a cloud provider can gain access to customers' sensitive personal information, such as social security numbers (SSNs). This has serious consequences, as criminals can use these data to impersonate

The associate editor coordinating the review of this manuscript and approving it for publication was Fan-Hsun Tseng.

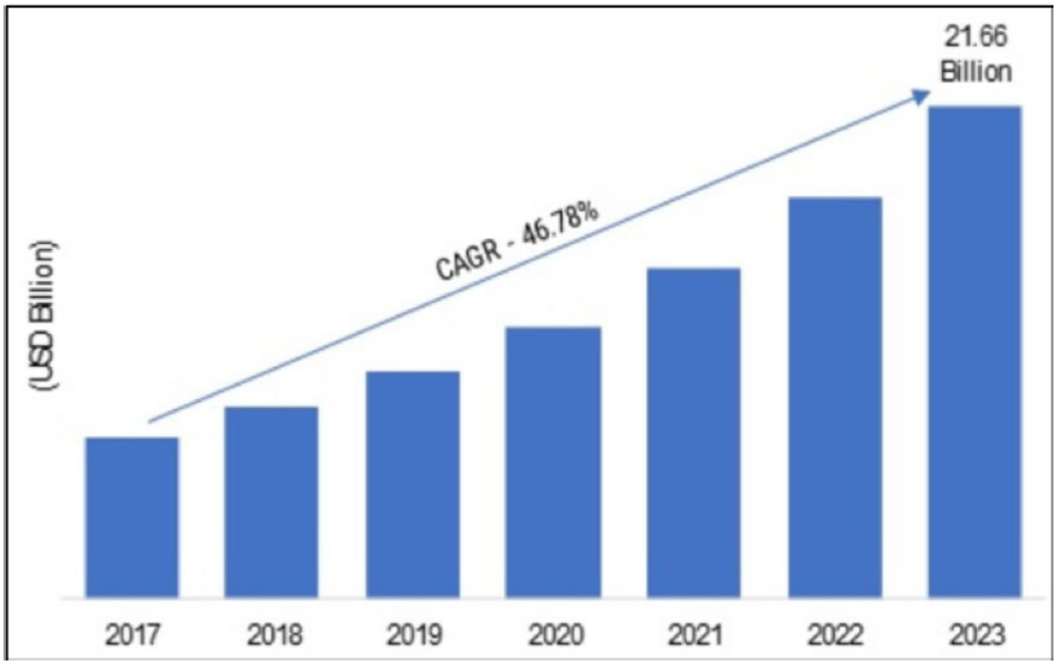


FIGURE 1. Anticipated market growth in 2023. Source: https://www.marketresearchfuture.com/reports/cloud-database-market-6847.

customers in situations such as financial transactions (e.g., telephone banking). Thus, sensitive data are restricted from being processed or sold to a third party. Therefore, significant evolution in the cloud computing environment could make such services unattractive to consumers if changes occur without also providing appropriate solutions for privacy breach issue. Such an issue must be tackled if cloud providers are to gain the trust of users and organizations so that they will outsource sensitive data without worrying about data leakages.

Data encryption effectively solves the problem of privacy breaches by ensuring that cloud providers cannot learn from the data they store. The easiest way is to encrypt the entire table and outsource it. Then to process a query, the entire table must be retrieved and decrypted. However, the application of this technique conflicts with purpose of the databases and the critical functionalities of cloud environments (e.g., searching). Other researchers have used a proxy (i.e., a trusted third-party server), rather than the user, as an additional component to carry out the encryption and decryption processes. To make this approach practical, each datum must be encrypted with more than one encryption algorithm to support various query types [2]. However, in the case of very large data sets, this approach comes with the penalty of a significant computational burden, as each datum might be decrypted more than once. Various researchers have proposed numerous systems using different encryption techniques to protect data confidentiality. In the following subsection, we explore the common encryption algorithms that have been adopted in state-of-the-art research on cloud database security.

A. ENCRYPTION ALGORITHMS

1) ORDER-PRESERVING ENCRYPTION

Order-preserving encryption (OPE) is a functional encryption technique to encrypt data such that range queries (e.g., maximum, minimum, and inequality operators) can be implemented on encrypted data without encrypting the operands or decrypting the data [3], [4]. This type of encryption uses a function to compare the order of the ciphertexts to allow comparison operations of the encrypted numeric data. This type of algorithm preserves the original data order. For example, if c_1 is the ciphertext of m_1, and c_2 is the ciphertext of m_2, then the comparison of c_1 and c_2 is as follows: $(c_1 < c_2 if m_1 < m_2)$; $(c_1 > c_2 if m_1 > m_2)$; $or (c_1 = c_2 if m_1 = m_2)$

The security of this type of encryption is downgraded if the adversary infers the ciphertext of a certain plaintext. Also, this type of encryption is vulnerable to inference attacks as in [5], [6].

2) DETERMINISTIC ENCRYPTION

In deterministic encryption (DE) schemes, given the encryption key k and the messages m, the ciphertext of m is always the same when encrypted with k, even in multiple executions of DE. While DE can be used in keyword searches, it does not preserve the order if used with numeric values. AES-SIV is an example of a deterministic algorithm. DE is widely used in securing cloud databases. However, privacy can be compromised in this scheme if the attacker is able to identify the ciphertext of a certain plaintext word (e.g., if he establishes the ciphertext of "Alice," then he is able to determine which tuples contain "Alice").

3) HOMOMORPHIC ENCRYPTION

Homomorphic encryption (HOM) is a method of encryption that allows for the performance of certain arithmetic operations (i.e., addition and multiplication) on ciphertexts without the need to decrypt them. There are two types of HOM: partially HOM and fully HOM. Partially HOM supports either addition or multiplication, while fully HOM supports both operations. HOM can secure numeric data in cloud environments; however, encrypting with HOM produces long ciphertexts, since this type of encryption is based on an asymmetric encryption system. Also, computation performance on HOM ciphertexts is downgraded as the volume of ciphertexts increases [7].

4) RANDOMIZED ENCRYPTION

Randomized encryption intends to produce different ciphertexts for each plaintext, i.e., no more than one ciphertext has the same plaintext. Essentially, in this scheme, in addition to the secret key, an initialization vector (IV) must be XORed with the first block of the plaintext, and a new ciphertext must therefore be obtained each time the algorithm is executed [8]. While this encryption technique provides the highest security level for outsourced databases, it has a major drawback in cloud databases,namely the inability to execute SQL queries over ciphertexts.

5) ONION ENCRYPTION

The term ''onion layers encryption'' was first developed by the authors in [2]. In onion encryption, cloud servers are able to execute different SQL statements while data remain secret. In the onion encryption, each layer is a ciphertext of a specific encryption methodology (e.g., DET, OPE, HOM, or RAN). The inner layer is the ciphertext of the algorithm with the lowest security level, while the outer layer is the ciphertext with the highest security level (i.e., randomized). The main flaw of this approach is the intensive computation that results from decrypting all of the encryption layers, which leads to slower query processing. In addition, the space required for the encrypted databases is about 3.75 times that required for the unencrypted databases [2].

B. SUMMARY OF CONTRIBUTIONS

In this research, we first designed a novel indexing scheme for randomized encrypted databases, which is based on defining a partitioning tree (PT) for all domain partitions for sensitive columns in a table (i.e., dividing each column into sub-columns where each sub-column represents a set of values). The PT is then used to encode records into bit vectors (BVs), wherein each bit position is mapped to a specific partition and is only set to one if the value belongs to the set of values represented by that partition. The BVs are used to retrieve only part of the outsourced encrypted records. Second, we developed a secure model based on our proposed indexing technique. In this model, we use store and handling the BVs on a private cloud server. Third, we proposed different algorithms to process most of the relational algebra operators on encrypted data without revealing the confidentiality of data. Fourth, we conducted several experiments to evaluate different aspects of the proposed system including computation overhead and space overhead against three well-known approaches: CryptDB [2], columns-based fragmentation [9], and one block-based encryption [10]. Our experiments showed that the proposed system outperformed most of the competing approaches in both time and space overhead.

The rest of this document is organized as follow: in section II, we explain different security enforcement schemes for the cloud databases, then we survey the state-of–the-art approaches to secure outsourced databases. Section III details the proposed system, followed by the implementation and evaluation details in section IV. Finally, we provided a conclusion of this research in section V.

II. RELATED WORK

The powerful features of the cloud computing environment, such as enormous capacity and high-performance computing units, have attracted database owners in both small and large companies. These features have made it necessary to obtain and maintain the incredibly expensive storage and computation units to process vast databases at affordable prices. In spite of the advantages of cloud computing, data security is the main drawback of using cloud services to outsource databases. The work of securing databases began when networking and internet technology advanced and were adopted by major companies. This, in turn, fueled the need for databases as a service for individuals and entities requiring high storage capacity and computation. Data privacy, integrity, and confidentiality are in danger when databases are outsourced, since the database owner loses control over who can access and read their data.

To address these issues, researchers began to develop various approaches to protect user data from unauthorized access and data breaches, using data access control, data encryption, or both [11], [12]. Data access control functions by regulating what object O a subject S can access and what operations Op that S can perform on O. Encryption, in contrast, works by encoding plaintext data into ciphertext (i.e., an unreadable format). Each method has its advantages and disadvantages. For example, access control is intended to increase computing performance, while encryption decreases it. In addition, encryption protects against data breaches from both internal and external attacks, whereas data privacy can be compromised in such cases when the access control mechanism is enforced, since adversaries might bypass predefined access roles to access data. Regardless of the heavy computational work required by encryption schemes, encryption is still the preferred security mechanism for most database users.

1) SECURITY ISSUES IN CLOUD COMPUTING

Despite the powerful features of cloud computing, there are many issues and vulnerabilities that can be exploited by malicious actors against outsourced data. One type of security issues, privileges abuse, involves the use of legitimate privileges for malicious purposes (e.g., a user in company A with the privilege to view company A's sales records who uses their privileges to fetch sales tuples and pass them to competitor company B). Another vulnerability in the cloud environment occurs when a user is assigned privileges that exceed what is necessary to perform their job. This can create a potential threat if such privileges are abused, either by the user or an attacker compromising the user's account. Cloud environments may also be vulnerable to SQL injection, i.e., injecting SQL queries to act against the objective of an application. The injected SQL statement is inserted into an executable statement which, in turn, can fetch data that the attacker wants to obtain (e.g., injecting a query to retrieve all of the records in a given table). Malicious insider attacks are a form of attacks that are performed from inside the cloud organization, with very little chance of detection. The attacker can access sensitive data and leak or maliciously process them in a way that violates system policies.

A data breach is defined as the accessing or obtaining of sensitive information—such as medical records, student information, employees' salaries, and so on—in an unauthorized manner. Such problems occur when data access is not restricted and when API access control is weak. Other cloud attacks may exploit the following: weak authentication; unpatched services; or insecure system architecture (e.g., keeping sensitive and non-sensitive data in the same database without implementing any form of encryption for the sensitive data).

One solution to concerns about the above-mentioned vulnerabilities is applying data encryption to the sensitive data. For example, in an SQL injection attack, if the attacker injects a query to fetch the entire set of the database, the attacker will learn nothing if the sensitive data are encrypted. However, security level differs by the type of encryption used. Weak encryption techniques can be compromised by cryptographic attacks, which exploit a vulnerability in the cryptographic, such as a weakness in a cipher, key management scheme, code, or cryptographic protocol. In this dissertation, we focus on encryption strategy as a method of protecting data from the cloud database vulnerabilities listed above. In the next subsection, we survey the most popular encryption schemes that have been used by state-of-the-art database security systems in the field of cloud computing.

2) ENCRYPTION-BASED SOLUTIONS TO PROTECT CLOUD DATABASES

Encryption is defined as encoding plaintext into unreadable formats, which preserves the confidentiality and privacy of the data. There are two types of encryptions: symmetric and asymmetric. In symmetric encryption, only one encryption key (i.e., the secret key *sk*) is used to encrypt and decrypt the data. The *sk* is known by both the encryptor and decryptor entities; however, it must be kept secret. The most popular symmetric encryption algorithms are the Advanced Encryption Standard (AES) and Blowfish algorithms. In [13], the authors conducted a comparative analysis of the AES and Blowfish algorithms and found that AES was faster than Blowfish by nearly 200 ms when used to encrypt or decrypt the same file. Asymmetric encryption, on the other hand, involves using two keys—a public key and a private key—to encrypt and decrypt the data. To guarantee confidentiality, the receiver's public key is used to encrypt the data, while the private key, associated with the receiver's public key, is the only key used in the decryption process and must be kept secret. Asymmetric encryption systems are computationally intensive and slow down the encryption and decryption processes [14]. Some examples of well-known asymmetric encryption systems include RSA, ElGamal, Diffie-Hellman, and ECC. For more details about asymmetric encryption algorithms, see [15], [16] and [17].

When choosing an encryption system, a variety of factors must be considered, including security level, computation overhead, and complexity, among others. For instance, the computation overhead of asymmetric encryption is higher than that of symmetric encryption because, in an asymmetric system, the usage of CPU cycles is higher than in a symmetric scheme. Moreover, the security level in both systems differs based on the type of algorithm used (e.g., randomized algorithms are more secure than deterministic algorithms) and whether a strong *sk* was used to encrypt the data. In a symmetric system, the security level is high, and the chance of compromising the *sk* (for, e.g., a *sk* length of 256 bits in AES) is virtually nonexistent. Accordingly, symmetric algorithms are more widely used than asymmetric algorithms.

Data encryption is the only solution for protecting outsourced databases that prevents data leakage resulting from any form of unauthorized data access in the cloud. However, it is challenging to execute SQL queries over encrypted databases. The existing literature on cloud database security offers a variety of techniques to overcome this problem and deal with outsourced encrypted databases. While most state-of-the-art systems aim to provide security, efficiency (i.e., time required to execute SQL queries) varies depending on the technique used; the higher security level provided, the lower performance level achieved. In the following subsection, we explore each strategy used to deal with outsourced encrypted databases. We then review the research offering solutions within each strategy.

3) CURRENT APPROACHES TO PROCESS SQL QUERIES OVER ENCRYPTED DATABASES

Early attempts to secure databases encrypted the whole database, with each record encrypted as one block (e.g., if a table has four sensitive columns before encryption, the encrypted table will have just one column to store the encrypted values). However, as mentioned earlier, problems occur when SQL queries must be executed over the encrypted data. The simplest solution to this matter is to fetch the entire outsourced table and decrypt it, then execute the query. While this approach can work well for small databases, it suffers from higher computation costs when applied to larger databases (e.g., a table holding millions of records).

Researchers have proposed numerous solutions to avoid retrieving entire outsourced encrypted databases by classifying records into categories before proceeding to the encryption process. The encrypted table will have an additional column(s) to store the category of the record. By using these categories, the end-user is able to retrieve only part of the encrypted data. In addition, the amount of fetched encrypted tuples is impacted by how data are categorized in the cloud (i.e., the more data categories there are, the fewer data are fetched). To address the issue of categorizing data, many authors (e.g., [18], [19]) have proposed approaches to dividing attributes into categories that can be used to query encrypted data. The techniques are based on dividing each attribute into ranges. The main encrypted table in the cloud will then have additional attributes—as many as the number of partitions among all attributes—to hold numeric values. The whole record will then be encrypted as one block and stored as an attribute value in the cloud. The early attempts technique was developed by [10], who proposed different techniques to execute relational algebra operators over the encrypted records. Their solution assigned an identifier for each value in the tuple, then used those identifiers to retrieve only encrypted records whose identifiers matched the requested identifier. Nevertheless, there are several limitations to using such an approach, including vulnerability to statistical attacks, as mentioned in [20], and heavy client-side computation due to decrypting every retrieved record's data (because all the fields of each record are encrypted as one block). The authors in [21] proposed a system to build an index for the plain data, then encrypt each page of the index individually. To execute a query, the corresponding page is loaded and decrypted. However, since all pages are encrypted with one key, the security of this scheme is downgraded. In addition, the size of the index will continue to grow, which could impact performance. To improve the security level, a unique encryption key could be used to encrypt each page of the index. In [20], the researchers suggested building a B-tree index, maintained on the client-side, over the plaintext data. In [22], the authors introduced a single values level encrypted index and suggested splitting the index into sub-indexes, i.e., each sub-index is for encrypted values using the same key in the column. The authors in [23] proposed a none–order-preserving index for the encrypted database. This index does not require interaction with the user once the query is submitted. The security of this scheme is higher than that of models based on order-preserving indexes, which may be vulnerable to statistical attacks. Hahn *et al.* [24] propose a system to join encrypted databases. The idea is based on applying a selection operation first, then enforce the join over selected data. That only leaks the frequency of use and access patterns. This method is interesting; however, the delay is high because of the asymmetric cryptosystem they use.

In [2], Popa *et al.* developed CryptDB as the first practical system for executing Standard Query Language (SQL) queries over encrypted databases. Two attack scenarios were addressed using onion layers encryption: cloud attack and proxy attack. Each datum is encrypted by more than one encryption algorithm in which the outer layer ciphertexts produced by a randomized encryption algorithm. CryptDB uses a proxy to perform the crypto operations for the user. One of the drawbacks of CryptDB is that, because of the excessive crypto operations and many layers of decryption, it introduces a high computational burden. In addition, because it is challenging to execute an analytical load to encrypted data on a server, CryptDB was improved in [25] to support complex queries and large data sets. MONOMI solves this problem by splitting the execution into two sets: a set of queries to outsourced encrypted data and a set to be executed on decrypted data on the user's side. Authors in [26] proposed an enhanced version of CryptDB to accelerate query processing. Instead of using AES, they used AES-NI, which was reflected in the speed of the query processing time. They also suggested improvements to the hardware to accelerate query processing in CryptDB. There are many different systems proposed on top of CryptDB, such as the one presented in [27].

Liu *et al.* [28] proposed a fully homomorphic order-preserving encryption system (FHOPE) to execute complex SQL queries over encrypted numeric data. This system allows cloud providers to run arithmetic and comparison operators over encrypted data without repeating the encryption, thus helping to resist homomorphic order-preserving attacks. The downside of that study is that the authors conducted their experiments using tables with less than 9,000 records. For improved measurement of the efficiency and scalability of this system, the tables should have more records (e.g., 100,000 or more). A variety of studies related to this system are provided in references [29], [30].

Cui *et al.* proposed P-McDb [31], a privacy-preserving search approach that allows users to execute queries over encrypted data. To avoid inference attack, this system requires two cloud servers, one for database re-randomizing and shuffling and one for data storing and searching. Instead of a total search, P-McDb supports partial searches of encrypted records that are described as a sub-linear manner. Further, P-McDb is a multi-user system. In the case of a user revocation, the data cannot be re-encrypted. Another limitation of this system is that communication with two cloud providers will add more latency when compared to other systems such as those described in [2]. More proposed systems related to P-McDb are described in references [9], and [32], [33].

Osama *et al.*, in [34], proposed different approaches for partitioning attributes of tables into multiple sub-columns based on the attribute's domain values. The methods were tested and introduced various delays. They use an order-preserving mapping function, which enables cloud servers to run different types of SQL-queries. The major disadvantage of this research is that only attributes with numeric values but not with string values were considered. Moreover, such a system only supports select statements.

In [35], the researchers proposed a secure database (SDB) approach, a system that divides data into sensitive and non-sensitive, with only sensitive data being encrypted. The initial idea was to split the sensitive data into two shares. The data owner (DO) keeps one share, and the second share is kept by the cloud service provider (CSP). Assuming the CSP is *curious*, the CSP can learn nothing from its share unless it obtains the DO's share. Also, the SDB allows different operators to share the same encryption, thus providing secure query processing with data interoperability. Similar studies can be found in [25], [36], and [37].

As presented in [38], some researchers used a technique called "Bucketization," in which the tuples are mapped to more than one bucket. This technique enables a "database as a service" (DAS) server to execute SQL-style queries over encrypted data. Each bucket contains a set of encrypted records ranging from the minimum to maximum value and assigned an identification (ID). Several studies based on this approach have been conducted [39], [40].

Some researchers [31], [41], and [42] have addressed cloud database privacy by adopting what is called a hybrid cloud. The technique is based on dividing data into sensitive and non-sensitive. Then, the sensitive data or attributes are outsourced to the user's private cloud while the non-sensitive data are migrated to the public cloud. The problem is that, because most users consider their data to be sensitive, this scheme is not practical for users of non-sensitive data. Also, the complexity of integration for this solution is high.

On the other hand, Amjad *et al.* [9] proposed a technique to prevent untrusted and suspicious cloud service providers from being able to learn from private data. This technique is based on vertical fragmentation, in which each sensitive encrypted column is outsourced to a different cloud server (slave cloud). In contrast, while the whole encrypted table is stored at the central server (master cloud). Because the encryption algorithms [2] and the proxy were used in this system, the proxy performed all the work of interpreting queries, encryption, and decryption. One of the limitations of this work is more communication delays, especially if the query condition contains more than one clause. Another example of research that uses this technique is [43].

Bouganim *et al.* [44] introduced a hardware/software system to address the problem of confidentiality leakage in the outsourced databases. The idea is that the user maintains and controls a mediator smartcard that is plugged in on the side. This smartcard is responsible for encrypting the data before putting them into the database and decrypting data before sending them to the user. The major disadvantage of this technique is that the user is limited by the capacity of the smartcard and cannot benefit from the storage provided by the cloud services. Similar studies can be found in [44], [45].

SafeBox [46] is a system based on an approach called access security broker (CASB). This approach allows users to search and share encrypted data while protecting sensitive information from being leaked if an attacker gains access to the cloud server (CS). This technique can be applied over encrypted databases or files and supports keyword-based searches within the encrypted contents. Several studies that use CASB can be found in [47]. Also, a detailed survey about the use of brokers in the Cloud can be found in [48].

III. METHODOLOGY

A. INTRODUCTION

The primary goal of this research, as stated in our previous work [49], [50] is to address the significant drawbacks of some of the state-of-the-art research in the field of cloud database security. They are described below.

Onion layers encryption means encrypting each datum using different encryption algorithms. The inner layer is the ciphertext of the algorithm with the lowest security level, while the outer layer is the ciphertext of the algorithm with the highest security level (i.e., randomized encryption algorithm) [2]. When it comes to search for a value, the whole column's values must be updated to the next layer (take off layers) This process might be executed more than once to achieve the desired result. For a large encrypted table, more excessive crypto operations are performed, leading to substantial computational overhead. To enable the cloud server to remove and adjust layers, the secret key is passed to the server, making the system vulnerable to an in-session attack. Also, the trusted but curious cloud provider(s) could learn about the data if the security layer is adjusted to a low-security level layer. To overcome this limitation, our proposed approach was designed to encrypt each datum in the table using only a randomized encryption algorithm (AES-CBC). Also, we fetch only those encrypted rows from the cloud server that are related to the query of the user, which reduces undesirable computations. We eliminate passing the secret keys to the cloud server to ensure that curious cloud provider(s) cannot learn from the outsourced database.

In systems that use vertical fragmentation (i.e., column-based fragmentation), the table is fragmented throughout a multi-cloud. So, each column is outsourced to a different cloud to preserve privacy and speed up the query processing [9]. This approach might be practical for tables that have a few attributes but not for tables that have hundreds of attributes. The reason is because the table owner must have multiple accounts with more than one cloud server. If two or more cloud providers collude, privacy will be compromised.

Communication delay is another concern when using such systems. To address this issue, the developed system requires only one server to outsource the encrypted table, a feature that will minimize communication costs, improve privacy, and accelerate query processing.

Homomorphic encryption (HE) is a technique in which SQL queries can be executed over the ciphertexts as if the data were not encrypted. HE is used to encrypt only numeric values and support arithmetic operators over encrypted numeric data. However, the space required to store the ciphertexts is too large. Also, the integrity of encrypted data is not preserved

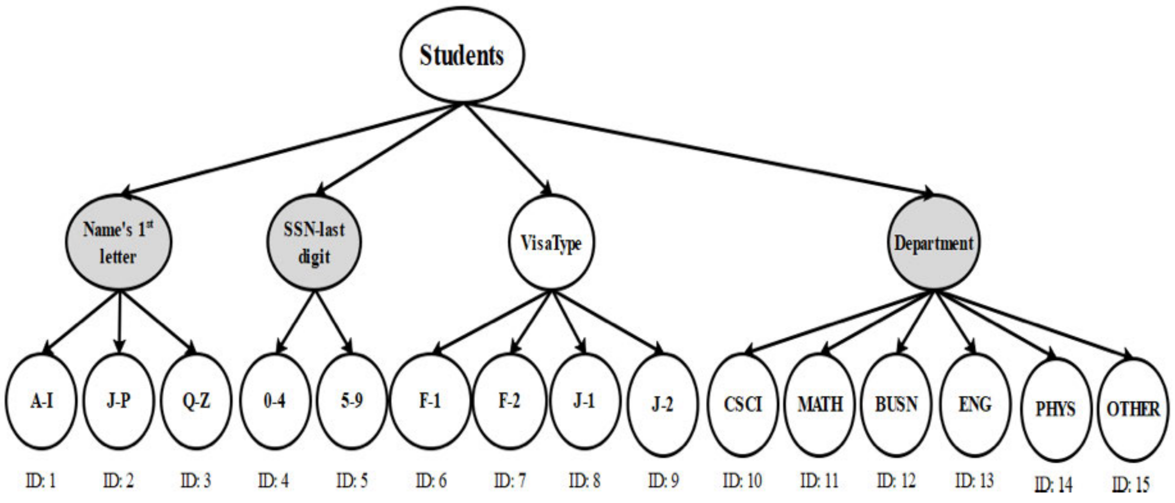


FIGURE 2. An example of the partitioning tree (PT) of the students table, see Table.1.

in such systems because the attacker can change the ciphertexts undetected. To date, this type of encryption supports only addition and multiplication over encrypted digital data. To overcome this, we use a symmetric randomized algorithm AES-CBC to preserve integrity because the modification of a ciphertext leads to an incorrect decryption result.

B. ENCRYPTION STRATEGY

To provide privacy and a high level of security, our approach used AES-CBC to encrypt sensitive data. Only the query manager (QM) keeps and maintains the secret keys (SKs). We used a symmetric algorithm rather than an asymmetric algorithm to gain a higher level of security and faster crypto processing.

C. THE QUERY MANAGER (QM)

In our work [49], we defined the query manager (QM) as a trusted server that resides in an organization's or company's private cloud. It works as an intermediary between users and the Cloud and is responsible for processing queries and encoding BVs for each table (we explain this step in detail in the discussion of partitioning trees). Also, the proposed system supports individuals' cases in which QM would be light software residing in the end user's system. For an organization, it performs the same functions as the QM server. While we assume that the user can encrypt only columns that have sensitive data, the proposed approach even supports encrypting all of a table's attributes.

D. PARTITIONING TREE (PT)

As stated in our previous work [49], [50], the PT is the primary element of proposed system in which the query is appropriately rewritten for execution by the cloud server. The owner of the table participates in the construction of the PT by specifying which columns are sensitive and should be encrypted. Then, the table owner defines the possible partitions for each column and indicates whether the partitions are ranges of values or non-ranges of values. Thus, the values in each column are partitioned into multi-partitions in which each partition includes a set of values. Then, the QM builds the PT based on these specifications. As shown in Figure.2, the name of the table is the root of the tree, and the second-level nodes are the sensitive columns that have to be encrypted. Nodes in the third level, each of which is assigned an ID, represent the partitions of all the sensitive columns.

The second-level nodes are assigned a color of either white or gray. Gray nodes imply that the partitions of the column are ranges of values (e.g., the Name column must have range partitions based on the first letter of the name, while students Visa Type have non-range partitions, such as F-1, F-2, etc.). The PT is stored locally in the QM. We encourage the data owner to define as many as possible partitions for each SC, which can narrow the range of retrieved encrypted records from the outsourced table and, in turn, increase the performance of the proposed systems, and achieve faster query processing.

We are not concerned about memory consumption in our solutions because the sensitive information in every row is encoded into bits (i.e., the smallest computation unit). Based on the PT, this does not consume memory or searching time. In section III-F, we explain in detail how and where to store bit vectors in each proposed system. Algorithm 1 shows how the QM constructs the PT of the students' table (Table.1) and Figure.2 shows the PT of the students' table.

E. ENCODING APPROACH

The encoding process is an essential step in the proposed systems. The proposed scheme is used to encode records' sensitive data into bit vectors (BVs), which can be exploited to retrieve the required encrypted records (i.e., candidate records for the user's query without the need for decrypting data). The QM parses each record to get the names of the sensitive

Algorithm 1 PT Construction

```
1: Input: file.txt containing the name of the table, sensitive columns, and partitions;
2: Create a tree and add a node, represent the table name (TN), as the root of the tree;
3: int Id=0;
4: for each sensitive $SC_i$ do
5:     $Node_i = TN.addchild$(the name of the $SC_i$)
6:     if $SC_i$ contains range of values then
7:         Nodei.AssignColor ('Gray')
8:     else
9:         Nodei.AssignColor ('White')
10:    end if
11:    while partitions! = null do
12:        Nodei.addchild('partition value')
13:    end while
14: end for
```

columns and their values. It then uses the PT to encode the tuples before proceeding to the encryption process, to bit vectors (BVs). Each bit position in the BV is mapped to a partition node from the third level nodes (e.g., the first bit in the BV is mapped to the node having the ID =1). The encoding process is accomplished as follows:

1) 1) For each record R_j in the main table T, the QM creates a BV having a length equal to the number of nodes in the third level of the corresponding PT. It then initializes all its bits to zeroes (e.g., if the bit vector has 10 bits, the number of nodes at the third level of the PT is 10).
2) For each record R_j, the bit b_m that mapped to the partition node PN_m under SC_i is set to one if the datum equals the values represented by PN_m. Then, the QM assigns an index to the newly created BV.

For the sake of clarity, the encoded BVs of the records in Table.1 are as in Figure.3. In section III-F, we present the details of how and where to store the BVs. Algorithm.2 delineates the process of the encoding step.

TABLE 1. The Students Table.

ID	Name	SSN	VisaType	Department
01	Alice	12701	J-1	MATH
02	Ryan	25678	F-2	BUSN
03	Mark	46932	F-1	CSCI
04	John	42213	J-2	PHYS

F. PROPOSED SYSTEM: BIT VECTORS AS A MATRIX (BVM)

In this section, we explain in details the proposed system. Please note that we have published part of this work in [49].

1) SYSTEM DESCRIPTION

In this system, we store and process the BVs of each outsourced table locally at the QM. We assume that the QM is a trusted server residing in either the end user's machine

```
1 <100100010010000>
2 <001010100001000>
3 <010101000100000>
4 <010100010000010>
```

FIGURE 3. The BVs of Records in Table.1.

Algorithm 2 PT Construction

```
1: Define a vector $V$;
2: for each record $R_j$ in Table-T do
3:     Parse $R_j$ to get the value(s) of the sensitive columns $SC_s$;
4:     Define a bit vector $BV_j$ of length n where n = the number of nodes in the 3rd level of the T
5:     for each $SC_i$ in $R_j$ do
6:         if value $v$ = value of the $n^{th}$ sub-node of the $SC_i$ || $v \in$ value of the $n^{th}$ sub-node of $SC_i$ then
7:             Set $n^{th}$ bit in $BV_j$ to 1
8:         else
9:             Set $n^{th}$ bit in $BV_j$ to 0
10:        end if
11:    end for
12: end for
```

as an application or in the private cloud, Figure. 4. Because the only thing outsourced in this prototype is the encrypted table, the highest level of security is provided. The outsourced encrypted table preserves the structure of the original table. However, we add a column (used as a foreign key) to store the rows' indices (during the encoding process, the QM assigns a unique index number to every encrypted row and its BV). Further, we need these indices to fetch the encrypted records. To process a query, the QM needs to load the BVs to the main memory from the hard disk drive (HDD) and perform a rapid look-up to find which records are candidates for the user's query. Then, it pushes their indices into a list and rewrites the query to fetch any record whose index is on the list. In the following sub-sections, we present the supported statements for the relational algebraic operations.

2) BASIC OPERATIONS

The basic operations in database applications are *insert*, *select*, *update*, *alter*, and *delete* statements. We extended this prototype to support these operations, as explained below.

Insertion statements are straightforward. The QM receives the *insert* query, creates a new BV for the newly inserted record, appends it to the corresponding bit vectors matrix (BVM), and then encrypts sensitive data and sends it to the Cloud.

Select is an essential statement in all database applications. In our approach, it is also part of the execution of other

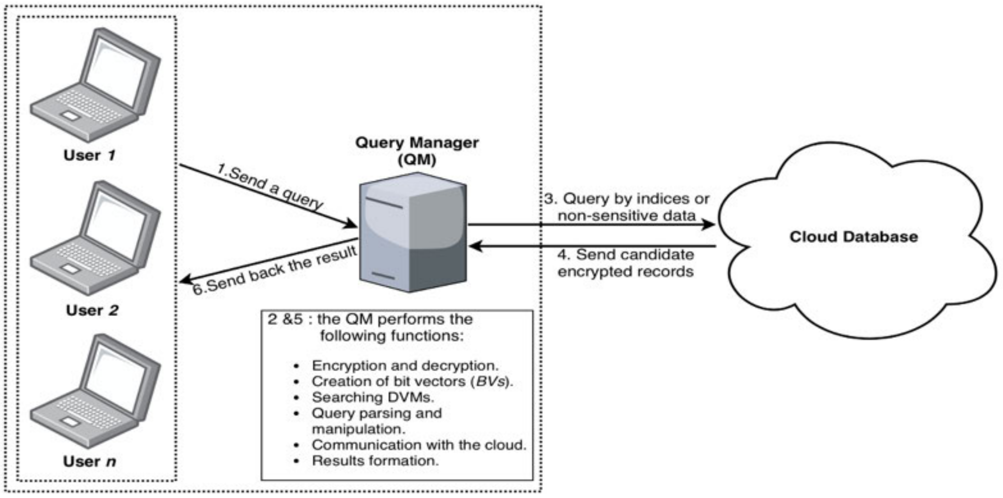

User 1
User 2
User n
Query Manager (QM)
1.Send a query
6.Send back the result
3. Query by indices or non-sensitive data
4. Send candidate encrypted records
Cloud Database
2 &5 : the QM performs the following functions:
• Encryption and decryption.
• Creation of bit vectors (BVs).
• Searching DVMs.
• Query parsing and manipulation.
• Communication with the cloud.
• Results formation.

FIGURE 4. BVM Architecture.

statements like *update* and *delete*. In the QM, the process of executing the *select* statement algorithm is to check which of the following cases is applicable and then enforce it.

- Case 1: None of the column(s) in the query condition is sensitive, so they are not stored in encrypted form in the Cloud. In this case, the QM directly searches and retrieves the data corresponding to the query condition from the Cloud. For example, in Table.1, Figure.1 and Figure.2, if the query condition is "WHERE ID = 03", the QM retrieves the records directly from the Cloud that satisfy this condition.
- Case 2: All column(s) in the query condition are sensitive, so they are stored in encrypted form in the Cloud. In this case, for each column that appears in the query condition, the QM retrieves the indices of corresponding bit vector(s) (BVs) from the BVM. Then it performs logical AND/OR operations based on conditions among indexes returned for each column. For example, if the query condition is "$Name = Mark$ AND $visatype = F1''$, the QM will find any BV having the bit that mapped to the node representing the values of the PD "Q-Z." Because "Mark" belongs to this group (i.e., PD "Q-Z"), any BV has the bit mapped to this partition is set to one will be added to a list L[]. The QM will do the same for visa type, then perform the logical operation AND between the two lists and rewrite the query accordingly to fetch the candidate records.

The *update* process is one of the functions provided by the QM. Through this function, the user issues a query to update record(s). The QM identifies record(s) using our *select* algorithm. Then based on the results of *select* algorithm, the QM issues a query to retrieve encrypted record(s) from the Cloud, decrypts them to find the exact records that match the query conditions, and issues a query to update the encrypted record and its BV. The steps are shown in Algorithm.4.

The *delete* process is used to delete a record from a table. In this case, the QM uses the search algorithm to find the candidate record(s), then removes the record(s) from the outsourced encrypted table and deletes the corresponding BVs from the BVM. Algorithm.5 outlines the steps in the deletion process.

Alter is one of the fundamental operations in any database that allows users to drop/add columns from/to relations. However, in this model, the QM first determines whether the dropped or added column is a sensitive column. In the case of "drop," the QM will look at the corresponding partitioning tree (PT) and drop all partition nodes of the predecessor node that represents the dropped column. It then deletes all bits mapped to the deleted nodes from the BVM. The QM then forwards the alter query to the Cloud, which will execute the query to drop the encrypted column. In the "add column" case, the QM asks the user if the column is sensitive. If it is sensitive, then it will add it with its partitions to the PT after receiving information from the user.

3) RELATIONAL ALGEBRA OPERATORS

Join Most current research in database security does not support the *join* operator because dealing with encrypted databases it is not a straightforward task, especially if AES-CBC is the encryption algorithm. The simple solution for such a task is to retrieve all encrypted tables from the Cloud and perform the *join* operator after decrypting them. However, this is not the optimal choice when it comes to massive tables. In this model, to avoid unnecessary computation, we need to move as much of the join computation as possible to the cloud site without decrypting data, leaving minimal work for the QM. To make BVM both practical and efficient in the *join* operator, we need to consider the following cases for the join condition: 1) The join condition has only non-sensitive columns. 2) The join condition

Algorithm 3 Select

```
1: Receive user query < Table name, List of columns $c_1, c_2, c_3 \ldots$, list of values $v_1, v_2, v_3 \ldots$>
2: Check query columns used in search condition
3: CASE 1:
4: if none of columns mentioned in query is sensitive then
5:     Search the corresponding table in cloud
6:     Return data.
7: end if
8: CASE 2:
9: if each column $C_i$ is sensitive then
10:    for each value v being searched for under column $C_i$ do
11:        Search the column representing that domain value in PT then for each $BV_j$ entry under domain value in $C_i$
12:        if bit = 1 then
13:            Add the index of $BV_j$ to List Li [ ]
14:        end if
15:        Define list F where F [ ] is the final list that contains the indices from the lists $(L_1[], L_2[], \ldots, L_i[])$ after performing AND or OR (based on query conditions) operations between them.
16:    end for
17:    Return F
18:    Retrieve encrypted records from the cloud based on the list F [ ].
19:    Decrypt and return data
20: end if
21: CASE 3:
22: if mixed columns (Sensitive and Non-sensitive) then
23:    Do Case 1 AND Case 2
24:    Remove duplication if found
25:    Return data
26: end if
```

Algorithm 4 Update

```
1: Receive user query < Table name, List of columns $c_1, c_2, c_3 \ldots$, List of values $v_1, v_2, v_3 \ldots$>
2: Use search Algorithm.3 to find the candidate record(s)
3: Fetch the record(s) from the encrypted table
4: Decrypt and find the exact record(s)
5: for each update value $uv$ in record $R_j$ do
6:     if $uv$ falls under non-sensitive column then
7:         Update the old value with $uv$ in the outsourced table
8:         if $uv$ falls under a sensitive column $SC_i$ then
9:             for each $SC_i$ in the update query do
10:                if the $uv$ falls under a $PD$ other than the previous $PD$ then
11:                    Set the bit that mapped to this $PD$ to 1 and unset the rest of bits that mapped to other PDs for this $SC_i$.
12:                end if
13:            end for
14:        end if
15:    end if
16: end for
17: Encrypted the updated values
18: Send data back to the cloud
```

Algorithm 5 Delete

```
1: Use Algorithm 3 (select algorithm) to find the required record.
2: Add the index of each fetched record satisfying the delete condition into list L [].
3: Generate a delete query to delete any record from the outsourced encrypted table its index is in L [].
4: Delete the BVs of the deleted records from the corresponding BVM.
```

involves only sensitive columns having limited distinct value partitions such as USA visa types. 3) The join condition contains only sensitive columns that have range partitions such as salary. 4) The join condition has at least two of the previous cases. So, we design an algorithm to enable the cloud provider to implement a join operator over encrypted tuples without decrypting the encrypted attributes.

To solve the first case, in which the join condition involves only non-sensitive attributes (i.e., unencrypted attributes), the QM will first determine if the attributes are sensitive or not. If they are non-sensitive, it will forward the query to the cloud database, which will implement the join query and return the join result to the QM. The QM then decrypts the join result and removes duplication if found. In the second case, the join condition contains only sensitive attributes that are not ranges of values. In this case, the QM creates a list $L_i[]$ for every partition of the attribute mentioned in the join condition, where $L_i[]$ will have the index of any record having its bit that mapped to partition $node_i$ is set to 1, before searching the BVM. Then the QM will rewrite the query to join all the tuples from both tables based on the indices of these lists. For example, there are two tables, A and B, both having the same attributes: ID, name, rank, department, salary. Now, assume that the query was to join them where A.rank = B.rank. Then let us say that the partition domains for rank attributes are manager, secretary, and employee. Now, the QM will create three lists for each table. The first list will contain the indices of the bit vectors in the corresponding BVM that have their bits mapped to the PD"manager" set to 1. The second list will hold the indices that have the bits mapped to the PD"secretary" set to 1, and the third list will contain the indices that have the bits mapped to the PD"employee" set to 1. In the next step, the QM rewrites the query to join the tuples that have their indices in list $L_i[]$ from Table A with the tuples that have their indices in list $L_i[]$ from Table B. In the third step, the Cloud will execute the query and return the result to the QM, which will decrypt and remove any duplications before sending the result back to the user.

In the third case, the join condition contains only sensitive columns that are ranges of values. This case is similar to the second case. However, in this case, the mapping between joined partitioning lists can be one too many. To illustrate this, consider Figure 3.4. Assume we want to join tables A and B by equality of salary. The salary column in Table A has three PDs that are [(10,000 to 20,000), (20,001 to 30,000), (30,001 to 40,000)], and the salary column has two PDs that are [(10,000 to 25,000), (25,001 to 40,000)]. We need to make sure that the QM rewrites the join query in a way that it maps the PDs from Table A to the corresponding PDs from Table B. To do that, the QM will create n lists for each table where n is the number of PDs in the table that has the fewest PDs in the joining column. So, in the above example, n = 2 since Table B has the least PDs. In the first list L_1[] in table A, the QM adds indices with bits that represent PD_1, or PD_2 is 1. In the second list L_2[] of table A, the QM adds indices with bits that represent PD_2 or PD_3 as 1. The following figure, Figure 3.4, shows how the PDs are mapped. The QM rewrites the query before forwarding it to the Cloud. After getting the join result back from the Cloud for each tuple, the QM decrypts only the encrypted column's value (only the encrypted columns involved in the join condition) and enforces the join condition. If the join condition is satisfied, the QM proceeds to decrypt all of the tuple's values before moving to the next tuple. If the join condition is not met, the QM will not decrypt the entire tuple's values and will move to the next tuple. We do so to avoid unnecessary decryption processes for those tuples that do not meet the join condition. The last case is when the join condition involves two or more of the previous cases. In that case, the QM might rewrite the query. Algorithm.6 below shows how the QM performs the join.

Union is one of the widely used operations in database systems in which tuples from two or more tables are merged. However, to execute a union operator, the number of attributes and datatype of both tables must be compatible. Even though the union operation removes the duplication from the union results, dealing with encrypted data complicates this step. To accomplish this, we could enable the cloud server to execute the union operator over encrypted tables and then send the result back to the QM. Doing so will move the union computation to the cloud server leaving only decryption and duplication removal to the QM. The QM will decrypt each tuple in the union result set and add it into a LinkedHashSet as soon as it receives it from the Cloud. Note that both tables must be encrypted with the same secret key to execute the union algorithm. Algorithm.7 delineates the processes.

Intersection is the process of finding the common subset out of two or more sets. However, executing intersection over encrypted databases is not easy without decrypting the data. If the tables to be intersected have a large number of records, the user is going to add a significant computational overhead by decrypting the whole set of the encrypted tuples from the intersected tables before executing the intersection operator. As a result, we cannot benefit from the cloud services because the computation is moved to the user's side rather than on the cloud side. Some of the previously proposed systems in

Algorithm 6 Join

```
1:  Input: the QM Parses user's query to get tables' names, attributes, and values.
2:  QM checks the PT of each table in the join and performs the following:
3:  Case:1
4:  if join column(s) is non-sensitive (not present in the PT) then
5:      Forward the query to the cloud server (CS)
6:      Define $LinkedHashSet_s$
7:      for every fetched $record_i$ do
8:          decrypt all values in i
9:          push i to s
10:         send s to the user
11:     end for
12: end if
13: Case:2
14: if join column(s) is a sensitive AND not ranges then
15:     for each PDj under the node that represent a SCi in PT of Table x do
16:         Create a list Lxj [ ]
17:         for each BV in $BVM_x$ do
18:             if the bit mapped to $PD_j$ is not 0 then
19:                 Push the BV' index to $Lx_j$[ ]
20:             end if
21:         end for
22:         for each PDj under the node that represent a SCi in the PT of Table x1 do
23:             if the value v of PDj of SCi from table x1 equals to the v of PDj of SCi from Table x2 then
24:                 $PD_j$ of $x_1$ join $PD_j$ of $x_2$
25:             end if
26:         end for
27:         Rewrite the query and send it to the cloud
28:         Define LinkedHashSet s
29:         for every fetched record i do
30:             decrypt all values in i
31:             push i to s
32:         end for
33:         Send s to the user
34:     end for
35: end if
36: Case:3
37: if join column(s) is a sensitive AND ranges then
38:     for each PDj under the node that represent the SCi in PT of Table x do
39:         Create a list $Lx_j$[]
40:         for each BV in $BVM_x$ do
41:             if the bit mapped to PDj is not 0 then
42:                 Push the BV' index to Lxj [ ]
43:             end if
44:         end for
45:         for each PDj under the node that represent the SCi in PT of Table x1 do
46: if the PDj of SCi from table x1 contains at least one value v where v ∈ PDj from table x2 then
```

```
47:         $PD_j$ of $x_1$ join $PD_j$ of $x_2$
48:       end if
49:   end for
50:   Rewrite the query and send it to the cloud
51:   Define LinkedHashSet s
52:   for each fetched record i do
53:       Decrypt only the join columns' values
54:       if the join condition satisfied then
55:           Decrypt the whole tuple's values
56:           Push i to s
57:       else
58:           Proceed to the next encrypted tuple
59:       end if
60:   end for
61:   Send s to the user
62:
63:   Case:4
64:   The query involves two or more of the above cases.
```

[2], [9], [10] can process queries over encrypted databases but will experience delays if the tables to be intersected have large numbers of tuples.

In this model, our goal is to move the computation as much as possible to the cloud side while eliminating unnecessary decryption processes at the QM. Moreover, we want to execute the intersection operator partially in the cloud database server leaving only the elimination of duplication at the QM. In this way, we exploit the high computational speed provided by a cloud database server to accelerate the query processing time. We explain the simulation of intersection as follows:

- The user sends the query to the QM, which is going to parse it to remove the headers of tables and columns.
- If the intersect operation involves k columns out of n columns, where n denotes the total number of columns in a table, the QM uses our *join* algorithm to join both tables by k columns and then rewrites the query. Otherwise, the QM chooses all SCs that are not in ranges to join the tables using our *join* algorithm before rewriting the query.
- The QM sends the translated query to the cloud database server, which will execute the query and send back the *join* result to the QM.
- Before returning the *intersecting* result to the user, the QM decrypts the encrypted *join* set and pushes it to a hash list to remove duplicates, if found.

Difference To execute the difference in this prototype, the intersection is first enforced between the tables to find the common tuples. Second, the QM sends a query to retrieve all tuples except the join result set. Third, the QM decrypts the result and sends back the result.

Duplication Removal Enforcing duplication removal over a query result (encrypted tuples) at the Cloud is impossible because we use a non-deterministic encryption algorithm (AES-CBC). Therefore, we leave the execution of

Algorithm 7 Union

```
1: Forward the user's query to the cloud server
2: Define a vector $v$
3: for each fetched $record_i$ in the union result set do
4:     Decrypt $i$
5:     Add $i$ to $v$
6: end for
7: v.distinct()
8: Send $v$ to the user.
9: v.clear()
```

Algorithm 8 Duplication Removal

```
1: execute steps 1-9 of Algorithm.3
2: if distinct keyword is present in the original query then
3:     define $LinkedHashSet_s$
4:     for each fetched $tuple_i$ do
5:         decrypt values of i
6:         push $i$ to $s$
7:     end for
8:     send $s$ to the user.
9: end if
```

this operator to be accomplished at the QM before sending back the user's result. That means the QM will decrypt the encrypted tuples retrieved from the Cloud. If the query contains the duplication elimination keyword *distinct* the QM will define a LinkedHashSet data structure that does not allow duplicated elements in the set. It will then add each decrypted tuple to the set. Note that the translated query to be executed by the cloud server will not have the "*distinct* keyword. Further, the keyword *distinct* usually appears in *select* queries, in which case the algorithm to eliminate duplication is the *select* algorithm, and we add three more steps to execute the *distinct* operator. See Algorithm.8.

Aggregation and Sort To implement the aggregation and sort operators (max, min, and count) over encrypted tables in the Cloud, we consider two cases for sensitive columns, ranges, and non-range columns. In non-range columns, we can process the query locally at the QM with no need to communicate with the Cloud. In such a case, we avoid the decryption computation that results from retrieving encrypted records from the Cloud. Consequently, we will achieve faster query processing. Specifically, the QM searches the BVM after looking up the corresponding PT using our search algorithm to obtain a list of all BVs' indices that satisfy the query condition. Note that the QM might not need to do further computations such as decryption processes and will, therefore, send the query result back to the user. For the sake of clarity, consider Table 3.1, Figure 3.1, and Figure 3.2, suppose a user send the following query:

select count(name) from students where department = 'computer science'; then, the query is processed as below:

- The QM will search the PT and find that Dept is a non-range column (node's color is gray). Then it will search the BVM (Figure 3.2) and push the index of any BV having the bit representing the PD "computer science" is not zero into a list L[].

- The QM counts the number of the elements in L[] and returns the number to the user.

On the other hand, the query process is divided into two phases. The first phase is accomplished at the QM, while the second phase is handled in the Cloud. The QM executes the aggregation or sort operators over decrypted records after the QM processes the query to retrieve only candidate encrypted tuples that are related to the query. For example, the query *Select* count ('name') from student where the name = 'Alice'; is processed as follows:

- The QM looks up the PT and will find the name is a range column (node's color is not gray). It will then search the corresponding BVM and add to the list L[] the indices of all BVs that have the bit assigned to the PD "A-F" is one.
- The QM then looks for any encrypted record in which its index is present in L[] from the Cloud using this query syntax: *select* name from the student where index in ("elements of L[] separated by commas ").
- The QM decrypts every fetched tuple's name and increments the count value only if the decrypted name value equals 'Alice.'
- The QM returns the value of the count variable to the user.

The SUM and AVERAGE functions are processed similarly as a count, but we sum the decrypted numbers. If the operation is average, we divide the sum over the number of decrypted values that meets the query conditions. The sorting operator can be executed similarly to the aggregation operator. However, we need to run the sort operator over decrypted data before sending the result back to the user. Algorithm.9 shows the process.

Project In project queries, the QM does a column-based retrieval; it will select all tuples for specific column(s). Further, we do not need to perform a PT lookup in the project. However, we need to decrypt the whole set of retrieved tuples at the QM. We do not need to remove duplication of doing any filtration at the QM.

Algorithm 9 Aggregate Functions

```
1:  Do steps 1 to 6 of Algorithm.3.
2:  if all SCs are non-ranges then
3:      if the operator is count then
4:          Do steps 9 to 12 of Algorithm.3
5:          Let x = Count the number of Li []
6:          Return x
7:      end if
8:      if the operator is max then
9:          Let m = the value of right most PD of the SC
            predecessor node in the PT
10:         Return m
11:     end if
12:     if the operator is min then
13:         Let n = the value of left most PD of the SC prede-
            cessor node in the PT
14:         Return n
15:     end if
16: end if
17: if all SCs are ranges then
18:     if the operator is count then
19:         Do steps 9 to 15 of Algorithm.3
20:         Define List $L_k$[]
21:         for each fetched record i do
22:             Decrypt i
23:             if i meets the query condition then
24:                 Add it to $L_k$[]
25:             end if
26:         end for
27:     end if
28:     Return size of $L_k$[] to the user.
29:     if the operator is max then
30:         for the right most PD of the SC do
31:             Do steps 10 to 15 of Algorithm.3
32:             Define a variable x
33:             for each fetched record i do
34:                 Decrypt i
35:                 if i > x then
36:                     x = i
37:                 end if
38:             end for
39:         end for
40:     end if
41:     Return x to the user
42:     if the operator is min then
43:         for the left most PD of the SC do
44:             Do steps 10 to 15 of select algorithm
45:             Define a variable x
46:             for each fetched record i do
47:                 Decrypt i
48:                 if i < x then
49:                     x = i
50:                 end if
51:             end for
52:         end for
53:     end if
54:     Return x to the user.
55: end if
```

IV. EXPERIMENTS AND EVALUATION

A. EXPERIMENTAL SETUP

We used a PC with 6GB of RAM, 1TB HDD, and a Core i5 processor with 2.8 GHz to conduct all the experiments for all the systems (BVM, OBT, CBF, and CryptDB). To implement the functions of the QM in the proposed system, we used Java to simulate each task as a java class or method. MySQL server was used on the user's machine and we used Java Database Connectivity (JDBC) as a connector from Java to the MySQL engine. All the experiments were performed on the local machine; therefore, the communication delay variable was removed from all the reported delays.

We implemented the OBT and CBF because their implementations are not available online unlike CryptDB, which is available for public use on GitHub [51]. While implementing the CBF, we adopted the implementation in [52] to encrypt numerical values in a way that preserves the order (OPE) and in [53] to support the additive homomorphic property. In our models, we used the randomized version of the AES-CBC to encrypt sensitive data. In our systems, each tuple's data are transmitted to the encryptor class as soon as the encoding step has been accomplished. The encryptor and decryptor classes call numerous cryptographic packages, including the "javax.crypto" package offering the classes and interfaces for crypto tasks; more information can be found in [54], and the table owner's secret key (SK) and the pre-generated initialization vectors (IVs) can be used to encrypt or decrypt each tuple. The SK is 256 bits, and each IV is 128 bits (the IV size equals the block size in the AES). To store the bit vectors (BVs), we stored them locally at the QM, and we wrote them in a text file for future use (in the future, the QM just reads the BVs set from the file.txt and loads them to the data structure).

B. DATASETS AND PARTITIONING TREE

We randomly generated four tables. We defined a list of values for each attribute and let a java program constructs tuples by randomly picking values from the lists. The sizes of the tables (i.e., the number of records) were 10k, 20k, 50k, and 100k records. We had 24 attributes in total for all tables, and we considered all of them, except ID attributes, as sensitive attributes. In our study, although we could use the proposed models with small tables, we focused on the large tables since it is easier to test the penalties introduced by each scheme. Table.2 presents the structure of the main tables. For each table, we created a table in the cloud according to the created algorithm for each model. We built a partitioning tree (PT) for all the tables based on Table.3 Table 4.2.

TABLE 2. The Structure of the Original (Plaon) Students' Table.

Name of the Attribute	Datatype	Storage Required (bytes)
ID	int	4
Name	varchar	20
SSN	int	4
Visa Type	varchar	6
Salary	int	4
Department	varchar	20

TABLE 3. The sensitive attributes and the number of partitions for students table.

Attribute	Sensitivity	Number of Partitions
ID	No	0
Name	Yes	3
SSN	Yes	2
Visa Type	Yes	4
Salary	Yes	5
Department	Yes	6
Total number of partitions		20

TABLE 4. The delay of the original database encryption comparison among all systems in minutes.

N.R	BVM	CryptDB	CBF	OBT
10k	12	44	55	13
20k	26	66	112	24
50k	65	158	291	55
100k	147	308	578	113

We generated the tables so that they were fairly distributed to the PT. For example, for the attribute (Name), not all the records were mapped to the first partition (node #1) under the name predecessor; instead, approximately 33% of the records were mapped to the first node, 33% assigned to the second node, and 34% assigned to the third node. We considered the same technique for the rest of the attribute partitions.

C. EVALUATION

The evaluation consisted of the following:

1) Testing and evaluating basic database operations (create, select, insert, update, and delete statements) execution cost
2) Testing and evaluating aggregation operations (sum, average, count, max, and min) execution cost
3) Testing and evaluating joining and setting operations (join, union, and intersection) execution cost

For each part, we considered different factors that play a role in the efficiency, such as the number of clauses in the query conditions, what the logic operation (AND/OR) is in the condition, and what encrypted attributes to retrieve for the tuples. In the discussion, we explore the factors that make the proposed models more efficient and what makes them inefficient. In addition, we discuss the impact of the PT size on the efficiency of the proposed model.

1) EXECUTION DELAY COMPARISON

a: ORIGINAL DATABASE ENCRYPTION AND INSERT STATEMENTS

In the proposed model, the original database encryption step involves parsing records, generating indices, building BVs, encrypting sensitive data, and inserting the encrypted data into the encrypted table in the cloud server. In Table.4, we compare the time taken by each system to encrypt each table. As seen in Table.4, the OBT is the most efficient system followed by BVM. On the other hand, the CBF experienced the highest delay since the encryption process required N insertion (N = number of columns + 1) into N different tables in different cloud servers. The second slowest model is CryptDB in both the creation and insertion processes due to heavy computation results from the onion layer encryption as seen in Figure.5. In summary, the proposed model is faster than the CryptDB and CBF models in both the creation and insertion processes.

Experiment 1: In this experiment, we calculated the average percentage of fetched encrypted tuples from the encrypted tables for the proposed system. We also studied

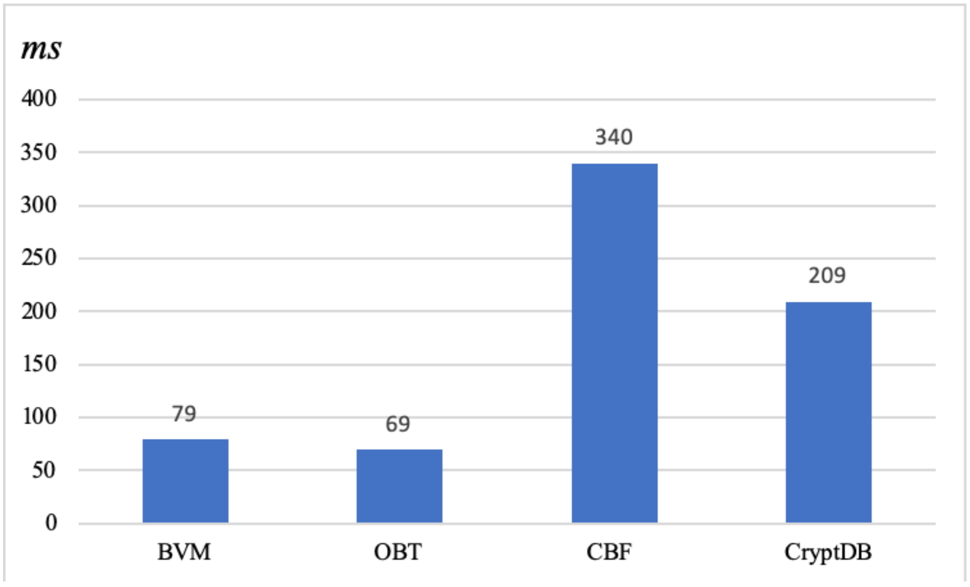


FIGURE 5. The delay comparison of insert statements for all systems, in milliseconds.

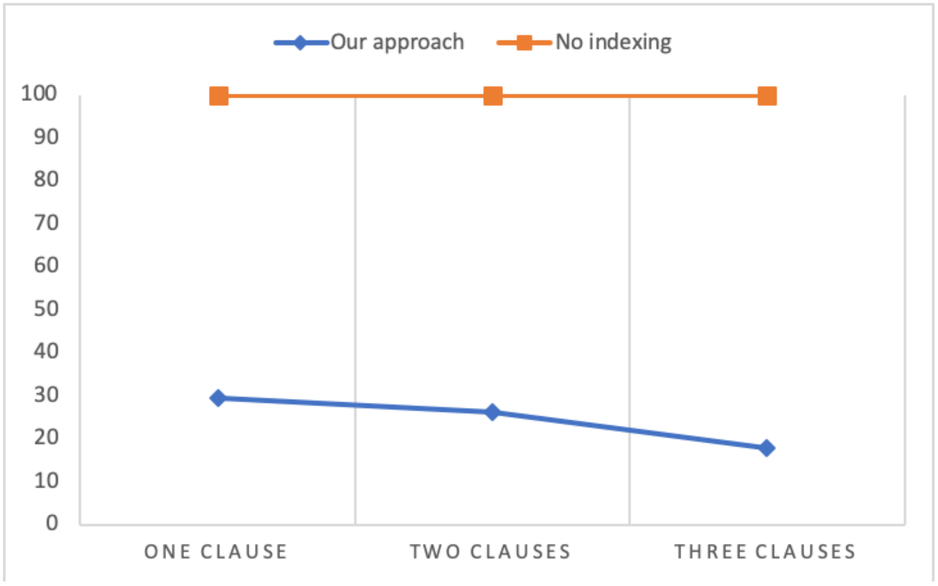


FIGURE 6. The percentage of retrieved encrypted tuples for the proposed model.

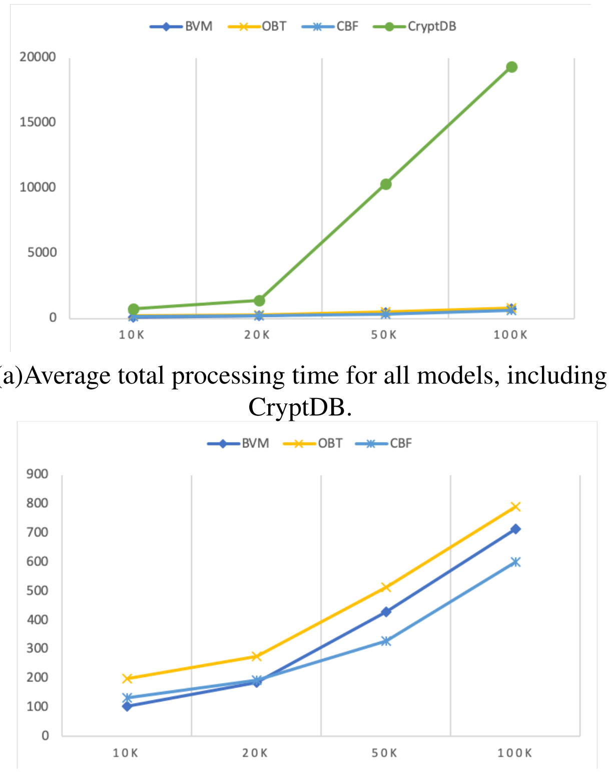
how the number of clauses in the query condition can contribute to narrowing the range of the fetched set. Figure.6 illustrates how our model dramatically drop the average retrieved encrypted candidate records for select statements to about 31% when only one clause is present in the query condition, while it fetched the least percentage when the condition clause had three clauses. On the other hand, without using the proposed system to manage the randomized encrypted database (i.e., no indexing), we must retrieve the entire outsourced encrypted table.

Experiment 2:

- **Select (*) Latency:** Table.5 presents the total runtime for all systems when executing *select* statements to retrieve single column values. The runtime we measured was the time from query parsing until the final query result was formed in milliseconds (*ms*). In the first case, we measured the average runtime when the condition clause of the queries features only one clause. The delay is the average delay of executing a select statement on each sensitive column. Furthermore, we tested *select*∗ statements when the condition had two and three clauses. Figure.7, Figure.8, and Figure.9 present the total execution time of *select*∗ statements for each model. As seen in the figures, the proposed system (BVM) performed better for databases with more than 50k rows, and the main factor that affects the performance of the

TABLE 5. Delays in milliseconds of executing (*select*∗). The average required records are 8% for one clause, 5% for two clauses, and 3% for three clauses.

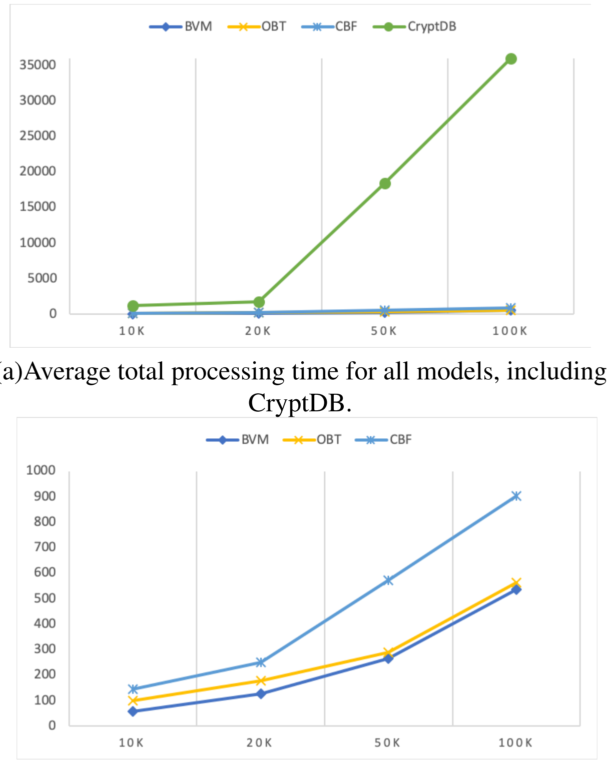
	1 clause				2 clauses				3 clauses			
	BVM	OBT	CBF	CryptDB	BVM	OBT	CBF	CryptDB	BVM	OBT	CBF	CryptDB
10k	242	200	231	870	136	98	238	1329	115	77	324	1558
20k	326	275	338	1633	259	179	483	2016	213	142	542	12626
50k	624	481	547	10764	479	289	746	18969	440	221	893	21634
100k	1107	791	970	20135	840	562	1478	37078	835	493	1604	59910



(a)Average total processing time for all models, including CryptDB.

(b)Average total processing time for all models, including CryptDB.

FIGURE 7. The delay comparison of executing select all statements (*select*∗) for all models in ms when the query condition contains only one clause.



(a)Average total processing time for all models, including CryptDB.

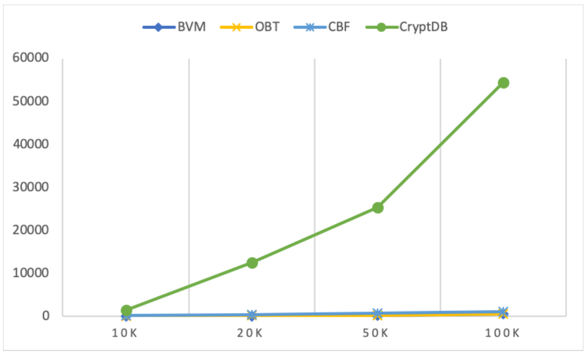
(b)Average total processing time for all models, excluding CryptDB.

FIGURE 8. The delay comparison of executing select all statements (*select*∗) for all models in ms when the query condition contains two clauses.

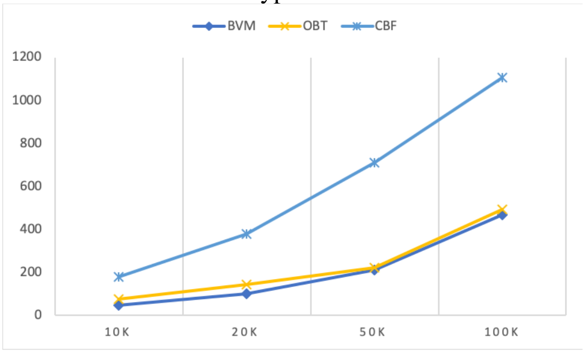
BVM is the time for loading the BVs from the hard drive to the main memory and then searching them. Forming the final lists of the candidate record indices is another factor that affects the delay in this model.

In terms of comparing the proposed system with the other approaches, as seen in Table.5, Figure.7, Figure.8, and Figure.9 the proposed system is faster than all the competing systems except for the OBT for *select* statements with two or three clauses. The OBT experienced the least delay since the amount of decrypted data was less than in our approaches (all values are stored as one block leading to fewer bytes to decrypt for each row). Cell-based encryption produces longer ciphertexts (i.e., the blocks less than 16 bytes will be padded) and then higher decryption overhead. However, when the *select* query is not to select all (*select*∗), our system performs better than the OBT because we eliminate decrypting whole rows in our models, whereas the OBT does not. When the query condition involved a single clause, the CBF system experienced delays comparable to our system, but when the number of clauses in the *select* statements was two or three, its delay was almost double that of our system's delays since two or three tables are searched to form the final *select* query (the query that retrieves the records from the main encrypted table in the master cloud). Finally, the CryptDB incurs the highest delay among the systems as a result of the decryption of the onion layers overhead. The delay also increases when the statement condition has two or three clauses because more onion layers are required to be slipped off in different columns.

- **Throughput:** Throughput is defined as the amount of data transferred at a given time. By measuring the throughput, we can tell which system is more responsive



(a)Average total processing time for all models, including CryptDB.



(b)Average total processing time for all models, excluding CryptDB.

FIGURE 9. The delay comparison of executing select all statements (*select*∗) for all models in ms when the query condition contains three clauses.

to user's queries when the requested data are increased since the end user is the one who will be affected by the system slowdown. To measure the throughput, we executed a set of queries to retrieve 25% then 50% of the records from a table holding 100,000 records. Then, we measure the amount of plain data (records' data after decryption) and divide it by the time taken by each system to deliver the required data to the user [55].

$$Throughput = \frac{\sum_{k=0}^{n}(unencrypted\ \ record's\ \ size\ \ (in\ \ bytes\ \ ))}{Total\ \ time\ \ taken\ \ to\ \ deliver\ \ the\ \ data\ \ (MS)} \quad (1)$$

As seen in Table.6, the proposed system achieved a higher throughput when compared with CryptDB and CBF systems which makes it the best choice for end users who seek a faster responsive system. OBT has the highest throughput and that is because it requires the least bytes requirements among all systems to encrypt data.

In Figure.10, we show the percent of the throughput for each system and we compare the systems with the throughput of MySQL when dealing with unencrypted data. To calculate the percent, we multiply the system throughput by 100 and divide it by the unencrypted throughput.

$$Throughput\ \ Percent = \frac{(System\ \ throughput * 100)}{(Unencrypted\ \ throughput)} \times 100 \quad (2)$$

By examining Figure.10, we can say that our system achieved a reasonable throughput percent and as the amount of required data increased (up to 50% of the rows), the throughput dropped slightly and still outperform CryptDB and CBF.

TABLE 6. The amount of plain data in kB that each system can deliver to the user per second.

	kilobytes per second (kB/s)				
	BVM	OBT	CBF	CryptDB	MySql
Requested rows = 25%	794	1440	727	66	4720
Requested rows = 50%	971	1447	816	57	10114

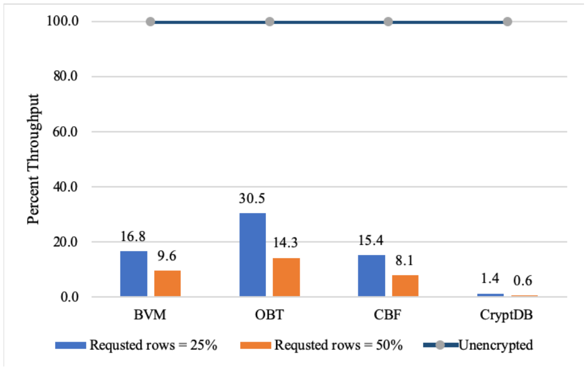


FIGURE 10. The percent of throughput of all system compared with the throughput of MySQL when requesting unencrypted data. Note, the requested data are fetched by a select query from a table with 100,000 records.

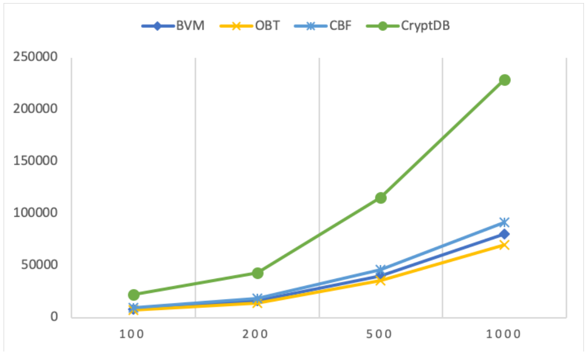


FIGURE 11. Comparison of the average delay of update statements for all models to update a number of existing tuples (100, 200, 500, and 1,000 tuples).

Experiment 3:

- **Update and Delete Statements:** Figure.11 depicts the average time cost taken by each system to execute update statements. In this experiment, we executed update statements with only one predicate. As seen in Figure.11, the update time cost is high in all systems and is the result of updating an encrypted field in the database systems.

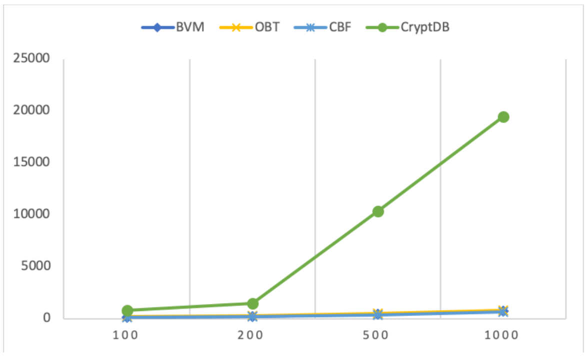


FIGURE 12. Comparison of the average delay of delete statements for all models to delete a different number of tuples (100, 200, 500, and 1,000 tuples).

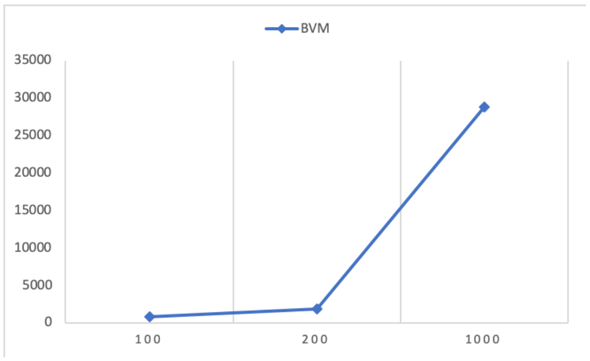


FIGURE 13. Join delay.

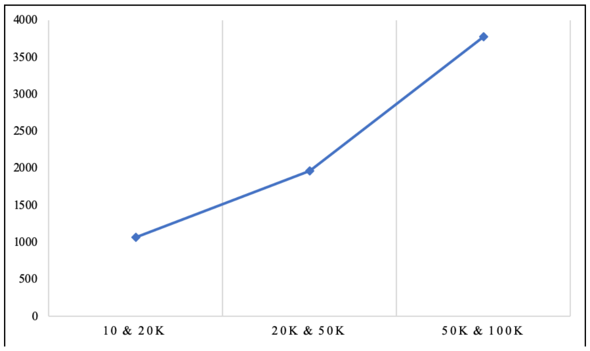


FIGURE 14. Union delay growth for different table sizes.

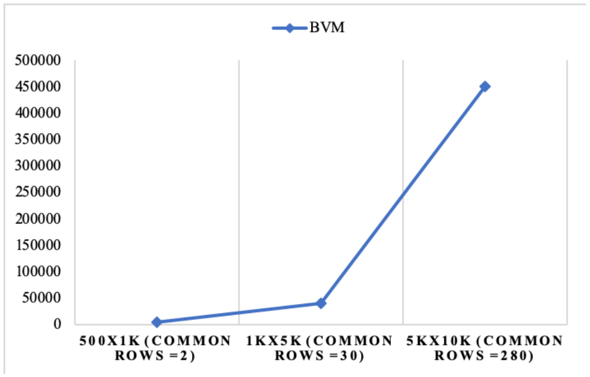


FIGURE 15. Union delay growth for different table sizes.

The x-axis represents the number of rows affected by the update statements, which means selecting the required data and then issuing 100 insert statements in the first case, 200 insert statements in the second, and so on. The OBT system has an update delay (7,130 ms), however, the update process is risky when two or more substrings in the decrypted block match the updated value (i.e., any substring from the updated record that matches the new substring will be updated to the new substring). Thus, update in the OBT is not practical. By zooming to the delays of our system, it experienced a slightly higher delay than the OBT but still performed faster than CBF and CryptDB. In CryptDB, the update cost is the highest of the compared systems.

Figure.12 demonstrates the time taken by each model to delete different numbers of records (100, 200, 500, and 1,000 tuples) when the delete condition has only one clause. The delete process selects the required rows and then deletes them. The delete is efficient in the proposed system since the deletion is performed after executing the select operation to retrieve the needed tuples. Instead of sending a single query to delete each record, we maintain the index of the record and then issue one query at the end to delete any record of its index in the delete query, that is, delete from TABLE_NAME where index in ( ). On the other hand, CryptDB is the slowest system to execute delete statements for the same reasons we mentioned earlier (i.e., onion layers decryption).

- **Join, Union, and Intersection:** In this study, we report the delay of the proposed system for join and union queries. In join queries, we did not include the competing systems in this experiment because the join has not been implemented in CryptDB (as the author stated in [51] and the CBF. Therefore, we excluded all the competing systems from this experiment, and we performed the experiment on tables with the specifications displayed in Table.7.

  We joined two tables by the equality of the encrypted ID, and their structures are in Table.9 and Table.8 below. The ID column in both tables is partitioned into 10 partitions based on the last digit in the ID as (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). Figure.13 demonstrates that the delay of the proposed system (BVM). The high delays in the join queries is due to the nature of the join calculation and the amount of the decrypted data. Then, after enforcing the join condition (at the QM), the decrypted tuples were pushed into a LinkedHashSet to remove the duplication. In conclusion, although we experienced high join delays, we successed to execute the join statements in our system over databases encrypted with a randomized encryption algorithm, such as AES-CBC.

  Figure 14 demonstrates the average total time of processing the union queries of the proposed system.

TABLE 7. Specifications of the joined tables.

Table size (number of rows)	Number of rows matching the join condition
500	100
1000	200
10000	1000

TABLE 8. The structure of original international students table.

Name of the Attribute	Data type	Is sensitive?
ID	int	Yes
Name	varchar	Yes
Address	int	Yes
Citizenship	varchar	Yes

TABLE 9. The structure of TA students table.

Name of the Attribute	Data type	Is sensitive?
ID	int	Yes
Name	varchar	Yes
SSN	int	Yes
Visa_Type	varchar	Yes
Salary	int	Yes
Department	varchar	Yes

The delay we measured is from the time the QM intercepts the query until the final query result is formed. To remove the duplications, we pushed the decrypted tuples from both tables to a LinkedHashSet, which ensures no duplicated records exist. As seen in Figure.14, our system experienced a linear delay growth as the sizes of the tables increased.

In Figure.14, we show the latency, in milliseconds, of the intersection operator when executed on different tables with a different number of records. To perform this experiment, we reduced the tables' sizes because we experienced execution failures due to the lake of memory. Moreover, the leading cause to get this kind of error is the massive join computations that result from implementing the intersection operator. Note that the intersection queries were to intersecting tables based on all columns (i.e., not a partial intersection).

In Figure.15, we show the latency, in milliseconds, of the intersection operator when executed on different tables with a different number of records. To perform this experiment, we reduced the tables' sizes because we experienced execution failures due to the lake of memory. Moreover, the leading cause to get this kind of error is the massive join computations that result from implementing the intersection operator. Note that the intersection queries were to intersecting tables based on all columns (i.e., not a partial intersection)

V. CONCLUSION

Cloud computing is an attractive computing environment for all kinds of users and companies. But, privacy breaches, not only by malicious attackers but also by curious providers, is the downside of this type of service, because users lose access control over outsourced data. There are many solution for this problem and data encryption is the effective one. However, executing SQL queries over encrypted data is challenging, especially if a randomized encryption algorithm, like AES-CBC, is used for the encryption. In this research, we first introduce the QM, a trusted server, which works as an intermediate between the cloud server and user(s) and performs all the crypto processes. In addition, we design a novel indexing technique based on predefining partitions for each sensitive attribute, and then encode each tuple to bits, accordingly. The bits are used to retrieve candidate tuples for a specific query that minimize the range of the retrieved encrypted tuples. Based on this encoding scheme, we proposed a secure systems to stores and maintains the index data (i.e., the bit vectors [BVs]) locally at the QM, i.e., in the private cloud. Besides, we design different algorithms to accomplish query execution of different SQL relational algebra operators, and we make it resistant to diffrent attack scenarios, such as inference attacks. We test the proposed system by implementing it and comparing its performance against some of well-known state-of-the-art systems like CryptDB. We evaluate it in terms of execution time and space requirements. We find that the proposed system require both less execution time and space when compared with most other competing systems.

REFERENCES

[1] Online. (1999). *Cloud Database Market*. [Online]. Available: https://www.marketresearchfuture.com/reports/cloud-database-market-6847

[2] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Processing queries on an encrypted database," *Commun. ACM*, vol. 55, no. 9, pp. 103–111, Sep. 2012.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2004, pp. 563–574.

[4] Y.-D. Jang and J.-H. Kim, "A comparison of the query execution algorithms in secure database system," *Int. J. Electr. Comput. Eng.*, vol. 6, no. 1, p. 337, Feb. 2016.

[5] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 644–655.

[6] D. Pouliot and C. V. Wright, "The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1341–1352.

[7] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 698–706, Mar. 2015.

[8] F. Oggier and M. J. Mihaljević, "An information-theoretic security evaluation of a class of randomized encryption schemes," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 2, pp. 158–168, Feb. 2014.

[9] A. Alsirhani, P. Bodorik, and S. Sampalli, "Improving database security in cloud computing by fragmentation of data," in *Proc. Int. Conf. Comput. Appl. (ICCA)*, Sep. 2017, pp. 43–49.

[10] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2002, pp. 216–227.

[11] M. Nabeel, E. Bertino, M. Kantarcioglu, and B. Thuraisingham, "Towards privacy preserving access control in the cloud," in *Proc. 7th Int. Conf. Collaborative Comput., Netw., Appl. Worksharing (CollaborateCom)*, 2011, pp. 172–180.

[12] Y. Zhu, H. Hu, G.-J. Ahn, D. Huang, and S. Wang, "Towards temporal access control in cloud computing," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2576–2580.

[13] J. Raigoza and K. Jituri, "Evaluating performance of symmetric encryption algorithms," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2016, pp. 1378–1379.

[14] M. B. Yassein, S. Aljawarneh, E. Qawasmeh, W. Mardini, and Y. Khamayseh, "Comprehensive study of symmetric key and asymmetric key encryption algorithms," in *Proc. Int. Conf. Eng. Technol. (ICET)*, Aug. 2017, pp. 1–7.

[15] M. Chakraborty, B. Jana, T. Mandal, and M. Kule, "An performance analysis of RSA scheme using artificial neural network," in *Proc. 9th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2018, pp. 1–5.

[16] T. P. Innokentievich and M. V. Vasilevich, "The evaluation of the cryptographic strength of asymmetric encryption algorithms," in *Proc. 2nd Russia Pacific Conf. Comput. Technol. Appl. (RPC)*, Sep. 2017, pp. 180–183.

[17] A. Boicea, F. Radulescu, C.-O. Truica, and C. Costea, "Database encryption using asymmetric keys: A case study," in *Proc. 21st Int. Conf. Control Syst. Comput. Sci. (CSCS)*, May 2017, pp. 317–323.

[18] O. M. B. Omran and B. Panda, "Efficiently managing encrypted data in cloud databases," in *Proc. IEEE 2nd Int. Conf. Cyber Secur. Cloud Comput.*, Nov. 2015, pp. 266–271.

[19] O. M. B. Omran and B. Panda, "A new technique to partition and manage data security in cloud databases," in *Proc. 9th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2014, pp. 191–196.

[20] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," in *Proc. 10th ACM Conf. Comput. Commun. Secur. (CCS)*, 2003, pp. 93–102.

[21] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, and Y. Wu, "A framework for efficient storage security in RDBMS," in *Proc. Int. Conf. Extending Database Technol.* Berlin, Germany: Springer, 2004, pp. 147–164.

[22] E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes, "Designing secure indexes for encrypted databases," in *Proc. IFIP Annu. Conf. Data Appl. Secur. Privacy*. Berlin, Germany: Springer, 2005, pp. 54–68.

[23] W.-K. Wong, K.-W. Wong, H.-Y. Yue, and D. W. Cheung, "Non-order-preserving index for encrypted database management system," in *Proc. Int. Conf. Database Expert Syst. Appl.* Cham, Switzerland: Springer, 2017, pp. 190–198.

[24] F. Hahn, N. Loza, and F. Kerschbaum, "Joins over encrypted data with fine granular security," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Apr. 2019, pp. 674–685.

[25] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 289–300, Mar. 2013.

[26] Y.-F. Zhuang, C.-Z. Wei, J. Li, and W.-G. Li, "Performance enhanced for CryptDB based on AES-NI acceleration," *DEStech Trans. Comput. Sci. Eng.*, Jul. 2017.

[27] A. Kumar and M. Hussain, "Secure query processing over encrypted database through CryptDB," in *Recent Findings in Intelligent Computing Techniques*. Singapore: Springer, 2018, pp. 307–319.

[28] G. Liu, G. Yang, H. Wang, Y. Xiang, and H. Dai, "A novel secure scheme for supporting complex SQL queries over encrypted databases in cloud computing," *Secur. Commun. Netw.*, vol. 2018, pp. 1–15, Jul. 2018.

[29] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2009, pp. 169–178.

[30] K. Li, W. Zhang, C. Yang, and N. Yu, "Security analysis on one-to-many order preserving encryption-based cloud data search," *IEEE Trans. Inf. Forensics Security*, vol. 10, no. 9, pp. 1918–1926, Sep. 2015.

[31] S. Cui, M. R. Asghar, S. D. Galbraith, and G. Russello, "P-McDb: Privacy-preserving search using multi-cloud encrypted databases," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 334–341.

[32] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[33] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. NDSS*, vol. 71, 2014, pp. 72–75.

[34] O. M. Omran, "Data partitioning methods to process queries on encrypted databases on the cloud," Ph.D. dissertation, Univ. Arkansas, Fayetteville, AR, USA, Tech. Rep. 1580, 2020.

[35] S. Shastri, R. Kresman, and J. K. Lee, "An improved algorithm for querying encrypted data in the cloud," in *Proc. 5th Int. Conf. Commun. Syst. Netw. Technol.*, Apr. 2015, pp. 653–656.

[36] M. R. Asghar, G. Russello, B. Crispo, and M. Ion, "Supporting complex queries and access policies for multi-user encrypted databases," in *Proc. ACM Workshop Cloud Comput. Secur. Workshop*, Nov. 2013, pp. 77–88.

[37] W. K. Wong, B. Kao, D. W. L. Cheung, R. Li, and S. M. Yiu, "Secure query processing with data interoperability in a cloud database environment," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2014, pp. 1395–1406.

[38] T. Raybourn, "Bucketization techniques for encrypted databases: Quantifying the impact of query distributions," Ph.D. dissertation, Bowling Green State Univ., Bowling Green, OH, USA, 2013.

[39] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proc. 30th Int. Conf. Very Large Data Bases*, vol. 30, 2004, pp. 720–731.

[40] J. Wang, X. Du, J. Lu, and W. Lu, "Bucket-based authentication for outsourced databases," *Concurrency Comput., Pract. Exper.*, vol. 22, no. 9, pp. 1160–1180, 2010.

[41] J. Li, Y. K. Li, X. Chen, P. P. C. Lee, and W. Lou, "A hybrid cloud approach for secure authorized deduplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1206–1216, May 2015.

[42] M. Tao, J. Zuo, Z. Liu, A. Castiglione, and F. Palmieri, "Multi-layer cloud architectural model and ontology-based security service framework for IoT-based smart homes," *Future Gener. Comput. Syst.*, vol. 78, pp. 1040–1051, Jan. 2018.

[43] V. H. Hacigumus, B. R. Iyer, and S. Mehrotra, "Query optimization in encrypted database systems," U.S. Patent 7 685 437, Mar. 23, 2010.

[44] L. Bouganim and P. Pucheral, "Chip-secured data access: Confidential data on untrusted servers," in *Proc. 28th Int. Conf. Very Large Databases (VLDB)*, 2002, pp. 131–142.

[45] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 264–278.

[46] G. Wang, C. Liu, Y. Dong, H. Pan, P. Han, and B. Fang, "SafeBox: A scheme for searching and sharing encrypted data in cloud applications," in *Proc. Int. Conf. Secur., Pattern Anal., Cybern. (SPAC)*, Dec. 2017, pp. 648–653.

[47] C. Liu, G. Wang, P. Han, H. Pan, and B. Fang, "A cloud access security broker based approach for encrypted data search and sharing," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Jan. 2017, pp. 422–426.

[48] S. S. Chauhan, E. S. Pilli, R. C. Joshi, G. Singh, and M. C. Govil, "Brokering in interconnected cloud computing environments: A survey," *J. Parallel Distrib. Comput.*, vol. 133, pp. 193–209, Nov. 2019.

[49] S. Almakdi and B. Panda, "Secure and efficient query processing technique for encrypted databases in cloud," in *Proc. 2nd Int. Conf. Data Intell. Secur. (ICDIS)*, Jun. 2019, pp. 120–127.

[50] S. A. A. Almakdi, "Secure and efficient models for retrieving data from encrypted databases in cloud," Ph.D. dissertation, Univ. Arkansas, Fayetteville, AR, USA, 2020.

[51] R. A. Popa. (2014). *CrypDB*. [Online]. Available: https://github.com/CryptDB/cryptdb/

[52] A. Madkour. (2018). *Ope*. [Online]. Available: https://github.com/aymanmadkour/ope/

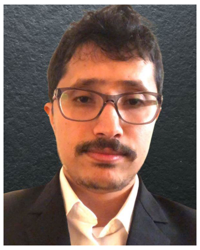
[53] *Paillier*. Accessed: Oct. 2019. [Online]. Available: https://www.csee.umbc.edu/~kunliu1/research/Paillier.html/

[54] *Package Javax Crypto*. Accessed: Oct. 2019. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html#package_description/

[55] J. Douglas, "Querying over encrypted databases in a cloud environment," M.S. thesis, Boise State Univ., Boise, ID, USA, Tech. Rep. 1520, 2019.

SULTAN ALMAKDI received the B.S. degree in computer science from King Khalid University, Abha, Saudi Arabia, in 2010, the M.S. degree in computer science from the University of Colorado Denver, Denver, USA, in 2014, and the Ph.D. degree in computer science from the University of Arkansas, Fayettiville, USA, in 2020. He is currently working as an Assistant Professor with the Department of Computer Science and Information Systems, Najran University, Saudi Arabia. His research interests include cloud security, fog security, edge computing security, the IoT security, and computer security. He received a Graduate Certificate in cybersecurity from the University of Arkansas, in 2020.

BRAJENDRA PANDA (Senior Member, IEEE) received the M.S. degree in mathematics from Utkal University, India, in 1985, and the Ph.D. degree in computer science from North Dakota State University, USA, in 1994. He taught mathematics in a four-year college in India. In 1988, he came to USA to pursue higher education in computer science. He taught undergraduate computer science at the College of West Virginia, Beckley, from 1993 to 1994. After completion of his Ph.D., he joined the Department of Computer Science, Alabama A&M University and he moved to the University of North Dakota, as a faculty of computer science, in 1997. He worked as a Research Fellow with the Rome Research Site of the Air Force Research Laboratory, from Summer 1997 to 1998. He joined the Computer Science and Computer Engineering Department, University of Arkansas, as an Associate Professor Fall Semester, in 2001. His research interests include database systems, computer security, and information assurance. He has published extensively in these areas and has received almost 2.5 million dollars in research funding. His research has been mostly supported by the National Science Foundation, Department of Defense, Air Force Office of Scientific Research, and Air Force Research Laboratory.



MOHAMMED S. ALSHEHRI (Graduate Student Member, IEEE) received the B.S. degree in computer science from King Khalid University, Abha, Saudi Arabia, in 2010, the M.S. degree in computer science from the University of Colorado Denver, Denver, USA, in 2014, and the Ph.D. degree in computer science from the University of Arkansas, Fayettiville, USA, in 2021. He received a Graduate Certificate in cybersecurity from the University of Arkansas, in 2020.

ABDULWAHAB ALAZEB (Graduate Student Member, IEEE) received the B.S. degree in computer science from King Khalid University, Abha, Saudi Arabia, in 2007, and the M.S. degree in computer science from the Department of Computer Science, University of Colorado Denver, USA, in 2014. He is currently pursuing the Ph.D. degree with the University of Arkansas, USA. His research interests include cybersecurity, cloud and edge computing security, and the Internet of Things.