PROJECT REPORT

NAME: DARSHIT BIMAL GANDHI

ROLL NO: 22M0824

PROJECT TITLE: Simulating Secure KNN Computation Concepts with Python

CS 741: Advanced Network Security and Cryptography

1. INTRODUCTION

In order to reduce the strain on local storage and computer resources, cloud computing has become popular as a trend of outsourcing database and query services to a strong cloud. Nevertheless, there is a chance that secret and private data will be compromised while using cloud computing and private data storage. As a result, before being stored in the cloud, important apps must first be encrypted. Additionally, the data must be in the encrypted domain's computation-friendly ciphertext form in order to run various data mining methods, such as k-NN. Data is therefore encrypted before being outsourced to the cloud using a searchable encryption algorithm.

In this project, I have implemented **secure KNN algorithm** that can be directly performed on the encrypted data on the cloud and I have used **Asymmetric Scalar-product-Preserving Encryption (ASPE)** to implement the key-generation, data encryption, query encryption, and secure KNN. By **lowering the query encryption time using ASPE**, the encryption strategy becomes more practical and efficient. Additionally, it aids in **achieving query secrecy and data privacy**.

2. IMPLEMENTATION DETAILS

In my implementation, I have a total of seven Python files, namely main.py, key_gen.py, data_enc.py, query_enc.py, data_enc_for_user.py, data_dec.py and knn.py

2.1. **key_gen.py.** This file contains the implementation of how the Data Owner (DO) generates the data encryption key. I have one function in this file:

- **key_generation(n, d, c, epsilon, eta)**:
  This procedure samples the base secret matrix M_{base} of $\eta \times \eta$-dimension. Furthermore, a long-term secret vector s of length (d + 1) is also generated. This method also generates a fixed vector w of length c uniformly at random for each data vector to be encrypted. The secret key of the encryption scheme is (s,M_{base}, w) and is returned to main.py.

2.2. **data_enc.py.** This file contains one function:

- **data_encryption(data_vectors, secret_key, public_params)**:
  This procedure generates an ephemeral secret vector z of length ϵ. Then it will create the encrypted version of the data vectors as follows: $\tilde{p}_i = (s_1 - 2p_{i1}, ..., s_d - 2p_{id}, s_{d+1} + ||p_i||^2, w, z)$ Then this procedure will form pi' as pi' = $\tilde{p}_i . M_{base}^{-1}$. This method also computes the Euclidean norm (EN) of data vectors and stores the maximum norm as Max_norm = Max(EN(p1),..., EN(pm)), which will be used for query encryption later.

2.3. **query_enc.py.** This file contains the three-step query encryption algorithm. There are basically three functions:

- **step1_query_encryption(query_vector, user_id, beta1, d):** This procedure samples a diagonal matrix N_{dXd} of real numbers and computes the encrypted query q$^{\cdot}$ as q$^{\cdot}$ = β_1 · q · N.
- **step2_query_encryption(encrypted query, Max_norm, M_{base}, c, epsilon, eta, beta2):** This procedure first computes the largest number present in the encrypted query vector q$^{\cdot}$ as q_{max} = max(q$^{\cdot}$). Next, it samples a temporary secret matrix $M_t^{\eta X \eta}$ with all elements except the diagonal elements of matrix M_t uniformly at random larger than q_{max}. In contrast, the diagonal elements are sampled uniformly at random larger than Max_norm. Later, matrix M_t is multiplied with M_{base} to obtain a temporary secret matrix for the query encryption as $M_{sec} = M_t \cdot M_{base}$. The obtained query vector q$^{\cdot}$ is appended to obtain a new query vector $q^{'}$ of η-dimension as $q^{'}$ = (q$^{\cdot}$, 1, x, 0_ϵ). Here x is an ephemeral integer vector of length c and 0_ϵ is a zero vector of length ϵ. Next, it uses the operator O to convert vector $q^{'}$ into a $\eta \times \eta$ dimensional diagonal matrix $q_{\eta\eta}$ as follows O : $q^{'} \longrightarrow q_{\eta\eta}$ with $q_{ii} = q_i^{'}$. Next, it performs computation $q\hat{} = \beta_2(M_{sec}q_{\eta\eta} + E)$. Elements of the error matrix E are sampled uniformly at random larger than q_{max}. This method returns the doubly encrypted query matrix $q\hat{}$ to the QU and sends the temporary secret matrix M_t and user ID to the CSP as (user id,M_t).
- **step3_query_decryption(doubly_encrypted_query, d, N, eta):** This procedure removes QU's encryption layer to obtain $\tilde{q}_{enc}$. It constructs a diagonal matrix $N^{'}_{\eta X \eta}$ with first d diagonal elements same as that of the

matrix N_{dXd} and rest all 1. It will then compute its inverse and use it to remove QU's encryption layer as follows: $q\tilde{}_{enc} = q\hat{} \cdot N'^{-1}$. Then it will create the $q\tilde{}_{vec}$ such that the j^{th} element of vector $q\tilde{}_{vec}$ is obtained by adding all columns of row j of matrix $q\tilde{}_{enc}$.

2.4. **data_enc_for_user.py.** This file contains one function:

- **convert_encrypted_data_for_user(encrypted_data, M_t):** For each user, this function will create a temporary database as $_ip'' = _jp' \cdot M_t^{-1}$.

2.5. **knn.py.** This file contains two functions:

- **compute_knn_encrypted_custom_k(q_tilda_vector, encrypted_data_for_user, user_id, k):** This function will compute dot product = $p''_i \cdot q\tilde{}_{vec}$ and will check for which p''_i will it give the least dot product. Then it will return the first k p''_i after sorting it according to the dot-product.
- **compute_knn_unencrypted_custom_k(query_vector, data_vector, user_id, k):** This function will calculate the Euclidean norm of the un-encrypted query vector and unencrypted data vectors and will check for which data_vector will it give the least norm. Then it will return the first k data_vectors after sorting it according to the norms.

2.6. **data_dec.py.** This file contains one function:

- **data_decryption(encrypted output, Mt, Mbase, secret_key, d):** In this procedure, we need to decrypt this p''_i that we got from knn.py to get the original pi, so for this, we will perform the reverse operations that we have performed in data_enc.py and data_enc_for_user.py

3. WORKFLOW OF THIS IMPLEMENTATION

The security model of secure k-NN computation consists of three entities: a data owner (DO), a cloud service provider (CSP), and a group of query users (QU). The Query User (QU) wants to query the database that is sent by the Data Owner (DO) to the Cloud Service Provider (CSP). But we want to do this in an encrypted version. The following steps specifies the workflow of this algorithm:

(1) First, DO will generate its secret key using the **key_generation** function. Then DO uses a searchable encryption algorithm similar to the function **data_encryption** to encrypt its database before storing it in the cloud.

(2) A QU encrypts its query vector using the encryption algorithm **step1_query_encryption** with the intention of computing the k-NN of its query vector and then will send this encrypted query to Data Owner (DO) for query re-encryption by the DO's secret key.

(3) When DO receives the query vector from a QU, it uses the **step2_query_encryption** procedure to re-encrypt it before sending it back to the QU.

(4) As stated in the function **step3_query_decryption**, QU removes its encryption layer upon receiving a doubly encrypted query vector from the DO and forwards the encrypted query vector to the CSP for k-NN computation.

(5) CSP verifies the user ID and calculates k-NN using the relevant data as shown in the function **compute_knn_encrypted_custom_k**. The derived k-NN solution is then sent to the QU. But this solution is currently encrypted.

(6) **data decryption** procedure is used to decrypt the encrypted data vectors that we got from the procedure compute knn encrypted custom k

4. Instruction for running the code

All helper files are been included in the main.py. So, we only need to run **"python3 main.py"**.

5. SYSTEM SPECIFICATIONS

I have tested this algorithm on Windows and Linux system, having 8 GB RAM, Intel Core i5-8250U CPU, 1.60 GHz frequency octa-core processor.

REFERENCES

[1] Secure KNN Computation On Cloud by Tikaram Sanyashi, Nirmal Kumar Boran, and Virendra Singh