Strongly Secure and Efficient Range Queries in Cloud Databases under Multiple Keys

Ke Cheng†‡, Yulong Shen†‡, Yongzhi Wang§, Liangmin Wang¶, Jianfeng Ma*†‡, Xionghong Jiang¶†‡ and Cuicui Su†‡

*Shaanxi Key Laboratory of Network and System Security, Xidian University, Xi’an, China
†State Key Laboratory of Integrated Service Networks (ISN), Xidian University, Xi’an, China
§School of Computer Science and Technology, Xidian University, Xi’an, China
¶Department of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China
∥School of Systems Information Science, Future University Hakodate, Hakodate, Japan
Email: kecheng@stu.xidian.edu.cn; ylshen@mail.xidian.edu.cn; ywang@park.edu; wanglm@ujs.edu.cn; jiang@fun.ac; jfima@mail.xidian.edu.cn; ccsu.hannah@gmail.com

Abstract—Cloud database provides an advantageous platform for outsourcing of database service. To protect data confidentiality from an untrusted cloud, the original database is often encrypted and then uploaded to the cloud. However, in order to support functional queries, existing secure databases require users to encrypt their data under the same public/symmetric key, which restricts the usage scenarios since users do not really trust each other in practice. Imagine a scenario where a user uploaded his/her own encrypted data to the cloud database and another user wants to execute private range queries on this data. This scenario occurs in many cases of collaborative statistical analysis where the data provider and analyst are different entities. Then either the data provider must reveal its encryption key or the analyst must reveal the private queries. In this paper, we overcome this restriction for secure range queries by enabling query executions on the multi-key encryption data. We propose a secure cloud database supporting range queries under multiple keys, in which all users could preserve the confidentiality of their own different keys, and do not have to share them with each other. At a higher level, our system is constructed on a two-cloud architecture and a novel distributed two-trapdoor public key cryptosystem. We prove that the proposed scheme achieves the goal of a secure query without leaking data privacy, query privacy, and data access patterns. Finally, we use extensive experiments over a real-world dataset on a commercial cloud platform to verify the efficacy of our proposed scheme.

I. INTRODUCTION

Advances in cloud computing have recently enabled users to outsource their data to clouds for large-scale data storage, management and query processing [1]. With recent incidents of massive data breaches [2], however, privacy concerns hinder the adoption of cloud paradigm [3] [4]. These concerns could be mitigated by encryption before outsourcing the sensitive data to clouds. The cloud-based encrypted database is now considered as a promising technique to process rich SQL queries on remote encrypted data [5]. Secure range query, as one of the most common database operations, has become widely supported in many cloud-based encrypted databases. CryptDB [6] is the initial exploration of secure cloud databases, which utilizes order-preserving encryption (OPE) to realize secure range query processes. A recent database Arx [7] achieves secure range queries by mutable OPE [8] with consideration of both security and efficiency. Since OPE is relatively weak in providing privacy assurance, Two-Cloud Secure Database [9] is proposed to minimize the leakage in range queries by eliminating the statistical properties. An alternative route is trusted hardware-based databases [10]–[12], which use trusted hardware to process range queries efficiently.

However, these works have all assumed a single key setting of databases, in which all clients are fully-trusted and share the same key for computability on encrypted data from multiple clients. This assumption does have manifest flaws. First, these encrypted databases can totally be broken once the unique key is leaked from any compromised client. Moreover, the clients in databases, as the lucrative targets for attackers, are clearly the weak links in the whole system. All of which led prior databases to be far from practice. The above studies inspire us to construct a Secure cloud Database under Multiple Keys for range queries, we call it MKSDB, in which each client holds his/her own unrelated key without sharing it with other clients or cloud servers. The clients can encrypt origin data or query conditions, and decrypt the results with their respective keys. Meanwhile, the range queries can be evaluated on the multi-key encryption data effectively. Consider a real world example for a health medical database below:

Example. A consortium of hospitals pools patient records in a cloud database for clinical research (for computing global characteristics), while each hospital prefers not to disclose the private data to other participates. The encrypted relation Patients(Name, Id, Pbbg, Plbg, Psbg)§ is shown as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Id</th>
<th>Pbbg</th>
<th>Plbg</th>
<th>Psbg</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_{pk_{1}} (Ada)</td>
<td>E_{pk_{1}} (45)</td>
<td>E_{pk_{1}} (202)</td>
<td>E_{pk_{1}} (338)</td>
<td>E_{pk_{1}} (210)</td>
</tr>
<tr>
<td>E_{pk_{1}} (Bee)</td>
<td>E_{pk_{1}} (15)</td>
<td>E_{pk_{1}} (206)</td>
<td>E_{pk_{1}} (274)</td>
<td>E_{pk_{1}} (170)</td>
</tr>
<tr>
<td>E_{pk_{1}} (Carl)</td>
<td>E_{pk_{1}} (3)</td>
<td>E_{pk_{1}} (192)</td>
<td>E_{pk_{1}} (292)</td>
<td>E_{pk_{1}} (92)</td>
</tr>
<tr>
<td>E_{pk_{1}} (Deb)</td>
<td>E_{pk_{1}} (25)</td>
<td>E_{pk_{1}} (239)</td>
<td>E_{pk_{1}} (437)</td>
<td>E_{pk_{1}} (362)</td>
</tr>
<tr>
<td>E_{pk_{1}} (Eda)</td>
<td>E_{pk_{1}} (6)</td>
<td>E_{pk_{1}} (215)</td>
<td>E_{pk_{1}} (328)</td>
<td>E_{pk_{1}} (147)</td>
</tr>
</tbody>
</table>

By taking advantage of MKSDB, each hospital first en-

§Pbbg: Pre-breakfast blood glucose measurement, Plbg: Pre-lunch blood glucose measurement, Psbg: Pre-supper blood glucose measurement.
crypted data records by its own key in local, then upload-s the encrypted data to cloud for constructing a federated database. One example of a range query can be: 

\[
\text{SELECT Id FROM Patients WHERE } \text{Pbbg} + \text{Pbbg} + \text{Pbbg} > \text{E}_{pk_1}(700), \text{ where } \text{E}_{pk_1}(x) > \text{E}_{pk_1}(y) \Rightarrow x > y. \]

That is, the hospitals are able to send encrypted queries under their own keys to the cloud database, and the query results can be decrypted only by themselves.

Our system is constructed on a two-cloud architecture and a novel distributed two-trapdoor public key cryptosystem. We first propose a basic secure-range-query protocol, which not only protects data privacy and query privacy, but also hides data access patterns. To improve the protocol’s performance, we employ data packing technique to speed up linear operations on ciphertext via batch processing, while reducing the usage of expensive operations to the minimum. In addition, our protocols support multi-client queries over the encrypted federated database, yielding a scalable solution to secure range query in practice.

For designing a range query protocol on multi-key encryption data, there are many challenges that need to be addressed. We have to achieve the protocol designs given an encryption scheme with semantic security, rather than property-preserving encryptions (e.g., OPE) with weaker privacy assurance. Furthermore, in the multi-client setting, a client can easily customize the query conditions to detect other participates’ data characteristics. So the data access pattern is necessary to be protected, e.g., the indices of results should not be revealed to other clients and clouds. In addition, the application of data packing requires a carefully-designed protocol to deal with the issue of possible overflows in the computations. We summarize our contributions as follows:

1) We study the secure-range-query problem on multi-key encryption data with semantic security for the first time. A secure cloud database under multiple keys – MKSDB is proposed to support the range query, in which all clients could preserve the confidentiality of their own different keys, and need not share them with other clients and clouds, which makes our system more practical.

2) We present a secure range query protocol, which exploits the additive homomorphism as well as distributed decryption to ensure the above function is achieved while providing strong security. Moreover, we adopt a data packing technique to greatly increase the efficiency of the basic protocol without sacrificing the strong security.

3) We prove that our protocols are secure under the semi-honest model through the proof-by-simulation method. We also build a prototype system running over the MySQL in C++, which verifies the efficacy of MKSDB through extensive experiments over a real-world dataset on a commercial cloud platform.

The rest of this paper proceeds as follows. Section II discusses the related work and Section III introduces the background definitions and our problem setting. The details of our system are presented in Section IV. Our system is analyzed theoretically and evaluated through extensive experiments in Section V and Section VI. Finally, Section VII concludes this paper and discusses potential future directions.

II. RELATED WORK

Secure range query in cloud databases has been extensively studied in recent years. The differences between our system and previous work are summarized in Table I. CryptDB [6] leverages order-preserving encryption (OPE) [8] to support SQL queries efficiently over encrypted data. In order to achieve ideal security, Arx [7] encrypts the data with mutable OPE [8] for range queries. To improve practicality, a recent scheme [13] is proposed to support range queries under multiple keys as ours. However, it only provides weaker privacy assurance based on OPE. To achieve semantic security in range queries, Two-Cloud Secure Database [9] is proposed, which introduces a series of protocols for more secure range queries based on Paillier encryption. For the scene of multiple clients, however, it still requires to share the same key with all clients. This is the gap that our work aims to fill.

The trusted hardware-based scheme is an important branch of cloud encrypted databases. TrustedDB [10] introduces the idea of adding a trusted hardware to the server, using the secure module to provide privacy-preserving query processes. Cipherbase [11] uses FPGAs to support rich SQL queries to execute in the cloud without providing visibility of database to the cloud. EnclaveDB [12] uses trusted hardware Intel SGX to support full functionality. These solutions have higher communication bandwidth and lower latency compared to transactional systems, thereby achieve competitive performance. Recently, Zheng et al. [14] present a distributed data analysis platform based on Spark SQL which can support a wide range of queries while hiding access patterns. However, these systems that use trusted hardware focus primarily on expanding functions or improving performance, they do not eliminate the limits caused by single key setting.

Our work is also related to the privacy-preserving computation tools under multiple keys. In [15], Peter et al. study outsourcing multi-party computation under multiple keys. They leverage an additively homomorphic encryption with double trapdoor decryption to build a secure computation scheme. Their design, however, suffers from the complex interactions between the two servers. In [16], Wang et al. show how to construct a computation platform over the ciphertexts of multiple clients. They combine the two-cloud model and proxy re-encryption to solve the single key setting problems. Nevertheless, this design does not support complex operations like the range operation. In addition, these works lack the scalability due to fairly complex processes (e.g., proxy re-encryption). With the further study, Liu et al. [17] propose a more comprehensive calculation toolkit with multiple keys to perform advanced operations such as division and comparison. However, performance becomes one critical limitation in its design. For example, a single execution of addition or multiplication would take a few seconds. Garbled circuits (GC) [18], as a universal secure multi-party computation tool, can easily
achieve the same function as our database for range query under multiple keys, through the experiments (See Section VI), we show its performance is far less than ours.

III. PRELIMINARY

A. Problem Statement

Consider multiple clients (e.g., hospitals, research institutions) who agree to construct a federated database relation \( R \) on a cloud server. For personal privacy and data security reasons, each client encrypts the objects by his/her own public key, then sends the encrypted data to cloud \( S_1 \). Therefore, \( S_1 \) maintains an encrypted relation \( ER \) corresponding to \( R \) collected from different sources. Assume \( R \) has \( n \) objects \( o_1, \ldots, o_n \) and each \( o_i \) has \( m \) attributes which are numerical values. We consider a SQL query \( q = \text{SELECT * FROM R WHERE } F_W(\cdot) > a \), where the function \( F_W(\cdot) \) is that weighted linear combinations on the attributes, i.e., \( F_W(o_i) = \sum w_j \cdot x_j(o_i) \), \( w_i \) is a user-defined weight and \( x_j(o_i) \) is the value of the \( j \)-th attribute for object \( o_i \). That is, a client wants to query the results that meet the range requirement based on the specific function scores. The client first encrypts the query by his/her own public key and sends it to the cloud server. The cloud server performs the query over the encrypted federated relation \( ER \) and returns the encrypted results to the client. Finally, the client can receive the encrypted results which can be decrypted alone only by his/her own key.

B. System Architecture and Security Model

Fig. 1 presents the system architecture of MKSDB. We assume the existence of two non-colluding semi-honest cloud providers, \( S_1 \) and \( S_2 \), where \( S_1 \) maintains an encrypted database contributed by multiple clients and \( S_2 \) provides crypto services. The two parties cooperate to answer range queries from multiple clients in a privacy-preserving manner. Note that the two-cloud architecture has been widely used in recent works [9], [19], [20]. In reality, these cloud servers are usually run by different competing companies, such as Amazon and Google, who are highly unlikely to collude with each other [21]. In addition, there is a trusted key generation center (KGC) in the system, which is only used to manage public/private keys. The KGC is not shown in Fig. 1 for brevity.

Security Model. In our security model, KGC is trusted to generate public/private keys and distribute these keys to each participant in the system. \( S_1 \) and \( S_2 \) are assumed to be curious and honest parties in the sense that they correctly follow the protocol specification but attempt to learn additional information by analyzing the transcript of messages received during the execution. Desired Security Properties. Our query protocols should be secure under the semi-honest adversaries model. Furthermore, Data privacy, query privacy, data access patterns should be protected, and the collusion between the cloud server and clients should be defended throughout the entire query scheme. Specifically, \( S_1 \) and \( S_2 \) should know nothing about the clear data of \( ER \) except the numbers of objects and attributes. Moreover, \( S_1 \) and \( S_2 \) should learn nothing about the query \( q \) except for a small amount of leakage that we explicitly describe. Access patterns to the data, such as the indices of the query results, should not be revealed to \( S_1 \) and \( S_2 \) to prevent any inference attacks. In cahoots with \( S_1 \) or \( S_2 \), the client cannot know the private information of other clients. Note that, the encrypted databases in existing works [6], [7], [9], [14] are incapable of defending this collusion attack, since the same key is shared among all the clients.

C. Cryptographic Tools

1) The Public-Key Cryptosystem with Distributed Decryption (PCDD): The two-trapdoor public key cryptosystem was first proposed by Bresson et al. [22]. Besides ordinary private key \( sk \) (aka weak key), there is a strong private key \( SK \) in the cryptosystem which can decrypt the encrypted data under different public keys. To reduce the risk of strong private key leakage, Liu et al. [17] adapted the cryptosystem to separate the strong key into two shares to enable distributed decryption. PCDD is defined as follows:

- **KeyGen:** Let \( N = pq \) and \( \lambda = \text{lcm}(p - 1, q - 1)/2 \), where \( p \) and \( q \) are large prime numbers, \( L(p) = L(q) = k \). Define a function \( L(x) = (x - 1)/N \), pick an element \( g \) of \( \mathbb{Z}_N^\star \) whose order is \( (p - 1)(q - 1)/2 \), then randomly select \( \theta_i \in [1, N/4] \) and compute \( h_i = g^{\theta_i} \mod N^2 \) for party \( i \). The public key is \( pk_i = (N, g, h_i) \), the corresponding weak private key is \( sk_i = \theta_i \). The strong private key is \( SK = \lambda \).
- **Encryption (Enc):** Given the public key \( pk_i \), one can encrypt a message \( m \in \mathbb{Z}_N \) by randomly choosing \( r \in [1, N/4] \) and computing \( m_{pk_i} = \{T_{i,1}, T_{i,2} \} \), where \( T_{i,1} = g^{r\theta_i}(1 + mN) \mod N^2 \) and \( T_{i,2} = g^r \mod N^2 \).
Decryption with weak private key (WDec): One can decrypt the ciphertext \([m]_{pk_i}\) with the weak private key \(sk_i = \theta_i\) by computing
\[
m = D_{sk_i}(m)_{pk_i} = L((T_{i,1} \cdot (T_{i,2})^{-1}) \mod N^2).
\]

Decryption with strong private key (SDec): Any ciphertext \([m]_{pk_i}\) can be decrypted using decryption algorithm \(D_{SK}(\cdot)\) with strong private key \(SK = \lambda\) by first calculating:
\[
T_{i,1} \mod N^2 = g^{\theta_0} \cdot (1 + mN\lambda) \mod N^2 = 1 + mN\lambda.
\]
Then, \(m\) can be recovered as:
\[
m = L(T_{i,1} \mod N^2) \cdot \lambda^{-1} \mod N.
\]

Strong private key splitting (SkeyS): The strong private key \(SK = \lambda\) can be randomly split into two parts. The partial strong private keys \(SK(j) = \lambda_j (j = 1, 2)\), s.t., \(\lambda_1 + \lambda_2 \equiv 0 \mod \lambda\) and \(\lambda_1 + \lambda_2 \equiv 1 \mod N^2\) hold at the same time.

Partial decryption with partial strong private key (PS-Dec): The algorithm \(D^+_{SK(j)}(\cdot)\) takes as input \(T_{i,1}\) and a partial strong private key \(SK(j) = \lambda_j\) then outputs the partial decrypted ciphertext
\[
CT_{i,j} = (T_{i,1})^{\lambda_j} = g^{\theta_0 \cdot \lambda_j} \cdot (1 + mN\lambda_j) \mod N^2.
\]

Decryption with partially decrypted ciphertext (DDec): The algorithm \(D^-(\cdot)\) takes as input \(CT_{i,1}\) and \(CT_{i,2}\) then computes
\[
m = L(CT_{i,1} \cdot CT_{i,2}).
\]

Ciphertext Refresh (CR): This algorithm can refresh the ciphertext \([m]_{pk_i}\) to get \([m]_{pk_i} = \{T'_{i,1}, T'_{i,2}\}\) without changing the original message \(m\) as follows: choose a random number \(r' \in \mathbb{Z}_N\), then compute
\[
T'_{i,1} = T_{i,1} \cdot h_i \mod N^2, T'_{i,2} = T_{i,2} \cdot g^{r'} \mod N^2.
\]
Note that the numbers involved in PCDD can be integers (i.e., \(m\) can be positive, negative or zero). However, the sign bit is lost in a modular operation. The method of how to handle this can be found in [17]. In addition PCDD cryptosystem is homomorphically additive:
\[
D_{sk_i}([x + y]_{pk_i}) = D_{sk_i}([x]_{pk_i} \cdot [y]_{pk_i} \mod N^2).
\]
\[
D_{sk_i}([a \cdot x]_{pk_i}) = D_{sk_i}([x]_{pk_i})^a \mod N^2.
\]
We can also find that \(T_{i,2}\) is only useful when performing weak decryption. For describing our protocols explicitly, we use \([m]^{(1)}_{pk_i}\) and \([m]^{(2)}_{pk_i}\) to denote \(T_{i,1}\) and \(T_{i,2}\), respectively. We summarize the notations throughout this paper in Table II.

2) Multi-Domain Secure Multiplication (MSM) Protocol: Suppose that \(S_1\) has two encrypted data \(E_{pk_i}(x)\) and \(E_{pk_i}(y)\), the goal of MSM protocol is to calculate \(E_{pk_i}(x \cdot y)\) without leaking \(x\) and \(y\) to the two servers. Here we skip the detail description of MSM Protocol (see [17] for details).

IV. SECURE RANGE QUERIES IN MKSDB

In this section, we propose MKSDB for secure-range-query based on the good cryptographic properties of PCDD.

A. Overview

To simplify our description of the protocol, we consider a simple case of two clients, one acts as a data contributor (Client \(C\)) and one acts as a data user (Client \(U\)), which is sufficient to demonstrate the general scenario. Firstly, the participants can get the desired public keys and private keys from the KGC. Then, Client \(C\) encrypts the records locally and uploads them to the database located in Cloud \(S_1\). Now an encrypted query can be issued by Client \(U\). After this, Cloud \(S_1\) and Cloud \(S_2\) cooperate to run secure query protocol and ultimately return encrypted results to the Client \(U\). Among all clients, only Client \(U\) is able to decrypt the results.

B. Database Initialization

The database initialization includes two steps: key generation and data encryption, which is depicted in Fig. 2.

1) Key Generation: Our construction is based on the aforementioned PCDD. The KGC generates a number of public/private key pairs \(pk_i/sk_i\) under the same public parameters in our system, then distributes them to each authorized clients. In our case, KGC sends the \(pk_c/sk_c\) to Client \(C\) (1) in Fig. 2) and \(pk_u/sk_u\) to Client \(U\) (2) in Fig. 2). In addition, the KGC splits the strong private key \(SK\) randomly into \(SK^{(1)}\) and \(SK^{(2)}\) by SkeyS algorithm, sending to \(S_1\) and \(S_2\) together with \(pk_c\) and \(pk_u\), respectively (3 and 4) in Fig. 2).

2) Data Encryption: After receiving \(pk_c/sk_c\) from KGC, Client \(C\) uses \(pk_c\) to encrypt his/her own relation \(R\). Assume that the number of objects and attributes in \(R\) is \(n\) and \(m\), respectively. For \(j\)-th attribute value \(x_j(\alpha_i)\) of object \(\alpha_i\), \(C\) encrypts it with \(pk_c\) as follows:

\[
ER_{i,j} \leftarrow [x_j(\alpha_i)]_{pk_c} = ([x_j(\alpha_i)]^{(1)}_{pk_c}, [x_j(\alpha_i)]^{(2)}_{pk_c}).
\]
then the encrypted relation \(ER\) can be denoted as \(ER = \{ER_{i,j}\}_{1 \leq i \leq n, 1 \leq j \leq m}\). After that \(C\) outsources the encrypted relation to \(S_1\) (5) in Fig. 2).
C. Secure Range Query Processing

In our scheme, query execution entails three stages: Query Request, Result Retrieve and Result Return. Next, we will describe the implementation details of each stage in turn.

1) Query Request: Consider that Client $\mathcal{U}$ issues the following query:

$$\text{SELECT } A_k \text{ FROM } R \text{ WHERE } F_W(\cdot) > a,$$

Where $F_W(o_i) = \sum w_j \cdot x_j(o_i)$ and $A_k$ indicates the $k$-th attribute name. The query condition $a$ should be encrypted into the ciphertext $[a]_{pk_u}$. If the weight $w_j$ is non-secret, $\mathcal{U}$ can send these weights to $\mathcal{S}_1$, and $\mathcal{S}_1$ simply computes the ciphertext $E_{F_k} = [F_W(o_i)]_{pk_e} = [f_i]_{pk_e}$ by the additively homomorphic property of the PCDD. Otherwise, $\mathcal{U}$ should send the encrypted weights $[w_i]_{pk_u}$ to $\mathcal{S}_1$, and then $\mathcal{S}_1$ can compute the ciphertext $E_{F_k}$ by the MSM protocol. We denote that $EF = \{[f_i]_{pk_e} \mid 1 \leq i \leq n\}$ and the encrypted query can be represented as follows:

$$\text{SELECT } A_k \text{ FROM } ER \text{ WHERE } E_{F(i)} > [a]_{pk_u}.$$  

2) Result Retrieve: To retrieve the $k$-th attribute values of all objects matching specific range in encrypted relation $ER$, $\mathcal{S}_1$ initiates Secure Query (SQ) protocol with $\mathcal{S}_2$. The basic idea of this protocol is based on the following feature: if $(f - a) \cdot l - \varepsilon > 0$, then $f > a$. Thereby, we construct our query protocol by homomorphically computing $(f - a) \cdot l - \varepsilon$ between two clouds. The main steps involved in SQ protocol are given in Algorithm 1.

First of all, Cloud $\mathcal{S}_1$ refreshes the ciphertext of $k$-th attribute values in $ER$ by CR algorithm to get $ER'_k = \{x_k(o_i) \mid 1 \leq i \leq n\}$, and shuffles the values in the $ER_k$ and $EF$ by a random permutation function $\pi$, such that the shuffled set $ER'_k = \{x_k(o_i(1)), \ldots, x_k(o_i(n))\}$ and $EF'' = \{[f_{\pi(1)}]_{pk_e}, \ldots, [f_{\pi(n)}]_{pk_e}\}$. For each value in $EF''$, $\mathcal{S}_1$ marks $f_{\pi(i)}$ with the random number $r_i \in \mathbb{Z}_N$. Such that $X_{\pi(i)} = (f_{\pi(i)} + r_i)_{pk_e}$, $Y_{\pi(i)} = [a + r_i]_{pk_e}$. Note that $\mathcal{S}_1$ chooses $l_i$ and $l_i \in \mathbb{Z}_N$, $0 \leq l_i < l$, then $f_{\pi(i)} > a$.

After receiving above ciphertexts, Cloud $\mathcal{S}_2$ decrypts them by $SK^{(2)}$ and recovers the plaintexts for getting $\alpha = (f_{\pi(i)} + r_i) \cdot l_i - l_i$ and $\beta = (a + r_i) \cdot l_i$. If the difference between $\alpha$ and $\beta$ is greater than 0, then $j = \pi(i)$ is the index of the satisfactory result. We mask the result value $x_k(o_j)$ with a random integer $u_j$, such that $S_j = [x_k(o_j) + u_j]_{pk_e}$. Then Cloud $\mathcal{S}_2$ partially decrypts it by $SK^{(2)}$ and inserts $(S_j, S''_j, [u_j]_{pk_u})$ into a vector array $L$. After all the result items are processed, $L$ is sent to $\mathcal{S}_1$. Assume that the number of result items is $n'$, i.e., $|L| = n'$. For each item in $L$, $\mathcal{S}_1$ recovers the plaintext $T_j = x_k(o_j) + u_j$ by $SK^{(1)}$. After that, $\mathcal{S}_1$ encrypts $T_j$ using Client $\mathcal{U}$’s public key $pk_u$ to get $[T_j]_{pk_u}$, then the mask in the record $[T_j]_{pk_u}$ is removed through multiplying by $([u_j]_{pk_u})^{N-1}$. That is, $\mathcal{S}_1$ outputs the results encrypted by $pk_u$. Noted that the security of Algorithm 1 is mainly dependent on the degree of confusion from the random number $r_i$. Suppose the size of $f_i$ is $\sigma$ bits and $r_i$ is $\sigma + \kappa$ bits, the statistical security of the $f_i + r_i$ for $f_i$ is approximately $2^{-\kappa}$. In fact, the value of $\kappa$ can be viewed as a security parameter.

3) Result Return: When receiving the encrypted result $[x_k(o_j)]_{pk_e}$ from Cloud $\mathcal{S}_1$, Client $\mathcal{U}$ decrypts it using his/her own private key $sk_u$ by $\text{BDec}$ algorithm to get the result.

D. Efficiency Improvement by Data Packing

In the terms of efficiency, SQ protocol suffers from a large number of expensive operations. To improve performance, we
use the data packing technique [23] to renovate our protocol, which can speed up linear operations on ciphertext via batch processing. There are two key observations behind this improvement: 1) A lot of common operations (e.g., encryption, decryption and multiplication) are performed repetitively. 2) The message space of PCDD, which is typical 2048-bits for a sufficient level of security, is not fully utilized as the realistic attributes typically assume small domains (e.g., 0 ≤ blood glucose ≤ 1000). The rationale of data packing is to pack the values in $ER_k$ and $EF$. Pick $n$ random integers $r_i$ ($1 \leq i \leq n$, $\mathcal{L}(r_i) = \sigma + \kappa$), pack those integers to obtain: $[r_1 | r_2 | \ldots | r_n]$, encrypt the package by $pk_c$ and $pk_u$, to obtain: $[r_1 | r_2 | \ldots | r_n]_{pk_c}$ and $[r_1 | r_2 | \ldots | r_n]_{pk_u}$.

5. $X_{Packed} \leftarrow [(f_{\pi(1)} + r_1) \ldots (f_{\pi(n)} + r_n)]_{pk_c}$, $Y_{Packed} \leftarrow [(a + r_1) \ldots (a + r_n)]_{pk_u}$.

6. Pick random integers $l, \epsilon_l (1 \leq l \leq n, \mathcal{L}(l) = \mathcal{L}(\epsilon_l) = \kappa)$, $0 \leq \epsilon_l < l$, and compute $E_{Packed} = [\epsilon_1 | \ldots | \epsilon_n]_{pk_c}$.

7. $P_{Packed} \leftarrow (X_{Packed})^\epsilon_l \cdot (E_{Packed})^{n-1}$, $Q_{Packed} \leftarrow (Y_{Packed})^\epsilon_l$.

8. $P_{\prime Packed} \leftarrow D_{S K_c}^{-1}(P_{Packed})$, $Q_{\prime Packed} \leftarrow D_{S K_u}^{-1}(Q_{Packed})$.

9. Send $P_{\prime Packed}, P_{\prime Packed}, Q_{\prime Packed}$ and $[x_k(o_\pi(i))]_{pk_c}$ ($1 \leq i \leq n$) to $S_2$.

Update 1: A lot of common operations (e.g., encryption, decryption and multiplication) are performed repetitively.

Update 2: 1) To handle possible overflows in the addition operation (need 1 bit) and subsequent multiplication operations (need $\kappa$ bits). In a similar way, $Y_{Packed} = [r_1 | r_2 | \ldots | r_n]_{pk_u}$. Since $S_1$ generates a random number $l$ and an encrypted random number package $E_{Packed}[\epsilon_1 | \epsilon_2 | \ldots | \epsilon_n]_{pk_c}$, and computes $P_{Packed} = (X_{Packed})^\epsilon_l \cdot (E_{Packed})^{n-1} = [(f_{\pi(1)} + r_1) \cdot \ldots \cdot (f_{\pi(n)} + r_n)]_{pk_c}$.

Next, $S_1$ generates a random number $l$ and an encrypted random number package $E_{Packed}[\epsilon_1 | \epsilon_2 | \ldots | \epsilon_n]_{pk_c}$, and computes $P_{Packed} = (X_{Packed})^\epsilon_l \cdot (E_{Packed})^{n-1} = [(f_{\pi(1)} + r_1) \cdot \ldots \cdot (f_{\pi(n)} + r_n)]_{pk_c}$. Then, $P_{\prime Packed}$ and $Q_{\prime Packed}$ are partially decrypted by $S K_c$ and these ciphertexts together with $[x_k(o_\pi(i))]_{pk_c}$ ($1 \leq i \leq n$) are sent to Cloud $S_2$ as shown in line 9. After that, Cloud $S_2$ decrypts the received data to get $\alpha$ and $\beta$. Next, $\alpha$ is treated as bit-strings and split into $n$ values of equal bits length $\alpha_\pi(i) = (f_{\pi(i)} + r_i) \cdot l - \epsilon_i$ ($1 \leq i \leq n$). And the values $\beta_\pi(i) = (a + r_i) \cdot l$ are calculated in the same manner. In the remaining part of Algorithm 2, data packing is also applied to obtain $[x_k(o_\pi(i))]_{pk_c}$.

For the unpacking operation, we can achieve it efficiently by bitwise operations. Note that for readability we assume any number of values in our protocols can be packaged into one message, when in fact, the amount of data in one packet is limited. We will analyze the factors which affect the required number of the packages later.

Next, we describe the Data Packing based Secure Query (DPSQ) protocol as shown in Algorithm 2. For brevity, we only show the parts that need to be updated. In line 4 of Algorithm 2, Cloud $S_1$ generates a set of $\sigma + \kappa$ bits random numbers $r_1, \ldots, r_n$, then use Equation 1 to pack them as $[r_1 | r_2 | \ldots | r_n]$. After that, the package is encrypted by $pk_c$ and compute $X_{Packed}$ as follows:

$$X_{Packed} \leftarrow [(f_{\pi(1)} + r_1) | (f_{\pi(2)} + r_2) | \ldots | (f_{\pi(n)} + r_n)]_{pk_c}$$

$$= [r_1 | r_2 | \ldots | r_n]_{pk_c} \prod_{i=1}^{n} [(f_{\pi(i)})_{pk_c}]^{2(\pi(i) + \kappa + l)}$$

Note that, $\sigma + 2\kappa + 1$ bits instead of $\sigma + \kappa$ bits are allocated for $f_{\pi(i)} + r_i$ to handle possible overflows in the addition operation (need 1 bit) and subsequent multiplication operations (need $\kappa$ bits). In a similar way, $Y_{Packed} = [r_1 | r_2 | \ldots | r_n]_{pk_u}$.
V. Theoretical Analysis

A. Security Analysis

We prove the security of our protocols using a formal security definition in semi-honest adversary model [24].

Definition 1 ((Security against semi-honest adversaries). Suppose that a two-party protocol $P$ asks $A$ to compute the function $f_A(x, y)$, and asks $B$ to compute $f_B(x, y)$, where $x, y$ are the inputs of $A$ and $B$, respectively. The view of $A$ (resp. $B$) during an execution of $P$ on $(x, y)$, denoted $\text{view}_A(x, y)$ (resp. $\text{view}_B(x, y)$), is $(x, r_A, m_1, \ldots, m_i)$ (resp. $(y, r_B, m_1, \ldots, m_i)$), where $r_A$ (resp. $r_B$) represents randomness of $A$ (resp. $B$) and $m_i$ represents the $i$-th message passed between the parties. Also let $O_A(x, y)$ and $O_B(x, y)$ denote $A$'s (resp. $B$'s) output. We say that protocol $P$ is secure against semi-honest adversaries if there exist probabilistic polynomial time (PPT) simulators $\text{Sim}_1$ and $\text{Sim}_2$ such that:

\[
\begin{align*}
(\text{Sim}_1(x, f_A(x, y)), f(x, y)) &\equiv (\text{view}_A(x, y), O(x, y)) \\
(\text{Sim}_2(y, f_B(x, y)), f(x, y)) &\equiv (\text{view}_B(x, y), O(x, y))
\end{align*}
\]

where $\equiv$ denotes computational indistinguishability.

We note that public-key cryptosystem with distributed decryption (PCDD) is semantically secure and Multi-Domain Secure Multiplication (MSM) protocol is secure under the semi-honest adversaries model. The formal security proofs of them can be found in [17]. Recall that the Cloud $S_1$ and Cloud $S_2$ are assumed to be semi-honest, and the two servers do not collude. SQ protocol is essentially a two-party computation protocol between $S_1$ and $S_2$, and its security can be stated as Theorem 1 under the above definition.

Theorem 1. As long as PCDD is semantically secure, SQ protocol is secure against semi-honest adversaries.

Proof. (Sketch) The function of SQ is that $S_1$ inputs an encrypted relation $ER$ and encrypted query conditions $EF, [a]_{pk_{\alpha}}$, then $S_1$ outputs the required results encrypted by $pk_{\alpha}$. As for security, we construct simulators in two distinct cases as SQ protocol is asymmetric for two parties. For each step, we show that for all PPT adversaries, the corrupted party’s view based on $S_1$ and $S_2$’s interaction is indistinguishable to its view when it interacts with a simulator instead (i.e., Equation (3) and (4) hold for each phase). Then, we can prove the security of the entire protocol.

Due to the space limit, we omit the formal security of DPSQ protocol. It is the same as SQ protocol’s as the basic idea of them are consistent. Next, we demonstrate that MKSDB can meet the desired security requirements, which is trivially guaranteed by the security of our protocols. (1) Data privacy: First of all, we recall that the clients encrypt the relations by $\text{Enc}$ algorithm locally before outsourcing them to the cloud. Since SQ protocol and DPSQ protocol are secure against the semi-honest adversaries, no information is disclosed to $S_1$ and $S_2$, data privacy can be preserved. (2) Query privacy: The query conditions $EF = \{[f_i]_{pk_{\alpha}}\} (1 \leq i \leq n)$ are computed by the additively homomorphic property of the PCDD or MSM protocol, and the condition $a$ is also encrypted by $\text{Enc}$ algorithm at the client before sending to the cloud. In addition, the secure query protocols are secure against the semi-honest adversaries. Therefore, $S_1$ and $S_2$ obtain no information about $EF$ and $a$ except for the query field (e.g., $A_k$), the query privacy is preserved. (3) Hiding data access patterns: During the query stage, $S_2$ could deduce the required index $j = \pi(i)$ and the index is shuffled by a random permutation function $\pi$, so $S_2$ does not know $[x_\pi(o_i)]_{pk_c}$’s position in the original relation. Since $S_1$ only get the ciphertexts and randomized plaintexts, $S_1$ does not know which data item belongs to the results. Thus, data access patterns are protected from both $S_1$ and $S_2$. (4) Defending collusion attacks: All data and query conditions are encrypted by the clients’ own different public keys before being sent to the cloud. Any client, in cahoots with $S_1$ or $S_2$, cannot get other clients’ weak private key or recover the strong private key, so the client cannot know the private information of other clients. In conclusion, the desired security properties in MKSDB can be achieved.

B. Complexity Analysis

1) Computational Complexity: In this part, we summarize the computational complexity of our protocols in Table III.

The computational overhead mainly depends on the cost of time-consuming algorithms and operations over the encrypted data, which can be categorized into four types: encryptions, decryptions, multiplications and exponentiations. Specifically, the encryption includes $\text{Enc}$ algorithm and $\text{CR}$ algorithm (as the cost of $\text{CR}$ algorithm is similar to the one of $\text{Enc}$ algorithm), decryptions includes $\text{PSDec}$ and $\text{DDec}$ algorithm.

For SQ Protocol in Algorithm 1, encryptions need to be done $n$ times (line 2). From line 4 to line 9, $3n$ encryptions, $2n$ decryptions, $3n$ multiplications and $3n$ exponentiations are performed to get $P_{\pi(i)}^j$ and $Q_{\pi(i)}^j$. From line 11 to line 20, $2n$ encryptions, $4n + n'$ decryptions and $n'$ multiplications are required for getting $L$. Besides, $2n'$ decryptions, $n'$ multiplications and $n'$ exponentiations are performed in line 24 and line 25 to remove the random factors from the ciphertexts. Therefore, SQ protocol totally requires $4n + 2n'$ encryptions, $6n + 3n'$ decryptions, $3n + 2n'$ multiplications and $3n + n'$ exponentiations. As for DPSQ protocol in Algorithm 2, it totally requires $n + 2T_1 + T_2 + 3T_3$ encryptions, $3T_1 + 3T_2 + 3T_3$ decryptions, $2n + n' + 2T_1 + 2T_2 + 2T_3$ multiplications and $2n + n' + 2T_1 + T_2 + T_3$ exponentiations by a similar analysis.

2) Communication Complexity: In the PCDD, each $[m]_i^{1, \alpha}$, $[m]_i^{2, \alpha}$ and $CT_{i, j}$ require $2|N|$ bits to be represented and the ciphertext $[m]_i^{\alpha}$ needs $4|N|$ bits. Thus, it takes $10n|N| + 8n'|N|$ bits between Cloud $S_1$ and Cloud $S_2$ to run SQ.
protocol, and $4T_1|N| + 4T_2|N| + 8T_3|N| + 2n|N|$ bits to run the DPSQ protocol.

VI. PERFORMANCE EVALUATION

A. Experiment Setup

We implement MKSDB in approximately 3000 lines of C++ code on the top of MySQL which is a widely used database system. Our system only calls MySQL C++ interface without modifying the kernel. We used the GMP Library version 6.1.2 for big integer operations. For the PCDD, we set $N$ as 2,048 bits ($\Gamma = 2048$) and statistical security parameter $\kappa = 40$ to achieve 112-bit security levels [25]. The experiments are conducted on the two Amazon EC2 c5d.2xlarge machines (as $S_1$ and $S_2$) running Linux 16.04, with 8-core Intel Xeon at 3.0 GHz and 8 GB of RAM each, and a PC (as clients) running Linux 16.04, with Intel E8400 CPU and 4 GB of memory. We ran our experiments under two network settings: LAN setting (two machines in the same region) and WAN setting (one machine located in Korea and another in the USA), the network bandwidths of them are about 45 Mbps and 88 Mbps. We use the diabetes dataset from UCI ML Repository, and customize it as the patient table (omit the name) in our Example (in Section 1), which contains 5000 objects and 4 attributes (i.e., $n = 5000$ and $m = 4$). We assume these records are contributed by one client and another client issues the range query statement (as shown in our Example$^3$).

To make a comprehensive performance evaluation, we compare our protocols with an advanced secure-range-query scheme [9] based Paillier cryptosystem (denote as P-SRQ), which also adopts the two-cloud structure but does not support multiple keys. In addition, we also make a comparison with the garbled circuit (denote as GC-SRQ) by using a state-of-the-art framework EMP-toolkit [26]. Cloud $S_1$ and Cloud $S_2$ can be viewed as the circuit generator and evaluator, and hold the random shares of inputs and outputs. All the above protocols are implemented using eight parallels. We collected 20 runs for each result point in our experiment and report the average.

B. Performance of Database Initialization

The database initialization consists of two steps: key generation and data encryption. We first evaluate the performance of key generation, it only takes 11 milliseconds to generate two partial strong private keys, and 10 milliseconds to generate a weak private-public key pair, which makes our system scalable for a large number of clients. For the above real-world dataset ($n = 5000$ and $m = 4$), data encryption under PCDD takes about 70 seconds. But our scheme still enjoys the practical performance as this is only a one-time cost.

C. Performance of Secure Range Query

1) Query Response Time: As a most critical performance indicator to evaluate the query scheme, query response time measures the interval between issuing a query and receiving the result at the client. In Fig. 3, we compare the query response time of the related schemes and our protocols for varying number of objects ($n$) and bit length of values($\sigma$) under both LAN and WAN settings. As shown in Fig. 3a and Fig. 3b, when fixing $\sigma = 30$, the query response time of all the protocols both grow as $n$ increases in LAN and WAN settings, since the number of time-consuming cryptographic primitives is positively associated with $n$. We could also find that our DPSQ protocol is the most efficient of all protocols. For example, when $n = 5000$, SQ, P-SQR and GC-SRQ protocols require 55, 26 and 15 seconds respectively, while it only takes 3.7 seconds in DPSQ protocol. This is because the data packing significantly reduce the number of expensive operations over the encrypted messages, which agrees with the computational complexity derived in Section V-B1. In the WAN setting, the gap between GC-SRQ and DPSQ protocol will become even larger, as GC-based solution needs the considerable interactions and communication costs, while our protocol has less sensitivity to the network bandwidth due to the relatively few communication costs.

Besides the number of records $n$, the bit length of record ($\sigma$) also affects the performance of some protocols. As shown in Fig. 3c and Fig. 3d, when fixing $n = 5000$, the query time of SQ and P-SRQ are nearly invariable with different bit lengths, while the time in GC-SRQ and DPSQ grow slightly as the bit lengths of values increases. It is because that the increased bit length would lead to a larger size of the garbled circuit, and for DPSQ protocol, it would increase the required number of the packages. However, our DPSQ protocol still enjoys the best performance of all the protocols. Specifically, the query time changes from 3.27/3.59 to 4.24/4.60 seconds as $\sigma$ varies from 10 to 50 in the LAN/WAN setting.

2) Communication Cost: We compare the communication costs of all protocols in Fig. 4. As shown in the Fig. 4a, when $\sigma = 30$, the communication costs of all protocols increase at a liner speed as the number of objects ($n$) increases. For the same reason explained above, data packing technique

\textsuperscript{3}http://archive.ics.uci.edu/ml/datasets.html
\textsuperscript{3}For this query in our test database, the numbers of results $n'$ are 95, 231, 375, 457, 610 when $n$ varies from 1000 to 5000.
heavily reduces DPSQ protocol’s communication cost. The communication cost of DPSQ protocol changes from 2.93 to 2.99 seconds as \( n \) varies from 1000 to 5000, which is equivalent to only 22% of the basic SQ protocol. In Fig. 4b, the communication costs of SQ and P-SRQ are constant as \( \sigma \) increases. For DPSQ protocol, the cost only grows slightly while the increased bit length results in a remarkable growth of communication cost in GC-SRQ. Due to the very low communication cost base, DPSQ protocol can provide a competitive performance of all protocols with increased \( \sigma \).

VII. CONCLUSION

In this paper, we proposed MKSDB, a secure cloud database for efficient range queries under multiple keys. This system is designed to allow multiple clients to construct a federated encrypted database, for the clouds to securely execute range queries on the multi-key encryption data. To achieve this goal, we presented a secure query protocol based on a novel distributed two-trapdoor public key cryptosystem (PCDD). On that basis, a performance-improved protocol with the application of data-packing is proposed which can considerably reduce the query time and improve the practicality a lot. Through the proof-by-simulation method, our protocols are proven secure under the semi-honest adversary model. Moreover, data privacy, query privacy and data access patterns are protected. Finally, we show the performance of our design by comparing with state-of-the-art works through extensive experiments conducted on a real-world dataset on Amazon clouds. In the future, we will continue to improve query performance over a larger database and extend our system to support more varied queries.

ACKNOWLEDGMENT

The corresponding author is Yulong Shen. The authors thank the anonymous reviewers for their valuable feedback. This research was supported in part by China NSFC Grants U1536202, U1736216, 61571352 and 61602364, Shaanxi Science & Technology Coordination & Innovation Project 2016KZTDGY05-07-01, Natural Science Foundation of Shaanxi Province Grants 2017JM6083.

REFERENCES