Privacy Preserving Cloud Computing

A Dissertation
Submitted in partial fulfillment of
the requirements for the degree of
Master of Technology
by

Aditya Jain
(Roll No. 203050003)

Supervisor:
Prof. Bernard Menezes

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

29 June 2022

Approval Sheet

This dissertation entitled "Privacy Preserving Cloud Computing" by Aditya Jain is approved for the degree of Master of Technology.

____

____

____

Examiners

____

____

____

Supervisor (s)

____

Chairman

Date: __________

Place: __________

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Aditya Jain
(Roll No. 203050003)

Date: 29 June 2022

Abstract

To leverage the benefits of Cloud Computing, Data Owners outsource their data management systems from their local sites to the cloud servers. Benefits include quick deployment, cost reduction, scalable storage, on-demand services from a pool of computing resources, etc. To ensure data privacy, Data Owners store their data in encrypted form in the cloud. Unfortunately, conventional encryption schemes don't support the analysis and processing of encrypted data. To make some cash, Data Owner may allow others to query his data. To ensure query privacy, a query user may want to send queries in encrypted form. Now the cloud has to perform computations on encrypted data and encrypted queries. Earlier works have proposed techniques to securely compute k-nearest neighbors (k-NN) on encrypted data (outsourced to the cloud server). These works have used Homomorphic Encryption. We identified vulnerabilities in previous works like Zhu *et al.* (2016), Singh *et al.* (2018), and Parampalli *et al.* (2019) that cannot be handled by homomorphic encryption alone, like code integrity, code confidentiality, etc. We propose a new scheme using secure enclaves to circumvent these vulnerabilities while improving performance. Our proposed design provides data privacy, query privacy, key confidentiality, query controllability, data integrity, code integrity, access pattern hiding, and defense against collusion attacks. Our scheme also improves performance by eliminating Paillier encryption and multiple matrix multiplications, which takes too much time. Our scheme can also be extended to provide code confidentiality.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud Computing has become very popular in recent years. It offers benefits like quick deployment, scalable storage, cost reduction, etc. These benefits attract Data Owners having a large amount of data and computations to outsource their data management systems from local sites to the cloud servers. But then the direct physical control of their data goes to the cloud server. So to ensure data privacy, Data Owner (DO) generally store their data in encrypted form on the Cloud Server (CS). Unfortunately, most conventional schemes don't support the analysis and processing of encrypted data (outsourced to a cloud server). Data Owner may also wish to earn money by allowing users' queries on his (Data Owner's) data. Query User may want to send his query in encrypted form to have query privacy. Now the cloud has to perform computations on an encrypted query and encrypted data.
The k-nearest neighbor has been used as a fundamental database query operation in many scenarios like Profile Matching in a Social Networking Database, Encrypted Keyword Search, etc. Generally, there are three parties involved (Fig. 1.1):

1. Data Owner (DO)
2. Query User (QU)
3. Cloud Server (CS)

Earlier works like Zhu *et al.* (2016) work claimed these four properties: Database Privacy, Query Privacy, Key Confidentiality, and Query Controllability. Singh *et al.* (2018) paper proved that Zhu *et al.* (2016) scheme lacked Query Controllability and they modified Zhu *et al.* (2016) work to provide Query Controllability and Query Check Verification (in addition). Later, Parampalli *et al.* (2019) proposed a scheme that achieves Data Privacy, Query privacy, Key Confidentiality, and Access Pattern Hiding but lacked

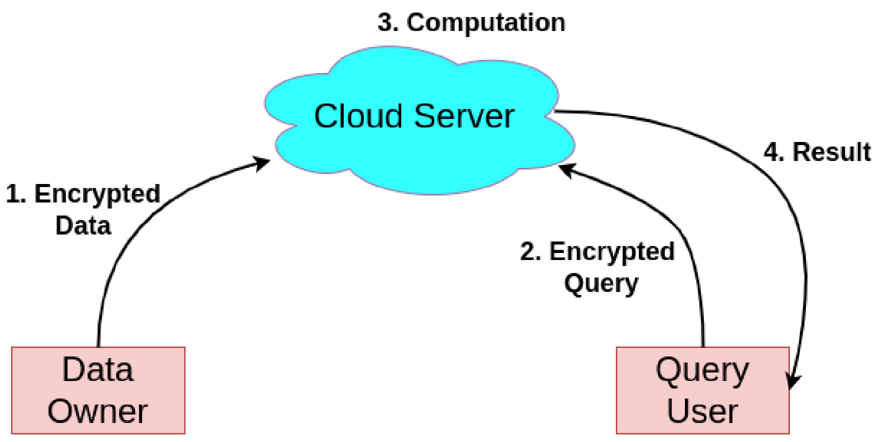


Figure 1.1: Parties involved in Secure Cloud Computing

Query Controllability. The following section discusses the motivations for our work in detail.

1.1 Motivation

We studied several applications in the area of AI / ML implemented on the cloud servers, with security as the paramount goal. These works attempted to address the challenge of providing data and query privacy. In particular, they enabled an encrypted query to be processed against an encrypted database using Homomorphic Encryption schemes. However, we found several lacunae in their design.

1.1.1 Security Motivation

1. Earlier work like Zhu *et al.* (2016) scheme claimed to provide Data Privacy, Query Privacy, Key Confidentiality, and Query Controllability. There have been attacks on their scheme, which broke down all the four security requirements they promised.

2. Earlier work like Zhu *et al.* (2016) scheme used the same matrix M to encrypt the queries of all query users. But we can attack their scheme to get M (Through Attack 2 discussed in Chapter 3). With M, an attacker can find all query users' queries from past to future.

3. Earlier works lacked code integrity and code confidentiality. In some cases, a client might want to keep his code confidential. E.g., the Client has a new ML model which he wants to keep secret from his competitors. Code integrity is needed as we need to ensure that some attacker has not modified the code. Homomorphic Encryption schemes alone cannot provide code confidentiality and integrity.

4. Earlier works do not consider collusion attack, which is entirely possible in real-life scenarios. E.g., Data Privacy may be compromised when the cloud server colludes with the query user (or when the cloud server acts as the query user). Similarly, Query Privacy is compromised when Data Owner colludes with Cloud Server.

5. Earlier works like Zhu *et al.* (2016) and Singh *et al.* (2018) scheme lacked Access Pattern Hiding.

1.1.2 Performance Motivation

1. Earlier works either lacked Query Controllability or used Paillier Encryption to provide Query Controllability. But Paillier Encryption takes too much time. We experimented on our laptop and found that Data Owner has to spend around 60 seconds to encrypt the query point of dimension 1000. This means that Data Owner has to spend 60 seconds per query user per query. Paillier Encryption won't scale well if there are millions of queries per second. Data Owner has already transferred his data and computation to the cloud. So having such a burden would demotivate Data Owners to use cloud computing.

2. Previous works like Singh *et al.* (2018) and Parampalli *et al.* (2019)'s work involve lots of matrices multiplication which is an overhead.

1.1.3 Availability Motivation

1. Previous works like Zhu *et al.* (2016) and Singh *et al.* (2018) work require Data Owner to be online all the time for query controllability. But this is not practical. If the Data Owner's system goes down, the whole scheme comes to a standstill.

1.2 Contribution

In our proposed design, we have used kNN computation as an example. Our design is a general one. Ours can be used to solve other real-life secure cloud computing problems. Our scheme achieves the following security requirements:

1. **Data Privacy:** Data Owner wants to keep his database private from the cloud server and query users.

2. **Query Privacy:** Query User wants to keep his query confidential from the data owner and cloud server. He also wants to keep the KNN response hidden from the cloud server.

3. **Key Confidentiality:** The key of the Data Owner (used for database encryption) should not be revealed to the cloud server or query user.

4. **Query Controllability:** Query Users should not be able to launch a feasible KNN query without the approval of the Data Owner.

5. **Data Code Integrity:** Data Owner and Query User should be able to verify (or detect) that the correct data and code (provided by Data Owner) are loaded on the cloud server.

6. **Per Query Key:** Query Users should be able to use different keys for different queries so that even if one key is lost, other queries are safe.

7. **Access Pattern Hiding:** Cloud Server should not be able to use KNN results to figure out corresponding encrypted database tuples.

8. **Data Privacy and Query Privacy against collusion attack:** Query User and Cloud Server should not be able to decrypt databases even if they collude. Similarly, Data Owner and Cloud Server should not be able to link Query User's identity to his query.

In addition, the following practical requirements are also guaranteed:

1. Data Owner need not be online.

2. Paillier Encryption and multiple matrix multiplications are eliminated to improve performance.

Our design uses two secure enclaves, one for running KNN code and the other (containing interfacing code) for decoupling the query user's identity from their query. We haven't provided code confidentiality in our design, but it can be added to the enclave without compromising query privacy. We have not added it as the KNN code in our case is simple and doesn't require code confidentiality. To add code confidentiality, we need to build a loader for the secure enclave (containing KNN code in our case), which can accept encrypted code and decrypt it inside the enclave and then load it. The concept is similar to the use of Shield module decrypting and loading the encrypted code in Baumann *et al.* (2015).

To the best of our knowledge, our scheme is the first to utilize secure enclaves to support KNN queries on encrypted cloud data and provide all the security requirements mentioned above. Our contribution can be summarized as the following:

1. We identified vulnerabilities in previous research on the "Privacy Protecting" cloud resident application (KNN computation-based application, which uses Homomorphic Encryption schemes).

2. We identified what features could not be provided by homomorphic encryption, like code integrity, code confidentiality, etc.

3. We developed a new scheme using secure enclaves to circumvent the above vulnerabilities while improving the performance.

4. Our scheme can easily be extended to other works like encrypted query search Cao *et al.* (2014) Li *et al.* (2020).

The rest of the report is organized as follows. In Chapter 2, we discuss the background needed. Chapter 3 discusses the prior works using Homomorphic Encryption. Chapter 4 discusses the flaws we identified in earlier works (discussed in chapter 3). Chapter 5 discusses our new schemes. Chapter 6 discusses the results and experiments performed. Chapter 7 provides a comparison table of our work with previous works. Chapter 8 focuses on the challenges we faced. Finally, we conclude with Chapter 9.

Chapter 2

Background

2.1 Homomorphic Encryption

Homomorphic Encryption allows users to perform computations on encrypted data without first decrypting it. A scheme is homomorphic w.r.t. * if for all inputs m_i and m_j the following is true:

$$E\left(m_i * m_j\right) = E\left(m_i\right) * E\left(m_j\right)$$

Homomorphic Encryption is of 3 types:

1. **Fully Homomorphic:** Allows an unlimited number of operations an unlimited number of times.

2. **Somewhat Homomorphic:** Allows some types of operations a limited number of times.

3. **Partially Homomorphic:** Allows only one operation an unlimited number of times. E.g., Paillier Encryption

We only need to know Partially Homomorphic Encryption to understand prior works. To be more precise, we need to know Paillier Encryption.

2.2 Paillier Encryption

Paillier Encryption is additive homomorphic. It has the following two properties:

1. $E_{pk}\left(m_1 + m_2\right) = E_{pk}\left(m_1\right) \times E_{pk}\left(m_2\right)$

2. $E_{pk}\left(c \times m_1\right) = \left(E_{pk}\left(m_1\right)\right)^c$

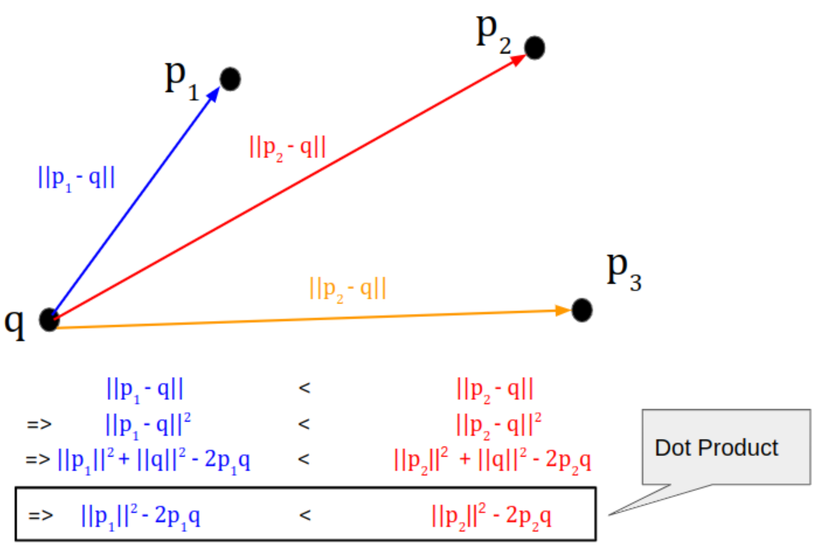

p1
p2
p3
q
	p1 - q	
	p2 - q	
	p3 - q	
|p1 - q|| < ||p2 - q|
=> ||p1 - q||² < ||p2 - q||²
=> ||p1||² + ||q||² - 2p1q < ||p2||² + ||q||² - 2p2q
=> ||p1||² - 2p1q < ||p2||² - 2p2q
Dot Product

Figure 2.1: k-Nearest Neighbor using dot product

Here, pk is the Paillier Public Key, m_1 and m_2 are messages and c is a positive integer. This two properties are enough to understand prior works. For more details, refer (Paillier, 1999).

2.3 k-Nearest Neighbor

k-Nearest Neighbor computation helps us find k nearest tuples of a given query point according to a certain distance metric, such as Euclidean distance (Zhu *et al.*, 2016). If we look at fig. 2.1, we observe that we can compute the Euclidean distance of p_i and q, i.e., $D(p_i, q) = \sqrt{\sum_{j=1}^{d} \left(p_{ij} - q_j\right)^2}$, using dot product. The same idea is used in previous works and our design as well.

2.4 Secure Enclaves

In our proposed design, we used Intel SGX for secure enclaves as it comes free (by default) in Intel Core Processors (6th Gen to 10th Gen) and Xeon Processors.

2.4.1 Intel SGX

Intel Software Guard Extensions (SGX) is a hardware-based data protection technology, developed by Intel Corporation (insujang, 2017). It provides confidentiality and integrity guarantees to security-sensitive applications. It assumes that privileged softwares (kernel, hypervisor, etc.) are potentially malicious (Costan and Devadas, 2016). Only CPU package (containing Cache) is trusted. SGX protects secrets by creating an

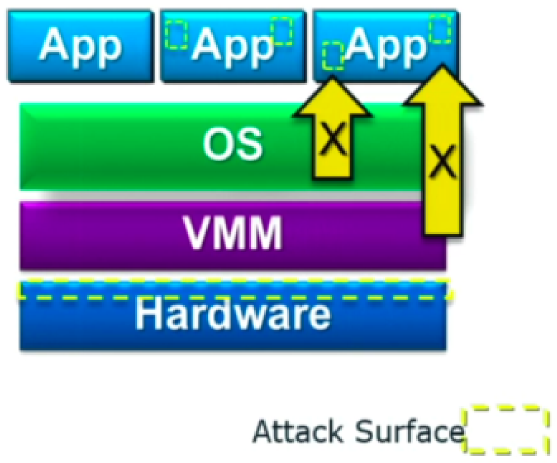


Figure 2.2: Attack Surface with Intel SGX (McKeen, 2015)

isolated execution environment for code and data called enclaves. It reduces the attack surface (Fig. 2.2). SGX offers two crucial properties:

1. **Isolation:** Data and code inside enclaves are isolated from untrusted software and other enclaves.

2. **Attestation:** It allows local/remote parties to authenticate the software running inside an enclave.

2.4.2 SGX Application

SGX applications are divided into two parts (Fig. 2.3):

1. **Trusted Part:** Part of the application dealing with sensitive data is called Trusted Part and is executed inside an enclave.

2. **Untrusted Part:** Remaining part of the application is called the Untrusted Part. This part creates an enclave and invokes them.

2.4.3 How SGX maintains confidentiality

Contents of enclaves are encrypted before sending them to the RAM using a new dedicated chip called Memory Encryption Engine (MEE). External reads on the memory bus can only observe encrypted data. Pages are decrypted when data comes back inside the physical processor core (Fig. 2.4).

Data Sealing: When enclave's data is sent to untrusted memory, it is encrypted using an encryption key derived from the CPU. The encrypted data block, also called the sealed

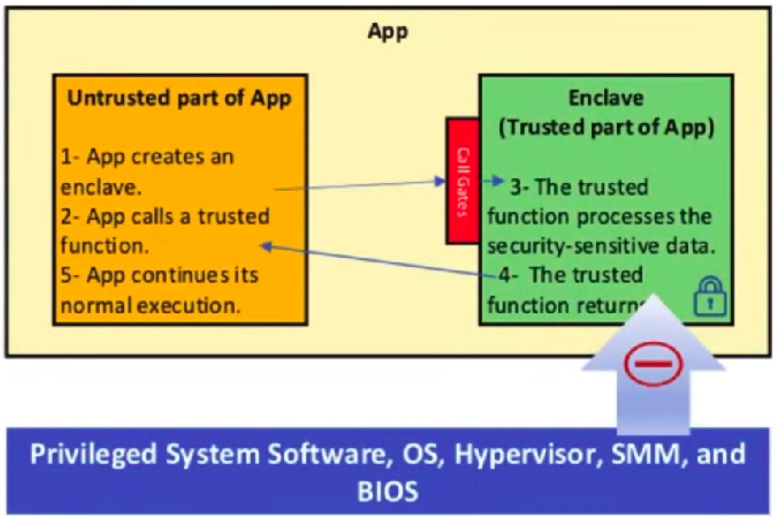

App
Untrusted part of App
1- App creates an enclave.
2- App calls a trusted function.
5- App continues its normal execution.
Call Gates
Enclave (Trusted part of App)
3- The trusted function processes the security-sensitive data.
4- The trusted function returns
Privileged System Software, OS, Hypervisor, SMM, and BIOS

Figure 2.3: SGX Application Execution Flow (Rebeiro, 2018)

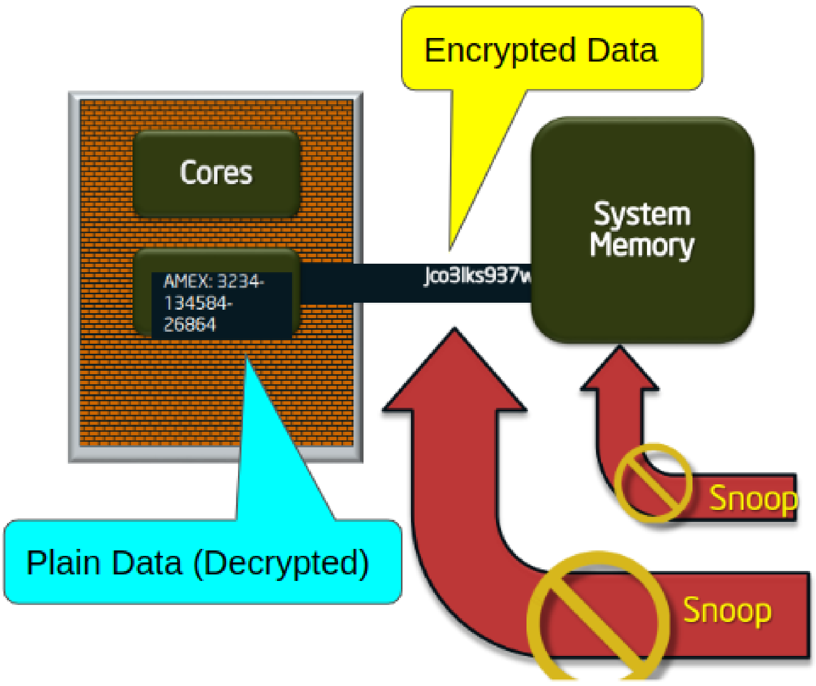


Figure 2.4: How SGX maintains confidentiality (McKeen, 2015)

data, can only be decrypted on the same system where it was created. This assures confidentiality, integrity, and authenticity of data.

2.4.4 Software Attestation

It allows local/remote parties to authenticate the software running inside an enclave. There are two types (Intel, 2015):

1. **Local Attestation:** It allows 2 enclaves on the same platform to authenticate each other.

2. **Remote Attestation:** It allows a remote party to authenticate the software running inside an enclave.

Enclave Measurement

Intel SGX generates a cryptographic log of all build activities when an enclave is created (ECREATE and EADD instruction) and initialized (EINT and EEXTEND instruction). The cryptographic log includes the hash of the following (McKeen, 2015):

1. Stack, Heap, Code, and Data

2. Each page's location within the enclave

3. Security flags being used

A 256-bit hash digest of the log called "Enclave Identity" is stored in the MRENCLAVE register. MRENCLAVE value represents the enclave's software TCB (Trusted Computing Base). A software TCB verifier has to obtain

1. Enclave's software TCB securely

2. Expected enclave's software TCB securely

3. And finally, compare the two values.

If the two values match, then the attestation is successful.

Enclave's REPORT

Enclave's Report contains the following (SGX101, 2019):

1. Code's and data's measurement inside the enclave

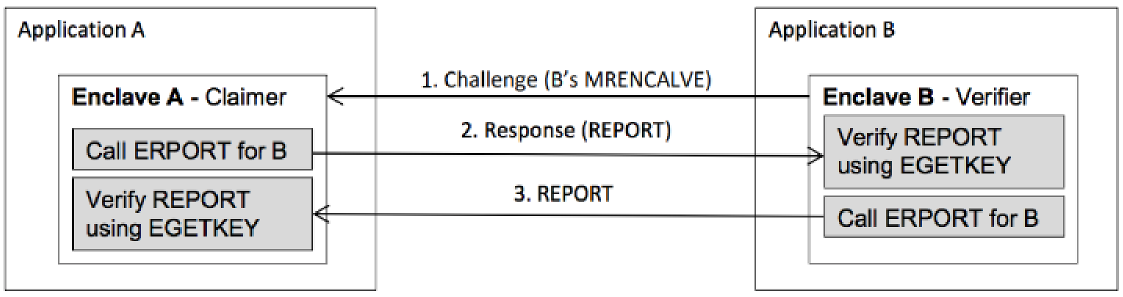


Figure 2.5: Local Attestation (SGX101, 2019)

2. User data field (In our scheme, we will add the public key of the enclave in this user data field. This public key will be used for secure data transfer)

3. Some security-related state information

4. A signature block over the above data, which can be verified by the same platform that produced the Report (For Local Attestation), etc.

Intel SGX Report is similar to a certificate issued by the certificate authority. Just as the certificate proves the identity and authenticity of a website (or other entities), SGX Report proves the integrity of the code and data inside the SGX enclave. It proves that the data and code inside the enclave haven't been tampered with by Cloud Server or attackers. We can also share encryption/decryption keys generated inside enclaves using the user data field of this report.

Local Attestation

It allows one enclave to attest its TCB to another enclave on the same platform (SGX101, 2019). Suppose there are two enclaves on the same platform, called Enclave A and Enclave B. The communication path between them needn't be trusted. Suppose B asks A to prove it's running on the same platform as B (Fig. 2.5). Steps:

1. B sends its MRENCLAVE value to A via the untrusted channel.

2. A uses EREPORT instruction to produce a report for B using B's MRENCLAVE. Then A sends this report back to B. A can also include Diffie-Hellman Key Exchange data in the REPORT as user data for trusted channel creation in the future.

3. After receiving the report, B calls EGETKEY instruction to get REPORT KEY to verify the REPORT (REPORT KEY is specific to the platform). If the report is verified, then B is sure that A is on the same platform.

4. B uses the MRENCLAVE received from A’s REPORT to create another REPORT for A and sends the REPORT to A.

5. A can do the same as step 3 to verify B is on the same platform as A.

By utilizing the user data field of the REPORT, A and B can create a secure channel using Diffie-Hellman Key Exchange. The shared symmetric key can encrypt information exchange.

Remote Attestation

It allows a remote party to authenticate the software running inside an enclave. Intel SGX extends Local Attestation by allowing a Quoting Enclave (QE) to use Intel EPID (a group signature scheme) to create a QUOTE out of a REPORT. Remote Attestation is a complex process happening in the background. We don’t need to know the full details. So we will discuss it at a very high level. For more information, refer (Costan and Devadas, 2016). Suppose there is an Enclave A that needs to be verified by a remote party.
Steps: (Intel, 2015)

1. Quoting Enclave verifies Enclave A through Local Attestation Steps.

2. After verifying the REPORT, Quoting Enclave signs the REPORT with the EPID private key and converts it into a QUOTE. Quoting Enclave sends this Quote to the remote party.

3. The remote party forwards the QUOTE to the Intel Attestation Centre, which responds whether the QUOTE is valid or not.

Chapter 3

Prior Works

The main goal of the previous related works (on secure k-NN computation on the cloud) has been to find k database tuples closest (in terms of Euclidean distance) to the query tuple (given an encrypted database of d-dimensions and an encrypted query of d-dimensions).

3.1 Zhu *et al.* (2016) Scheme

Zhu *et al.* (2016) claimed that their scheme achieves Data Privacy, Query Privacy, Key Confidentiality, and Query Controllability. They used matrix multiplication to alter (hide) the data points and query points Singh *et al.* (2018) and Paillier encryption for Query Controllability.

3.1.1 Data Owner sends Encrypted DataBase to Cloud Server

The data point p is represented for encryption as follows:

$$\hat{p} = \pi\left(S_1 - 2p_1, S_2 - 2p_2, \cdots, S_d - 2p_d, S_{d+1} + ||p||^2, \tau, v\right)$$

1. p_i represents the i^{th} dimension of data point p
2. Vector S = $\{S_1, S_2, \cdots, S_{d+1}\}$ and τ (c-dimensional vector) are constant vectors, generated by Data Owner during the KeyGen function call.
3. v (ϵ dimensional vector) is randomly chosen vector by Data Owner during the DBEnc function call for the data points.

π is an instance of random permutation applied to all data and query points by Data Owner. Data Owner's secret key is {S, τ, M, π}.
Data points are encrypted by Data Owner as $p' = \hat{p}M^{-1}$ where M is a (d+1+c+ϵ) x (d+1+c+ϵ) matrix. Data Owner sends p' to Cloud Server (Fig. 3.1).

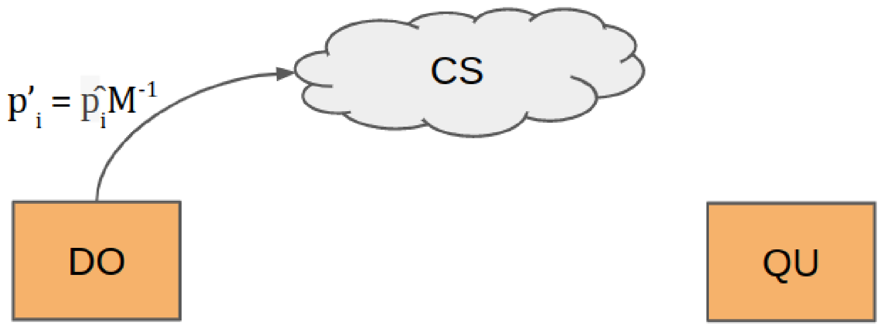


Figure 3.1: Data Owner sends Encrypted DataBase to Cloud Server

3.1.2 Query User and Data Owner cooperatively compute Encrypted Query

The query point q is represented for encryption as follows:

$$\hat{q} = \pi\left(q_1, q_2, \cdots, q_d, 1, R^{(q)}, 0_\epsilon\right)$$

1. $R^{(q)}$ is c sized random vector.

2. 0_ϵ is ϵ sized zero vector.

Query point are encrypted by both Data Owner and Query User engaging among themselves to cooperatively computing the encryption q' using the additive homomorphic Paillier encryption scheme Singh *et al.* (2018).

Steps for computing encrypted query by Data Owner and Query User Singh *et al.* (2018) (Fig. 3.2):

1. Query User generates a Paillier Encryption key pair {pk, sk} and encrypts each dimension of query point using the public key pk. It then sends pk and encrypted dimensions $\{E_{pk}(q_1), E_{pk}(q_2), \cdots, E_{pk}(q_d)\}$ to Data Owner.

2. Data Owner selects a random number β_q and random c sized vector $R^{(q)}$. It then constructs a partial Paillier encrypted vector $\bar{q}$ such that:

$$\bar{q} = \pi\left(E_{pk}\left(q_1\right)^{\beta_q}, E_{pk}\left(q_2\right)^{\beta_q}, \cdots, E_{pk}\left(q_d\right)^{\beta_q}, \beta_q, \beta_q \cdot \mathbf{R}^{(q)}, 0_\epsilon\right)$$

Using the vector $\bar{q}$ and matrix M, a Paillier encrypted $(1 + d + c + \epsilon)$ sized vector $A^{(q)}$ is computed such that:

$$A_i^{(q)} = \prod_{l=1}^{n} E_{pk}\left(\phi_l\right), \text{ where}$$

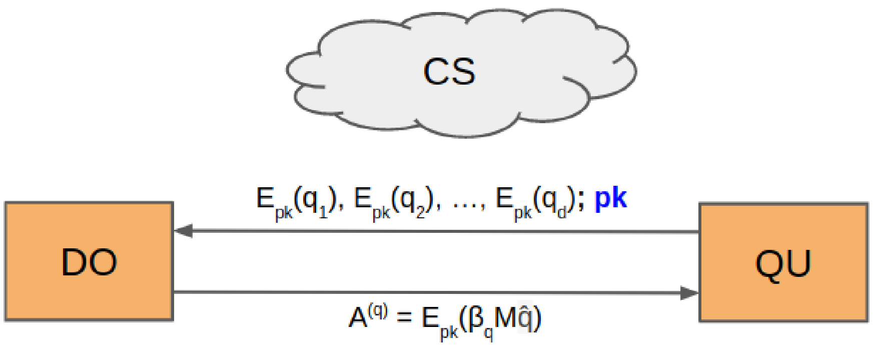

CS
$E_{pk}(q_1), E_{pk}(q_2), \ldots, E_{pk}(q_d)$; pk
DO
QU
$A^{(q)} = E_{pk}(\beta_q M\hat{q})$

Figure 3.2: Query User and Data Owner cooperatively compute Encrypted Query

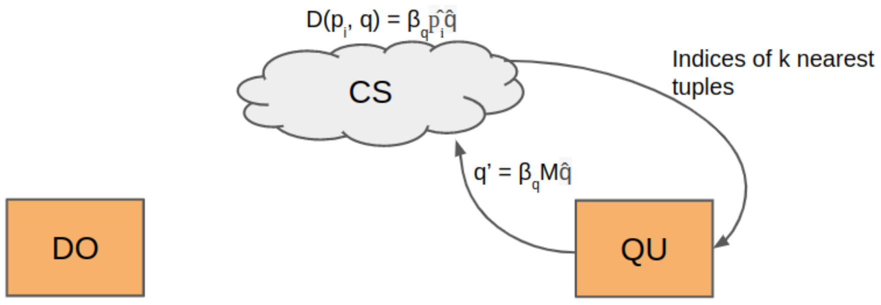

$D(p_i, q) = \beta_q \hat{p}_i \hat{q}$
CS
Indices of k nearest tuples
$q' = \beta_q M\hat{q}$
DO
QU

Figure 3.3: Query User sends Encrypted Query and get k-NN result from Cloud Server

$$\phi_l = \begin{cases} E_{pk}(\bar{q}_l)^{M[i,l]} & \text{if } \bar{q}_l \text{ is paillier encrypted} \\ E_{pk}(\bar{q}_l \times \boldsymbol{M}[i,l]) & \text{otherwise} \end{cases}$$

Data Owner sends vector $A^{(q)}$ back to Query User.

3. Query User decrypts each component of $A^{(q)}$ using the secret key sk to get q'.

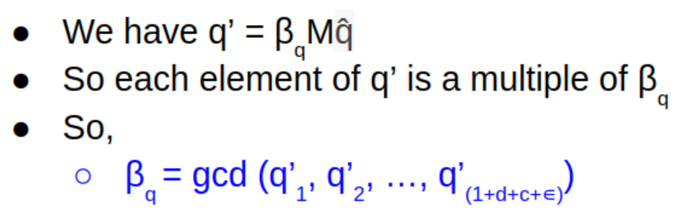
$$q' = \beta_q \cdot M \cdot \hat{q}^T$$

3.1.3 Query User sends Encrypted Query and get k-NN result from Cloud Server

Now, Query User sends this q' to Cloud Server. Cloud Server performs the KNN computation on p' and q', and returns the indices of K- nearest database tuples (Fig. 3.3).

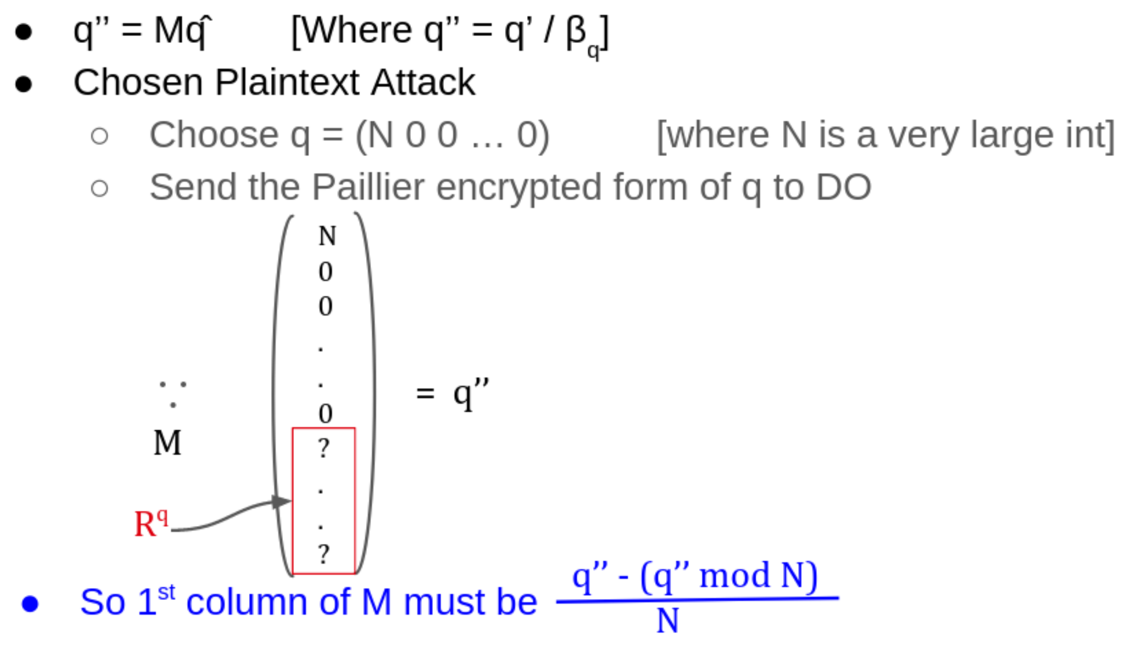
3.2 Attacks on Zhu *et al.* (2016) Scheme

This section is taken from IIT Bombay's "CS741: Advanced Network Security and Cryptography" course taught by Prof. Bernard Menezes.



- We have $q' = \beta_q M\hat{q}$
- So each element of q' is a multiple of β_q
- So,
  - $\beta_q = \gcd(q'_1, q'_2, \ldots, q'_{(1+d+c+\epsilon)})$

Figure 3.4: Attack 1: Obtaining the secret β_q

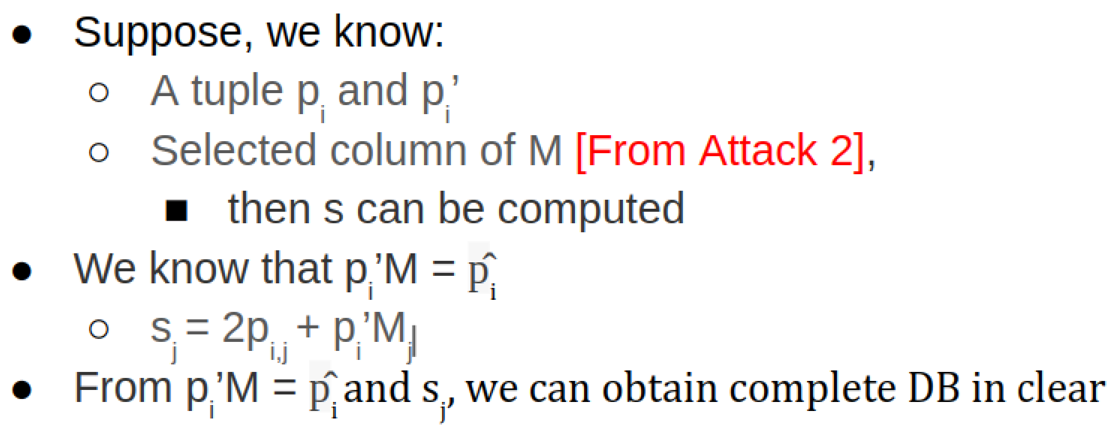


- $q'' = M\hat{q}$ [Where $q'' = q' / \beta_q$]
- Chosen Plaintext Attack
  - Choose $q = (N\ 0\ 0\ \ldots\ 0)$ [where N is a very large int]
  - Send the Paillier encrypted form of q to DO

$$\because M \begin{pmatrix} N \\ 0 \\ 0 \\ \cdot \\ \cdot \\ 0 \\ ? \\ \cdot \\ \cdot \\ ? \end{pmatrix} = q''$$

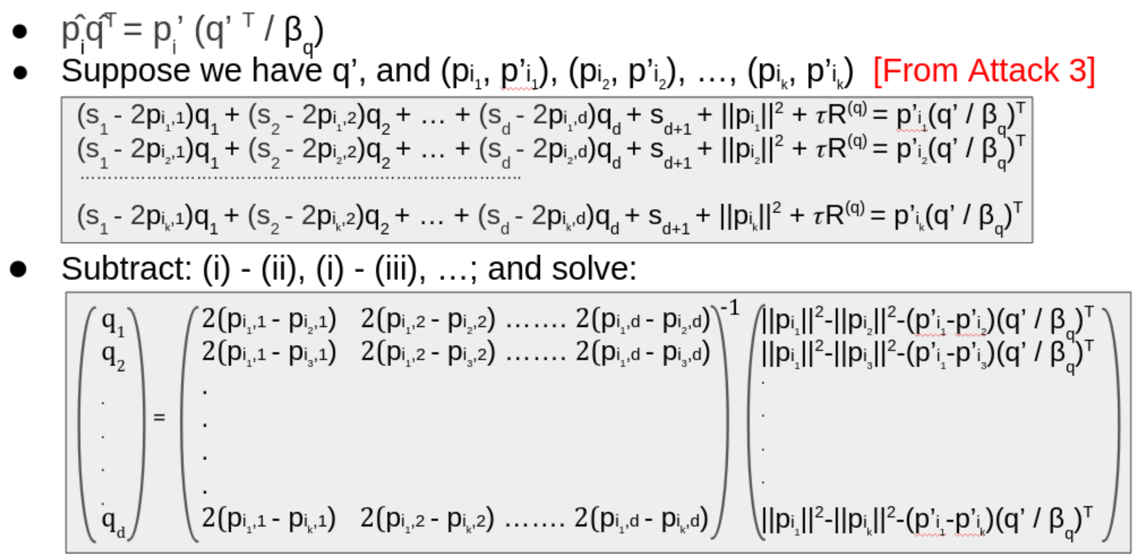
- So 1st column of M must be $\dfrac{q'' - (q'' \bmod N)}{N}$

Figure 3.5: Attack 2: Obtaining the secret M



- Suppose, we know:
  - A tuple p_i and p_i'
  - Selected column of M [From Attack 2],
    - then s can be computed
- We know that $p_i'M = \hat{p_i}$
  - $s_j = 2p_{i,j} + p_i'M_j$
- From $p_i'M = \hat{p_i}$ and s_j, we can obtain complete DB in clear

Figure 3.6: Attack 3: Extracting the complete Database in clear



- $\hat{p}_i \hat{q}^T = p_i' \, (q'^T / \beta_q)$
- Suppose we have q', and $(p_{i_1}, p'_{i_1}), (p_{i_2}, p'_{i_2}), \ldots, (p_{i_k}, p'_{i_k})$ [From Attack 3]

$$(s_1 - 2p_{i_1,1})q_1 + (s_2 - 2p_{i_1,2})q_2 + \ldots + (s_d - 2p_{i_1,d})q_d + s_{d+1} + \|p_{i_1}\|^2 + \tau R^{(q)} = p'_{i_1}(q' / \beta_q)^T$$
$$(s_1 - 2p_{i_2,1})q_1 + (s_2 - 2p_{i_2,2})q_2 + \ldots + (s_d - 2p_{i_2,d})q_d + s_{d+1} + \|p_{i_2}\|^2 + \tau R^{(q)} = p'_{i_2}(q' / \beta_q)^T$$
$$\cdots\cdots\cdots\cdots$$
$$(s_1 - 2p_{i_k,1})q_1 + (s_2 - 2p_{i_k,2})q_2 + \ldots + (s_d - 2p_{i_k,d})q_d + s_{d+1} + \|p_{i_k}\|^2 + \tau R^{(q)} = p'_{i_k}(q' / \beta_q)^T$$

- Subtract: (i) - (ii), (i) - (iii), …; and solve:

$$\begin{pmatrix} q_1 \\ q_2 \\ . \\ . \\ . \\ . \\ q_d \end{pmatrix} = \begin{pmatrix} 2(p_{i_1,1} - p_{i_2,1}) & 2(p_{i_1,2} - p_{i_2,2}) & \ldots\ldots & 2(p_{i_1,d} - p_{i_2,d}) \\ 2(p_{i_1,1} - p_{i_3,1}) & 2(p_{i_1,2} - p_{i_3,2}) & \ldots\ldots & 2(p_{i_1,d} - p_{i_3,d}) \\ . & & & \\ . & & & \\ . & & & \\ . & & & \\ 2(p_{i_1,1} - p_{i_k,1}) & 2(p_{i_1,2} - p_{i_k,2}) & \ldots\ldots & 2(p_{i_1,d} - p_{i_k,d}) \end{pmatrix}^{-1} \begin{pmatrix} \|p_{i_1}\|^2 - \|p_{i_2}\|^2 - (p'_{i_1} - p'_{i_2})(q' / \beta_q)^T \\ \|p_{i_1}\|^2 - \|p_{i_3}\|^2 - (p'_{i_1} - p'_{i_3})(q' / \beta_q)^T \\ . \\ . \\ . \\ . \\ \|p_{i_1}\|^2 - \|p_{i_k}\|^2 - (p'_{i_1} - p'_{i_k})(q' / \beta_q)^T \end{pmatrix}$$

Figure 3.7: Attack 4: Obtaining query in clear

3.3 Singh *et al.* (2018) Scheme

Singh *et al.* (2018) claimed that their scheme achieves Data Privacy, Query Privacy, Key Confidentiality, Query Controllability, and Query Check Verification. They used matrix multiplication to transform the data points and query points Singh *et al.* (2018) and Paillier encryption for Query Controllability and Query Check Verification.

3.3.1 Data Owner sends Encrypted DataBase to Cloud Server

Data Owner and Cloud Server jointly agree upon key pair $< K_{SBC}, W >$ where:

1. K_{SBC} is key of secure block cipher like AES, and
2. W is randomly generated $\eta \times \eta$ invertible matrix where $\eta = n + l$.

The data point p is represented for encryption as follows:

$$\hat{p} = \pi \left(S_1 - 2p_1, S_2 - 2p_2, \cdots, S_d - 2p_d, S_{d+1} + \|p\|^2, \tau, v \right)$$

1. p_i represents the i^{th} dimension of data point p
2. Vector S = $\{S_1, S_2, \cdots, S_{d+1}\}$ and τ (c-dimensional vector) are constant vectors, generated by Data Owner during the KeyGen function call.
3. v (ϵ dimensional vector) is randomly chosen vector by Data Owner during the DBEnc function call for the data points.

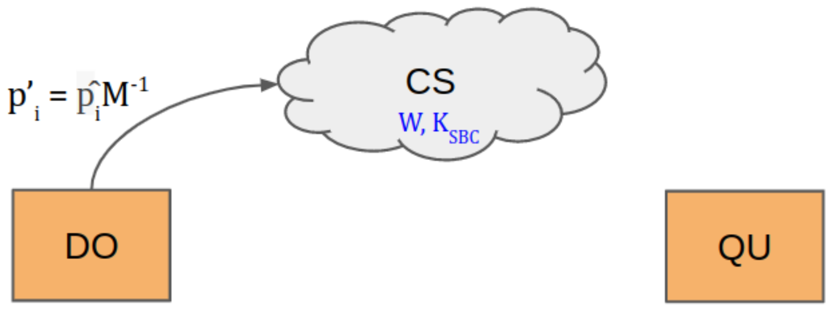


Figure 3.8: Data Owner sends Encrypted DataBase to Cloud Server

π is an instance of random permutation applied to all data and query points by Data Owner Singh *et al.* (2018). Data Owner's secret key is {S, τ, M, π}.
Data points are encrypted by Data Owner as $p' = \hat{p}M^{-1}$ where M is an n x n matrix, where n = (d+1+c+ϵ). Data Owner sends p' to Cloud Server (Fig. 3.8).

3.3.2 Query User and Data Owner cooperatively compute Encrypted Query

The query point q is represented for encryption as follows:

$$\dot{q} = \pi\left(q_1, q_2, \cdots, q_d, 1, R^{(q)}, 0_\epsilon\right)$$

1. $R^{(q)}$ is c sized random vector.

2. 0_ϵ is ϵ sized zero vector.

Query point are encrypted by both Data Owner and Query User engaging among themselves to cooperatively computing the encryption q' using the additive homomorphic Paillier encryption scheme Singh *et al.* (2018). Steps for computing encrypted query by Data Owner and Query User (Fig. 3.9:

1. Query User generates a Paillier Encryption key pair {pk, sk} and encrypts each dimension of query point using the public key pk. It then sends pk and encrypted dimensions $\{E_{pk}(q_1), E_{pk}(q_2), \cdots, E_{pk}(q_d)\}$ to Data Owner.

2. Data Owner selects a random number β_q and random c sized vector $R^{(q)}$. It then constructs a partial Paillier encrypted vector $\bar{q}$ such that:

$$\bar{q} = \pi\left(E_{pk}(q_1)^{\beta_q}, E_{pk}(q_2)^{\beta_q}, \cdots, E_{pk}(q_d)^{\beta_q}, \beta_q, \mathbf{R}^{(q)}, 0_\epsilon\right)$$

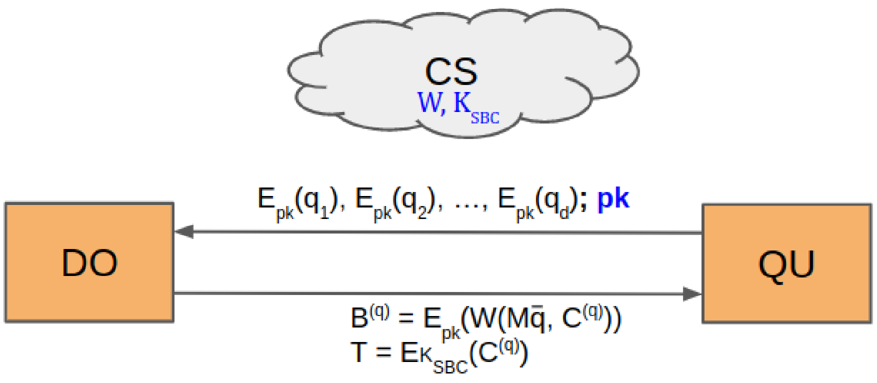

CS
W, K_SBC
E_pk(q_1), E_pk(q_2), ..., E_pk(q_d); pk
DO
QU
B^(q) = E_pk(W(Mq̄, C^(q)))
T = EK_SBC(C^(q))

Figure 3.9: Query User and Data Owner cooperatively compute Encrypted Query

Using the vector $\bar{q}$ and matrix M, a Paillier encrypted n sized vector $A^{(q)}$ is computed such that:

$$A_i^{(q)} = \prod_{l=1}^{n} E_{pk}\left(\phi_l\right), \text{ where}$$

$$\phi_l = \begin{cases} E_{pk}\left(\bar{q}_l\right)^{M[i,l]} & \text{if } \bar{q}_l \text{ is paillier encrypted} \\ E_{pk}\left(\bar{q}_l \times \boldsymbol{M}[i,l]\right) & \text{otherwise} \end{cases}$$

3. Data Owner choses l size query check vector $C^{(q)}$ and encrypts it with Secure Block Cipher using the key K_{SBC}, i.e., T = $E_{K_{SBC}}(C^{(q)})$. Data Owner then appends the check vector $C^{(q)}$ to the end of vector $A^{(q)}$, i.e., $\hat{q} = \{A^{(q)}, C^{(q)}\}$. Using a vector $\hat{q}$ and matrix W, a Paillier encrypted η sized $B^{(q)}$ is computed such that:

$$B_i^{(q)} = \prod_{l=1}^{\eta} E_{pk}\left(\phi_l\right), \text{ where}$$

$$\phi_l = \begin{cases} E_{pk}\left(\hat{q}_l\right)^{W[i,l]} & \text{if } \hat{q}_l \text{ is paillier encrypted} \\ E_{pk}\left(\hat{q}_l \times \boldsymbol{W}[i,l]\right) & \text{otherwise} \end{cases}$$

Data Owner sends vector $< B^{(q)}, T >$ back to Query User

4. Query User decrypts each component of $B^{(q)}$ using the secret key sk to get $\tilde{q}$.

3.3.3 Query User sends Encrypted Query and get k-NN result from Cloud Server

Query User sends <$\tilde{q}$, T> to cloud server. Cloud Server applies inverse matrix multiplication operation on $\tilde{q}$ w.r.t. matrix W to compute the vector <$q', C^{(q)}$> and then checks the verifiability condition, i.e., if ($C^{(q)} == D_{K_{SBC}}(T)$). If the condition is true Cloud Server proceeds with KNN computation using dot product on p' and q' (and return the k nearest tuple indices to QU), otherwise, report the query as fake to Query User (Fig. 3.10).

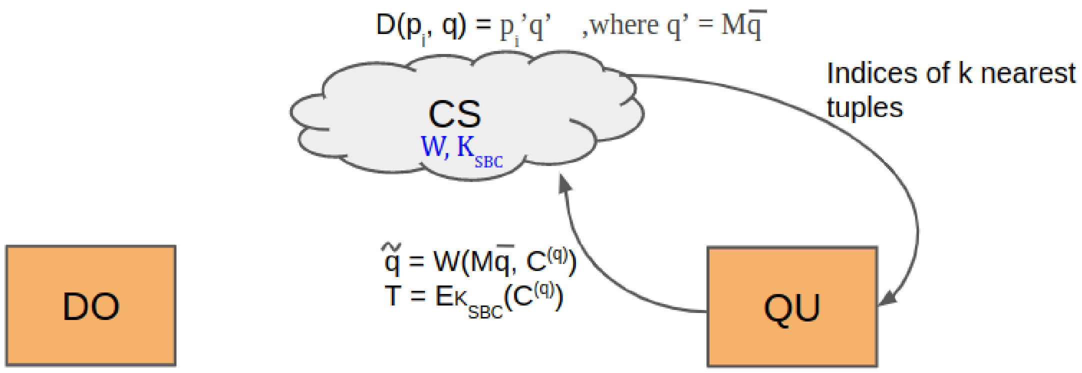

D(p_i, q) = p_i'q' ,where q' = Mq̄
CS
W, K_SBC
Indices of k nearest tuples
q̃ = W(Mq̄, C^(q))
T = E_{K_SBC}(C^(q))
DO
QU

Figure 3.10: Query User sends Encrypted Query and get k-NN result from Cloud Server

3.4 Parampalli *et al.* (2019) Scheme

Parampalli *et al.* (2019) claimed that their scheme achieves Data Privacy, Query Privacy, Key Confidentiality, and Hiding of Data Access Pattern. They also used matrix multiplication to transform the data points and query points.

3.4.1 Data Owner sends Encrypted Database to Cloud C1

Data Owner's secret key (generated by Data Owner during the KeyGen function call) are:

1. Data Encryption Key $k_p = \{\alpha, r, e, M\}$.
   (a) r = $\{r_1, r_2, \cdots, r_{d+1}\}$, $\alpha = \{\alpha_1, \alpha_2, \cdots, \alpha_{d+1}\}$ and e = $\{e_1, e_2, \cdots, e_{d+1}\}$ are constant vectors. d is the dimension of plain database points.
   (b) M is a $(2d + 2) \times (2d + 2)$ invertible matrix.
2. Classification Label Encryption Key $k_c = M_c$. M_c is (2×2) invertible matrix.

The plain database tuples consists of d-dimensional plain data points p_i and plain classification label c_i.

Encryption of plain database points p_i

The data point p_i $(1 \leq i \leq n)$ is represented for encryption as follows:

$$\dot{p}_i = \left(r_1 - 2\alpha_1 p_{i1}, s_{i1}, r_2 - 2\alpha_2 p_{i2}, s_{i2}, \ldots, r_d - 2\alpha_d p_{id}, s_{id}, r_{d+1} + \alpha_{d+1} \sum_{j=1}^{d} p_{ij}^2, s_{i(d+1)}\right)$$

1. p_{ij} represents the j^{th} dimension of data point p_i.

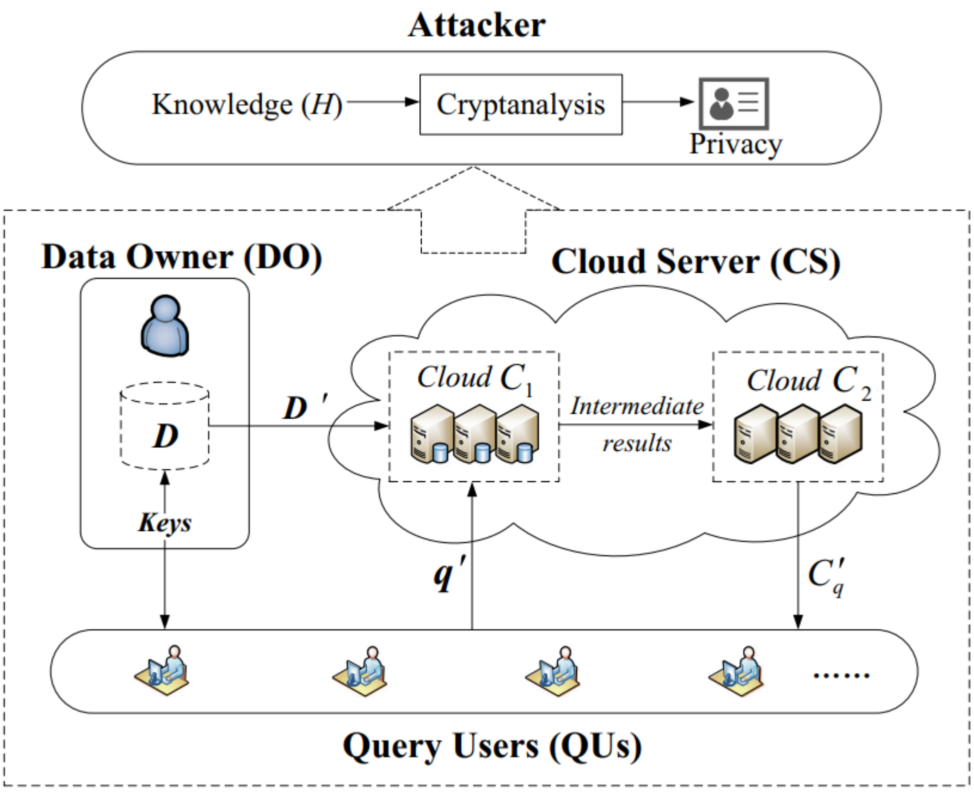


Figure 3.11: System Architecture in Parampalli's Scheme (Parampalli *et al.*, 2019)

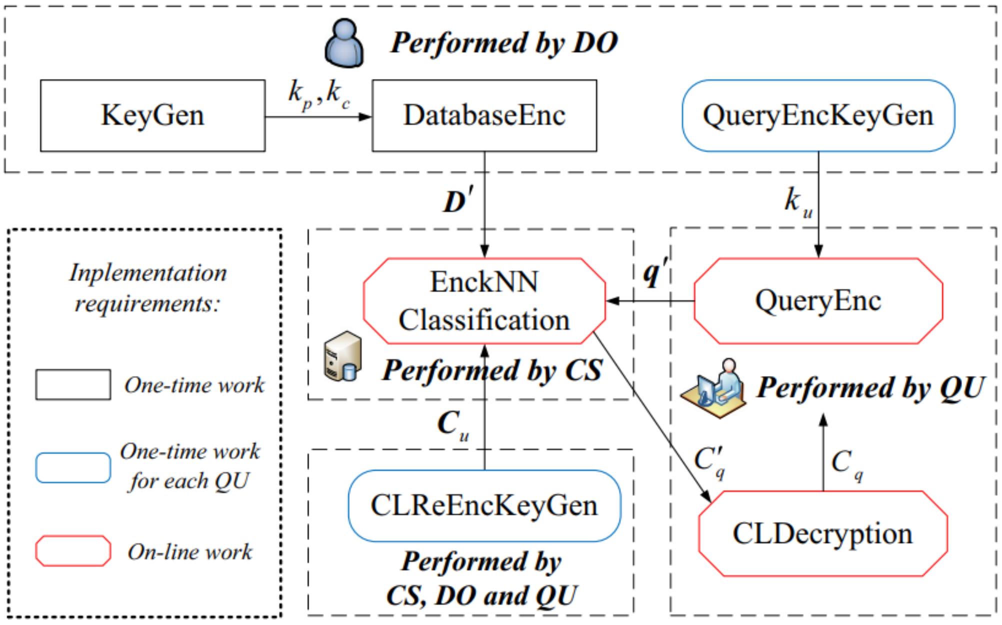


Figure 3.12: Different processes in Parampalli's Scheme (Parampalli *et al.*, 2019)

2. s_i ((d + 1) dimensional vector) is randomly chosen vector by Data Owner during the DBEnc function call for each data point. They are orthogonal to another vector e generated during KeyGen function call.

Data points are encrypted by Data Owner as $p'_i = \dot{p}_i \times \sigma_i M^{-1}$. ($\sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_n\}$ is a constant vector chosen by Data Owner during the DBEnc function call).

Encryption of plain classification labels c_i

Each c_i is split into 2 random numbers v_{i1} and v_{i2} such that $c_i = v_{i1} + v_{i2}$ and get vectors $v_i = [v_{i1}, v_{i2}]$ for $1 \le i \le n$. Classification Labels are encrypted by Data Owner as $v'_i = v_i \times M_c$.
Then, Data Owner sends the encrypted database $D' = \{t'_1, t'_2, \ldots, t'_n\}$ to cloud C_1 ($t'_i = (p'_i, v'_i)$). Data Owner also generates an n-length random permutation function π and computes $\hat{\sigma} = \pi(\sigma)$. After that, Data Owner sends π to C_1 and sends the permuted set $\hat{\sigma}$ to cloud C_2. These will be later used in kNN classification later.

3.4.2 DO provides unique query encryption key to each QU

Using DB encryption key k_p, Data Owner computes a unique encryption key k_u for an authorized query user u. DO generates a diagonal matrix L_u given below:

$$L_u = \begin{bmatrix} \beta_1^u & & & & & & \\ & \tau^u e_1 & & & & & \\ & & \beta_2^u & & & & \\ & & & \tau^u e_2 & & & \\ & & & & \ddots & & \\ & & & & & \beta_{d+1}^u & \\ & & & & & & \tau^u e_{d+1} \end{bmatrix}$$

1. τ^u is a constant and X^u is a positive constant generated by DO.

2. $\beta_i^u = X^u/\alpha_i$ for $1 \le i \le (d+1)$.

3. $\alpha, e \epsilon k_p$.

DO compute the unique query encryption key $k_u = L_u M^T$ where M ϵ k_p. Then DO sends this key k_u to QU and goes offline.

3.4.3 Query Encryption by Query User

The query point q is represented for encryption as follows:

$$\tilde{q} = (q_1, 1, q_2, 1, \cdots, q_d, 1, 1, 1)$$

QU encrypts the above query point as $q' = \tilde{q} \times \gamma L_u M^T$ and sends this q' to cloud C_1 for encrypted kNN classification.

3.4.4 DO, Cloud and QU together generates a unique Classification Label Re-Encryption Key for each QU

This step is jointly performed by DO, Cloud C_1 and QU.
Steps:

1. Cloud C1 sends a one-time randomly generated invertible matrix θ_u to QU. θ_u is a 2 x 2 matrix.

2. QU sends $A = Q_u \times \theta_u$ to DO. Q_u is a 2 x 2 invertible matrix generated by QU.

3. DO computes $B = M_c^{-1} \times R_u \times A$ and sends B to cloud C_1. R_u is a random matrix generated by DO as below:

$$\boldsymbol{R}_u = \begin{bmatrix} R^u_{11} & R^u_{12} \\ R^u_{21} & R^u_{22} \end{bmatrix}, R^u_{j1} + R^u_{j2} = 1 \text{ for } j \in [2]$$

4. Cloud C_1 computes re-encryption key $C_u = B \times \theta_u^{-1} = M_c^{-1} R_u Q_u$.

Here, Q_u is the private unique classification labels decryption key for Query User u. Q_u is the unique classification labels re-encryption key (for Query User u) but only known to cloud C_1. After generating C_u, Data Owner can stay offline from the following classification process.

3.4.5 kNN Classication

This step is performed by cloud C_1 and C_2 together. First, Cloud C_1 performs the following steps:

1. Re-encrypt the classification labels v'_i as follows:

$$\dot{v}_i = v'_i \times C_u = v_i R_u Q_u$$

2. Compute inner products between p'_i and q'_i.

3. Permute the two sets (obtained above), one of $\dot{v}_i$ and other of inner products using permutation function π generated during DataBaseEnc Phase by DO. Sends both the sets to cloud C_2.

Then Cloud C_2 performs the following steps:

1. Note that σ_i was multiplied during DataBaseEnc Phase to each data points and it must be removed before computing the kNN result. Using $\hat{\sigma} = \pi(\sigma)$ (obtained from DO in DataBaseEnc Phase), it computes the actual dot product by dividing dist (or $p'_i.q'_i$) by corresponding σ_i.

2. Based on the actual distances obtained in above step, it find out the set of top k re-encrypted kNN classification label C'_q and sends it to QU.

3.4.6 Query User decrypts classification label and get result

Query User u uses its unique classification labels decryption key Q_u to compute the plain kNN classification labels C_q as follows (Parampalli *et al.*, 2019):

1. Compute k vectors $\tilde{v}_i$ by multiplying each element of C'_q with Q_u^{-1} to get $(v_i \times R_u) = [\tilde{v_{i1}}, \tilde{v_{i2}}]$

2. Compute $\tilde{c}_i = \tilde{v_{i1}} + \tilde{v_{i2}}$ for each $\tilde{v}_i$ (obtained above) and get set of plain classification labels C_q of the query point q. For detailed proof, refer Parampalli *et al.* (2019).

Thus, Query User u obtained the correct plain kNN classication labels.

Chapter 4

Vulnerabilities Identified in Previous Works

Previous works considered Cloud Server, Data Owner, and Query User as honest-but-curious, which means each party will follow the steps strictly and return correct computation results. At the same time, they will try to infer as much information about others' data as possible from the data they get from other parties. But this assumption is not realistic. Data Owner may collude with Cloud Server, or Cloud Server acts as Query User, etc. In this chapter, we will discuss the vulnerabilities (we found) in previous works without assumptions (like honest-but-curious).

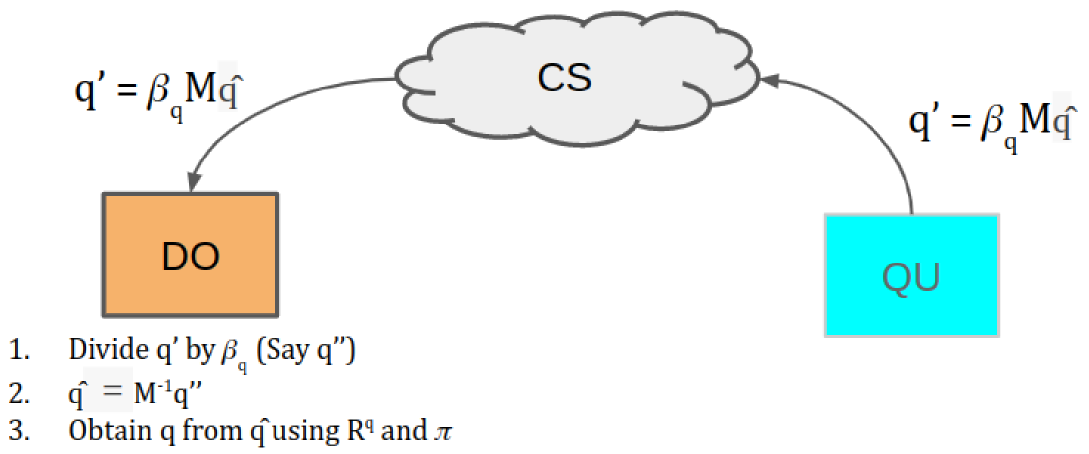
4.1 Security Vulnerability

4.1.1 Lacks Data and Code Integrity

Previous works (like Zhu *et al.* (2016), Singh *et al.* (2018), and Parampalli *et al.* (2019)) lacks data and code integrity. Its entirely possible for the Data Owner (or Attacker) to change the code in the Cloud Server and perform malicious activities (like directing all queries of query users to himself (Data Owner or Attacker)). In Zhu *et al.* (2016) scheme, Cloud Server or Query User can change the database with obtained secret matrix M and vector s (From attack 2 and 3 of chapter 3).

4.1.2 Collusion Attack Possible

Earlier Works (like Zhu *et al.* (2016), Singh *et al.* (2018), and Parampalli *et al.* (2019)) don't take care of collusion attacks. They assume that the three parties involved are honest, which might not be the case. There are two possibilities of collusion attacks.



q' = β_q M q̂
CS
q' = β_q M q̂
DO
QU
1. Divide q' by β_q (Say q'')
2. q̂ = M^{-1}q''
3. Obtain q from q̂ using R^q and π

Figure 4.1: When Data Owner colludes with Cloud Server (In Zhu *et al.* (2016) Scheme)

When Data Owner colludes with Cloud Server

In all 3 works (mentioned above), if Data Owner colludes with Cloud Server, then Query Privacy will be compromised. Cloud Server may give the query received from the Query User to the Data Owner. Then Data Owner can decrypt the query (Query Privacy lost).

1. **Zhu *et al.* (2016) Scheme:** Suppose cloud server gives $q' = \beta_q M \hat{q}$ to the Data Owner. Since Data Owner knows β_q, M, R^q and π, he can decrypt the entire query (Fig. 4.1).

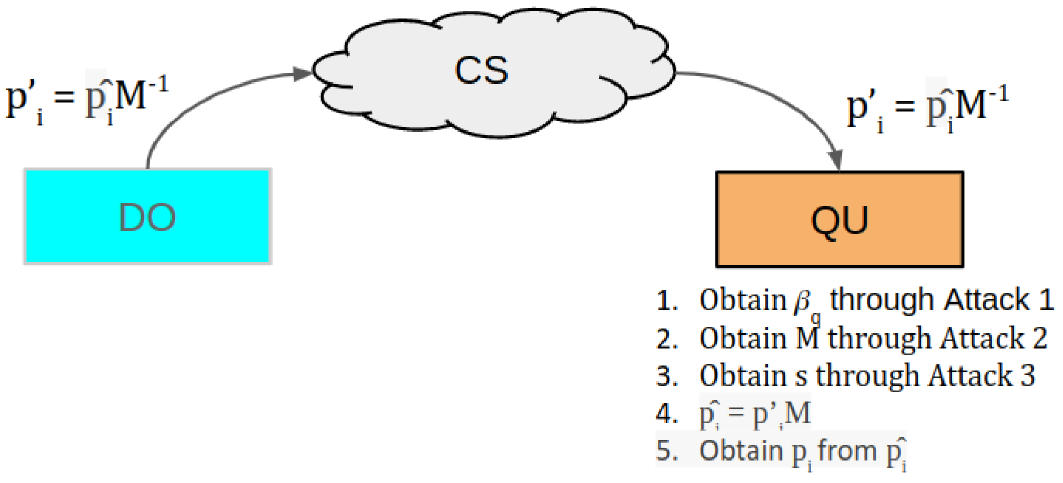
2. **Singh *et al.* (2018) Scheme:** Suppose the cloud server gives the encrypted query $\tilde{q} = W\left(M\left(\pi\left(\beta_q q_1, \beta_q q_2, \ldots, \beta_q q_d, \beta_q, R^q, 0_\varepsilon\right)\right), C^q\right)$ to the Data Owner. Since Data Owner knows β_q, W, M, R^q, and π, he can decrypt the entire query.

3. **Parampalli *et al.* (2019) Scheme:** Suppose the cloud server C_1 gives $q' = \widetilde{q} \times \gamma L_u M^\top$ to the Data Owner. Since Data Owner knows L_u, M and can get γ through gcd(q'), he can decrypt the entire query.

When Query User colludes with Cloud Server (or Cloud Server acts as Query User)

In Zhu *et al.* (2016) and Singh *et al.* (2018) scheme, if Query User colludes with Cloud Server (or Cloud Server acts as Query User), then Data Privacy will be compromised. Cloud Server may give the encrypted database received from the Data Owner to the Query User. Then Query User can decrypt the query (Data Privacy lost).

1. **Zhu *et al.* (2016) Scheme:** As seen earlier in chapter 3, Query User can obtain β_q, M and s through attack 1, attack 2 and attack 3 respectively. With M, s, and



CS
$p'_i = \hat{p_i}M^{-1}$
$p'_i = \hat{p_i}M^{-1}$
DO
QU
1. Obtain β_q through Attack 1
2. Obtain M through Attack 2
3. Obtain s through Attack 3
4. $\hat{p_i} = p'_i M$
5. Obtain p_i from $\hat{p_i}$

Figure 4.2: When Query User colludes with Cloud Server (In Zhu *et al.* (2016) Scheme)

encrypted database (got from cloud), Query User can decrypt the entire database (Fig. 4.2).

2. **Singh *et al.* (2018) Scheme:** Suppose the cloud server gives W matrix and encrypted Database to the query user. Now the query user can perform a chosen-plaintext attack similar to attack 2 (discussed in chapter 3). He will send (N, 0, 0, ..., 0) and get $\tilde{q} = W\left(M\left(\pi\left(\beta_{q_1}N, 0, \ldots, 0, \beta_{q_1}, R^q, 0_\varepsilon\right)\right), C^q\right)$. He can use W (got from cloud server) to get $\dot{q} = \left(M\left(\pi\left(\beta_{q_1}N, 0, \ldots, 0, \beta_{q_1}, R^q, 0_\varepsilon\right)\right), C^q\right)$. Now,

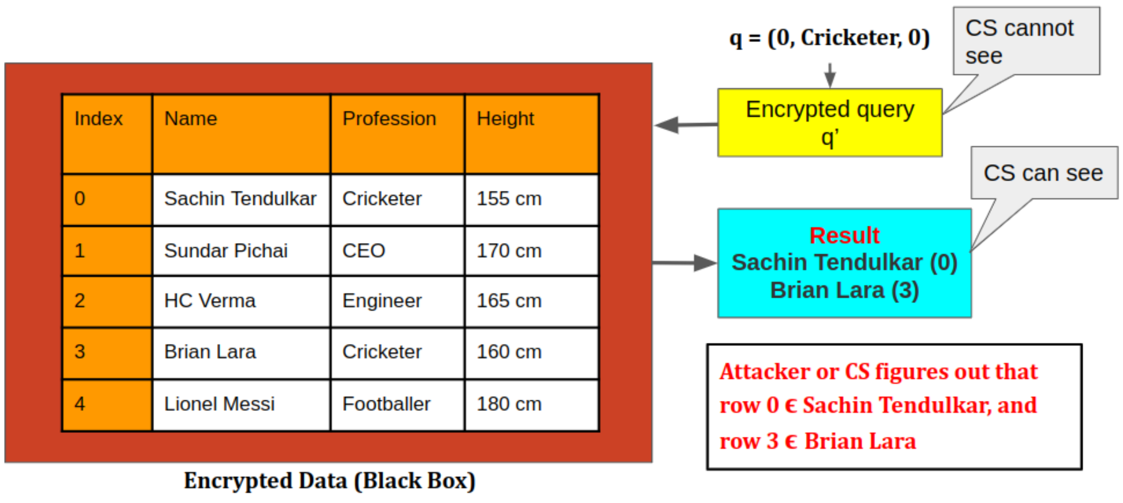
$$\beta_{q_1} \times M_1 = \frac{\dot{q} - (\dot{q}\%N)}{N}$$

where M_1 represents column 1 of M. We can obtain β_{q_1} by gcd of all elements of the result vector (obtained from above formula) and then can use this β_{q_1} to obtain column 1 of M. Similarly, we can obtain the entire M vector. Now we can use this M to get s (attack 3 of chapter 3). With M, s, and encypted database (got from cloud server), Query User can decrypt the entire database in clear.

We have implemented Singh *et al.* (2018) paper and collusion attacks on it. We got the entire database and query in clear through these attacks.

When Cloud Server C1 acts as Query User

In Parampalli *et al.* (2019) scheme, if Cloud C1 acts as Query User, then he has both classification label encryption key (C_u) and classification label decryption key(Q_u). Cloud C1 has v'_i (2-D encrypted classification label) from earlier steps. Now, Cloud C1 can multiply v'_i, C_u and Q_u^{-1} together to get decrypted classification label c_i. Thus, access pattern hiding will be compromised if cloud C1 acts as Query User.



Index	Name	Profession	Height
0	Sachin Tendulkar	Cricketer	155 cm
1	Sundar Pichai	CEO	170 cm
2	HC Verma	Engineer	165 cm
3	Brian Lara	Cricketer	160 cm
4	Lionel Messi	Footballer	180 cm

q = (0, Cricketer, 0)
CS cannot see
Encrypted query q'
CS can see
Result Sachin Tendulkar (0) Brian Lara (3)
Attacker or CS figures out that row 0 ∈ Sachin Tendulkar, and row 3 ∈ Brian Lara
Encrypted Data (Black Box)

Figure 4.3: Missing Access Pattern Hiding

4.1.3 Same Key Vulnerability

In Zhu *et al.* (2016) work, every query is encrypted with the same matrix M. If any query user obtains M through attack 1 and attack 2 (Chapter 3), he can decrypt all queries of all query users from past to future.

4.1.4 Lacks Access Pattern Hiding

Cloud Server should not be able to use KNN results to figure out corresponding encrypted database records. Earlier works (like Zhu *et al.* (2016) and Singh *et al.* (2018) work) return KNN computation results in clear. There are two issues with this:

1. Cloud Server can modify the returned result of KNN computation.

2. Cloud Server can figure out data/query from returned results.

Let us understand the 2nd point with an example (Fig. 4.3).
E.g., Query User generates a query to find out the top 2 cricketers in the Encrypted Database in the cloud. He sends the encrypted form of the query (jointly made by him and the Data Owner) to the cloud. Cloud Server cannot see the encrypted query. In Zhu *et al.* (2016) and Singh *et al.* (2018) work, the cloud server performs the KNN computation and returns the indices of the k-nearest database tuples. Suppose the result of the above query is index 0 and index 3. Then they would return 0 and 3, which doesn't make sense. In a real-life scenario like social networking database, they would have to return names, email IDs, etc. So if they return the names (or email ids) in the clear, like Sachin Tendulkar and Brian Lara, then the cloud server will figure out that row 0 belongs to Sachin Tendulkar

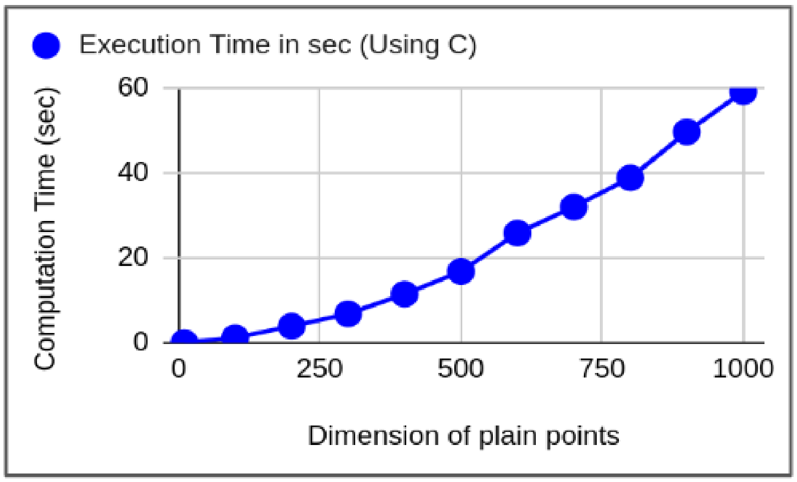


Figure 4.4: Paillier Encryption Overhead at Data Owner Side (Using C)

and row 3 belongs to Brian Lara. In this way, the cloud server would be able to perform an analysis of returned results and figure out the database and query.

4.1.5 Query Controllability Missing

In Zhu *et al.* (2016) scheme, the Query User can get secret β_q and secret M through attack 2 and attack 3, respectively (Chapter 3). Once we get secret M, we can launch any number of queries without going to the Data Owner. In Parampalli *et al.* (2019) scheme, once the Query User gets his unique query encryption key k_u from Data Owner (at the start), he can launch any number of queries without Data Owner's approval.

4.2 Performance Vulnerability

4.2.1 Paillier Encryption Overhead

Zhu *et al.* (2016) and Singh *et al.* (2018) work depend on Paillier Encryption to achieve query controllability and key confidentiality simultaneously. But from experiments, we found that Paillier Encryption takes too much time (Fig. 4.4). Data Owner will need to spend around 60 seconds to compute $E_{pk}\left(\beta_q M\hat{q}\right)$ when the number of query dimensions is 1000 (in the case of Zhu *et al.* (2016) work). In Singh *et al.* (2018) work, Data Owner will have to spend additional 60 seconds to obtain B^q. This will put too much burden on Data Owner, who will have to spend 60 sec and 120 sec per query per query-user in Zhu *et al.* (2016) and Singh *et al.* (2018) work, respectively. These schemes will not scale well if millions of queries per second or millions of users are using the system.

4.2.2 Matrix Multiplication Overhead

The previous works (using Homomorphic Encryption) involve many matrix multiplications, which is an overhead. E.g., Parampalli *et al.* (2019) work involves a lot of matrix multiplication (big overhead for large dimension query).

4.3 Availability Vulnerability

Zhu *et al.* (2016) and Singh *et al.* (2018) scheme require Data Owner to be online all the time, which is not practical. If the system of Data Owner goes down, the whole scheme will come to a standstill. In Parampalli *et al.* (2019) scheme, Data Owner has to remain online till all query user's are registered. Otherwise, only users registered at start can use the service.

Chapter 5

Our Proposed Design

5.1 System Model

In our design, there are three parties involved:

1. **Cloud Server:** Cloud Server is a third-party service provider which provides data storage and computation resources to Data Owner. In our design, there are two enclaves present on the cloud server. These two enclaves can be on the same or two different cloud servers. In our scenario, we have both enclaves on the same servers. Our main idea is to decouple the identity of the query user and the actual query so that it is infeasible for anyone to determine the query submitted by a query user. So, we have used two secure enclaves in our design:

   (a) Enclave E1 is responsible for database storage and KNN computation. Enclave E1 belongs to the Data Owner to protect Data Privacy. So Data Owner installs the KNN code in enclave E1.

   (b) Enclave E2 is responsible for interfacing with the query user and processing user (micro) payment. Enclave E2 belongs to the trusted 3rd party to preserve Query Privacy by decoupling Query User's identity from their query.

2. **Data Owner:** Data Owner is the entity (or person) who owns the database. In our case, Data Owner has a database D consisting of m database tuples, each tuple being d-Dimensional. To protect data privacy, the Data owner encrypts this database and sends it to enclave E1, where it is decrypted and stored.

3. **Query Users:** Query Users are those entities who want to query the Data Owner's data and get the correct result (k-NN result in our case) for their d-Dimensional query point. In our design, query users send their encrypted query to Enclave E2,

which forwards it to Enclave E1 for kNN computation. Enclave E2 gets the kNN result from Enclave E1 and sends it back to the Query User.

5.2 Threat Model

Almost all the previous works considered Data Owner, Cloud Server, and Query User as honest-but-curious, which means each party will follow the steps strictly and return correct computation results. At the same time, each party will try to infer as much information about others' data as possible from the data they get from others. But this assumption is not realistic. Data Owner may collude with Cloud Server, or Cloud Server may act as Query User, etc. We have not made such an assumption (like honest-but-curious).

5.3 Assumptions

In our design, we assume the following essential things (which is true in previous designs as well):

1. Data Owner has sent the correct database to the secure enclave in the cloud.
2. The cloud server provides its services properly, i.e., it doesn't block the data owner from sending its database into the secure enclave. Similarly, it doesn't block the query user's query.

We assume that the interfacing code in Enclave E2 is minimal, open-source (or known to everyone), and installed by a trusted 3rd party (similar to a Certificate Authority). We assume that several parties have verified this code. Once it is loaded in the enclave and verified, it cannot be changed by anyone as enclaves ensure the integrity of the code. In our proposed design, the two enclaves do not collude as Enclave E2 code is minimal and verified to be correct by multiple parties.

5.4 Power of our Proposed Design

In our proposed design, we have used kNN computation as an example. Our design is a general one. Ours can also be used to solve other real-life secure cloud computing problems. E.g., Suppose we have an ML application that takes the query of Query User as input. The query may be the medical record of the Query User. Now the application needs to process this medical record and gives some result like the Query User is OK or he may be suffering from heart disease, etc. This querying business should happen in such a way that the privacy of the querying user is preserved. Our design can be used in such a scenario. If our design is used, the Application Developer (analogous to Data Owner) will install the ML application (discussed above) instead of the kNN code on Enclave E1. All other activities will remain almost the same in our design. Instead of kNN computation, some other computation will be performed in Enclave E1 and the result will be returned to Enclave E2 which will forward it to Query the User.

5.5 Stages of Proposed Design

The database point $p = (p_1, p_2, \cdots, p_d)$ is stored as follows in our design:

$$\dot{p} = \left(-2p_1, -2p_2, \cdots, -2p_d, ||p||^2\right)$$

The query point Q in our design is as follows:

$$Q = (q_1, q_2, \cdots, q_d, 1)$$

5.5.1 Sending Database to Enclave E1 (in Cloud Server) in a secured manner

This is a one-time work done by the data owner (Fig. 5.1).
Steps:

1. Data Owner installs the application (considered untrusted) on the cloud, which creates an enclave E1 (considered trusted) on the cloud. This enclave E1 contains the KNN code. So the KNN code is successfully installed in Enclave E1.

2. Data Owner asks for an attestation report from enclave E1 to prove its integrity, i.e., to prove that it contains the KNN code installed by the Data Owner and not by Cloud Server or any attacker. This process is called Remote Attestation [Intel SGX Report is similar to a certificate issued by the certificate authority. Just as the certificate proves the identity and authenticity of a website (or other entities), SGX Report proves the integrity of the code and data inside the SGX enclave. It proves that the data and code inside the enclave haven't been tampered with by Cloud Server or attackers. We can also share encryption/decryption keys generated inside enclaves using the user data field of this report].

3. On receiving the request for the report, it generates a report (called Quote) and an RSA key pair (PK_{E1}, SK_{E1}). It keeps the secret key SK_{E1} within itself and sends the report along with the public key PK_{E1} to Data Owner.

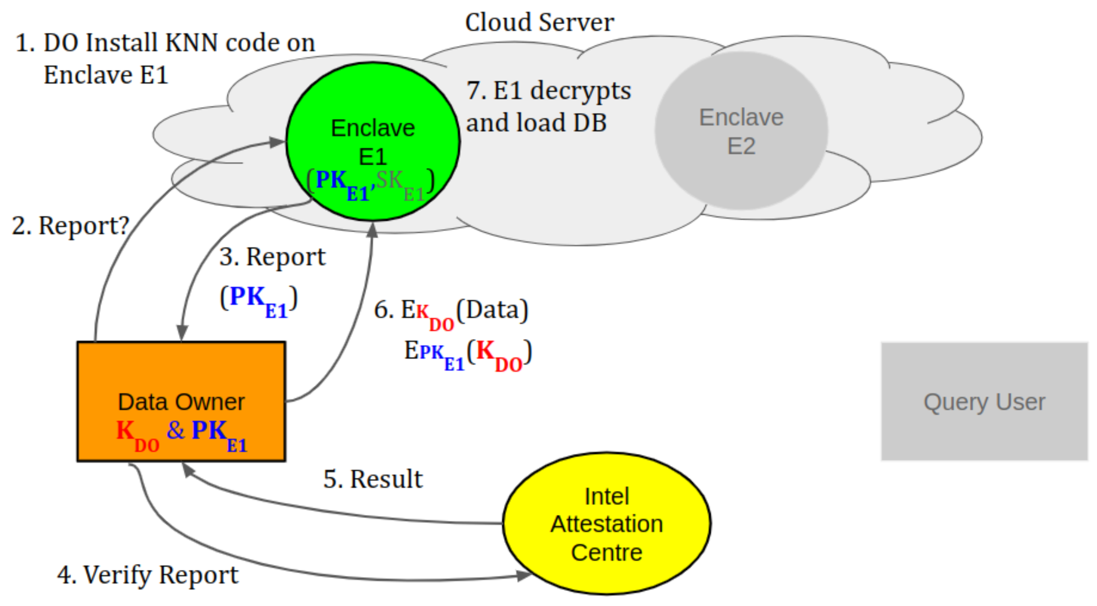


Figure 5.1: Sending Database to Enclave E1 (in Cloud Server) in a secured manner

4. Data Owner sends the report (called Quote) to the Intel Attestation Centre for verification.

5. Intel Attestation Centre replies with whether the report is valid or not.

6. Data Owner generates an AES key K_{DO} to encrypt the database. It needs to give the AES key K_{DO} to Enclave E1 for decrypting the database. So it sends the database (encrypted with K_{DO}) and AES key K_{DO} (encrypted with public key PK_{E1} of enclave E1). So Data Owner sends the following to Enclave E1.

   (a) $E_{K_{DO}}(Database)$

   (b) $E_{PK_{E1}}(K_{DO})$

7. Enclave E1 decrypts the $E_{PK_{E1}}(K_{DO})$ with SK_{E1} (which it had from step 2) and gets the K_{DO}. Now enclave E1 has the AES key K_{DO}. Now it can decrypt the encrypted database using the AES key K_{DO}. This decryption of keys and database happens inside the enclave. So it is not visible to any attacker or Cloud Server.

5.5.2 Verifying each others' (enclaves') integrity

This is a one-time work done by both enclaves (Fig. 5.2).

Steps:

1. A trusted 3rd party (similar to Certificate Authority) installs the application (considered untrusted) on the cloud, which creates an enclave E2 (considered trusted) on

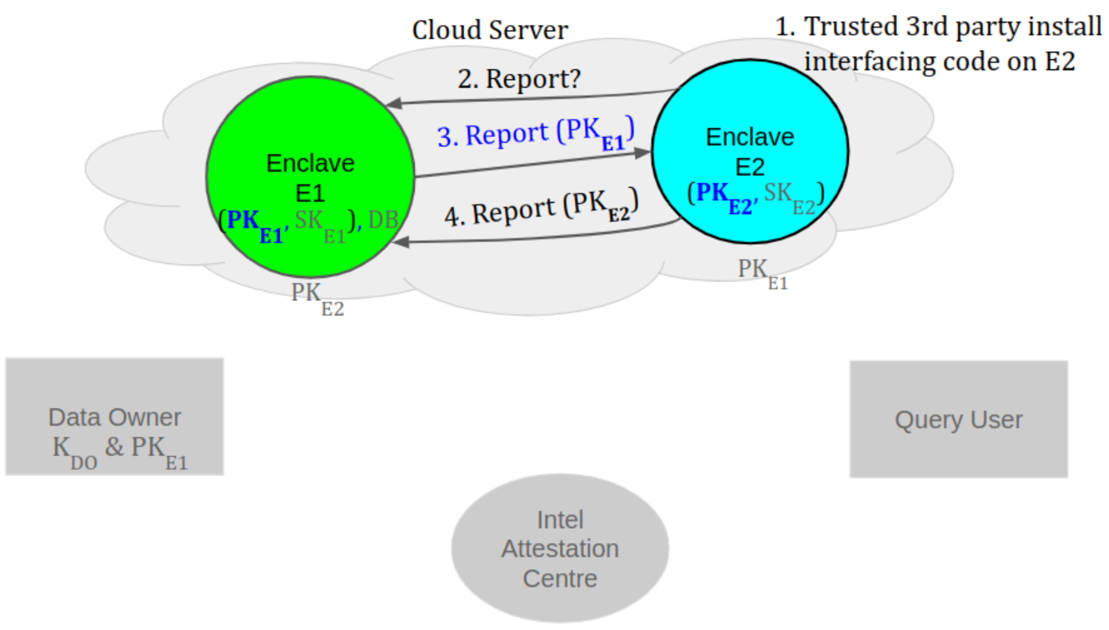


Figure 5.2: Verifying each others' (enclaves') integrity

the cloud. This enclave E2 will have minimal interfacing code (known to everyone and verifiable by everyone). Anyone can verify it through attestation.

2. Enclave E2 asks for an attestation report from Enclave E1 to prove its integrity, i.e., to confirm that Data Owner installs it and not CSP or any other attacker (Local Attestation if they are running on the same processor, otherwise, Remote Attestation).

3. On receiving the report request from E2, Enclave E1 (on the cloud) generates a report. It sends the report along with its Public key PK_{E1} (generated earlier) to enclave E2.

4. On receiving the report, Enclave E2 generates its report and an RSA key pair (PK_{E2}, SK_{E2}). It sends its report along with its Public key PK_{E2} to enclave E1.

If both of them run on the same processor, they can verify the reports locally and get the keys. Otherwise, if they are running on different processors (or on different clouds), they need to go to Intel Attestation Centre to verify the report). In the end, they both have each other's public keys.

5.5.3 Query User verifies Enclave E2's integrity

This is a one time work done by each query user (Fig. 5.3).

Steps:

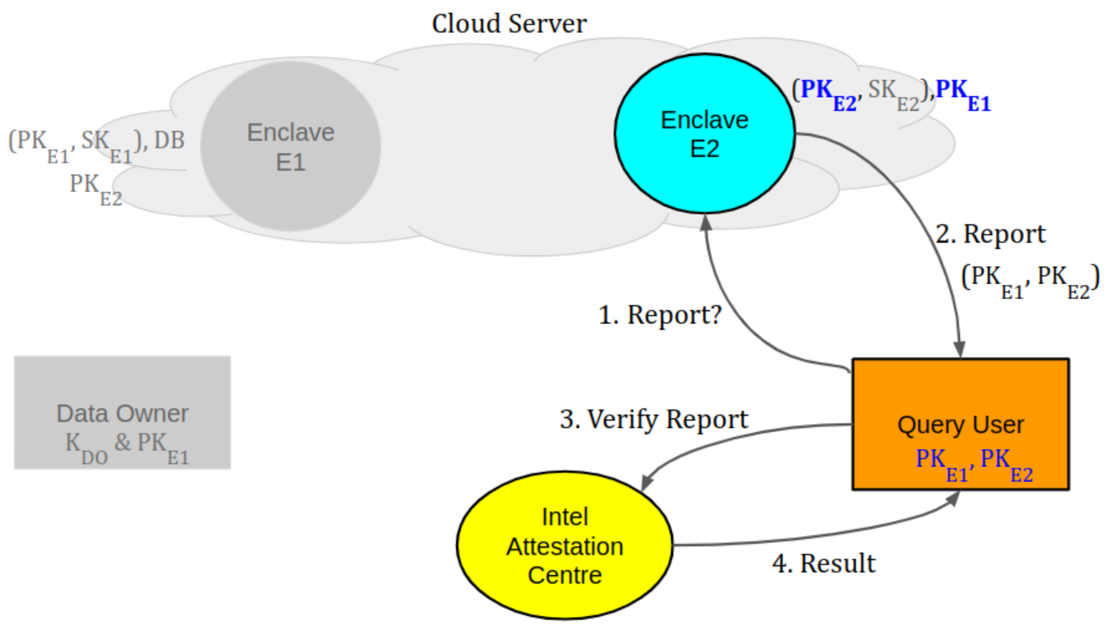


Figure 5.3: Query User verifies Enclave E2's integrity

1. Query User asks for an attestation report from Enclave E2 to prove its integrity, i.e., that it has been installed by a trusted 3rd party and not CSP or any other attacker (Remote Attestation). This step is required only once per query user (as the registered Query User will get enclave E2's public key (PK_{E2}) for later use).

2. On receiving the report request from the Query User, Enclave E2 (on the cloud) generates a report. It sends its report along with the Public key PK_{E1} (of enclave E1) and its public key PK_{E2} to the Query User.

3. QU sends the report to the Intel Attestation Centre for verification.

4. Intel Attestation Centre replies with whether the report is valid.

So now, the QU also has PK_{E1} of enclave E1 and PK_{E2} of enclave E2.

5.5.4 Query User sends query to enclave E1 through enclave E2

This work is done per query by Query User (Fig. 5.4).
Steps:

1. Query User generates its query Q, an RSA key pair (SK_Q, PK_Q), and a unique AES key K_{QU} (different for every query).

2. Query User encrypts the query with its AES key K_{QU}. This encrypted query needs to be sent to enclave E1 through enclave E2. But E1 will need the key K_{QU} to

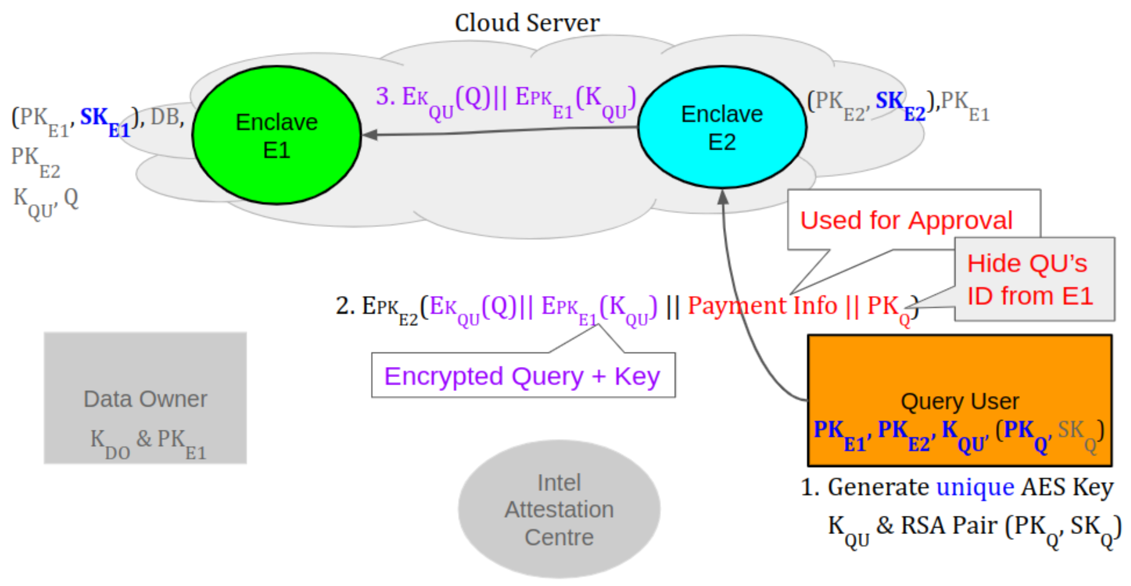


Figure 5.4: Query User sends query to enclave E1 through enclave E2

decrypt the query to perform KNN computation. So QU has to send K_{QU} as well. But this K_{QU} should not be visible to enclave E2. Otherwise, it can see the query in clear. Another thing to note is that we need to send the credentials and encrypted query in an encrypted form such that it is not visible to attackers or Cloud Server. So finally, we sent the query in the following way to E2:

$$E_{PK_{E2}}\left(E_{K_{QU}}(Q)\|E_{PK_{E1}}(K_{QU})\|Credentials\_or\_Payment\_Info\|PK_Q\right)$$

3. Enclave E2 decrypts the above encrypted query and gets the following:

$$E_{K_{QU}}(Q)\|E_{PK_{E1}}(K_{QU})\|Credentials\_or\_Payment\_Info\|PK_Q$$

Based on credentials or payment info, it decides whether to allow query or not [Query Controllability]. Now, the enclave E2 sends the following to enclave E1:

$$E_{K_{QU}}(Q)\|E_{PK_{E1}}(K_{QU})$$

Only Enclave E1 can decrypt $E_{PK_{E1}}(K_{QU})$ using SK_{E1} and get K_{QU}. Using K_{QU}, it decrypts the encrypted query. Now, enclave E1 can see the query in the clear, but it doesn't know whose query it is (Query User's identity is decoupled from his query).

5.5.5 KNN computation and sending result (k-nearest neighbors) to Query User

This work is done by enclave E1 per query (Fig. 5.5).

Steps:

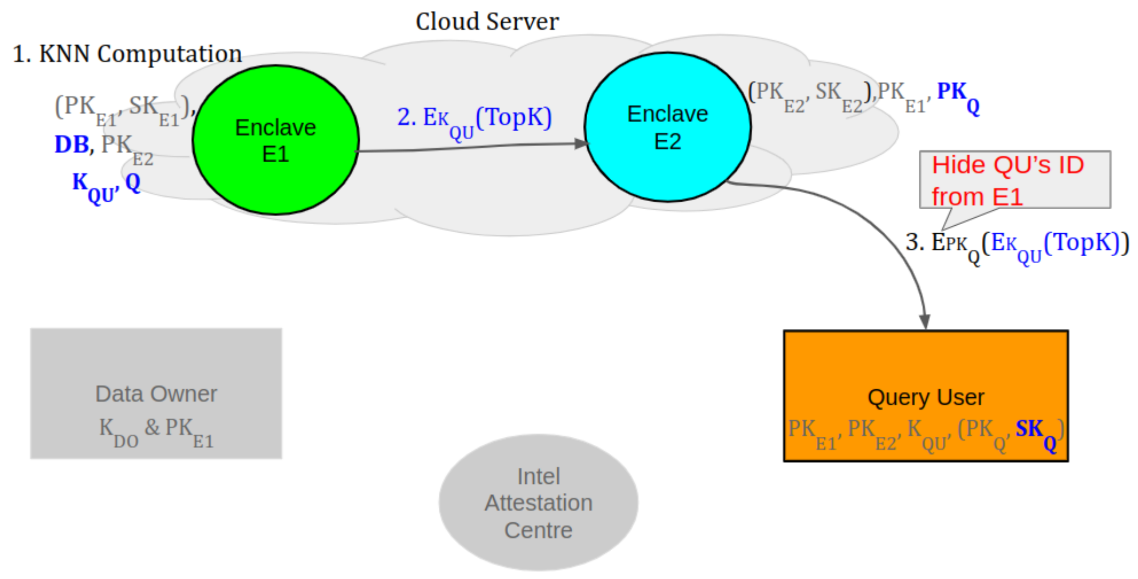


Figure 5.5: KNN computation and sending result (k-nearest neighbors) to Query User

1. Now, Enclave E1 has both database and query. It computes the KNN and obtains the results.

2. It encrypts the result with K_{QU}, so only QU can decrypt it. Now it returns the encrypted result $E_{K_{QU}}(Result)$ to Enclave E2.

3. Enclave E2 (or anyone else) cannot decrypt the result. E2 sends the $E_{PK_Q}(E_{K_{QU}}(Result))$ to the QU, who can decrypt it to get the result since it has both SK_Q and K_{QU}. $E_{K_{QU}}(Result)$ needs to be encrypted with PK_Q. Otherwise, Enclave E1 can link $E_{K_{QU}}(Result)$ to Query User (As he knows $E_{K_{QU}}(Result)$ and can tap the line between Enclave E2 and Query User to see to whom the result is returned).

Note: Query Users need not use the same K_{QU} again and again. Using a new randomly generated AES key every time will be advisable (as we want per query security).

5.6 Pros and Cons of the Proposed Design

5.6.1 Pros of the Proposed Design

Code Integrity

Data Owner installs the KNN code in Enclave E1. He then performs software attestation to verify the integrity of the code. He gives the database to the enclave only when he is satisfied. SGX will detect any modification in the code later.

Data Integrity

SGX ensures the integrity of the database. Modification by attacker or cloud is possible, but it will be detected. In other words, we are guaranteed that the data is integrity-protected because, otherwise, SGX would stop the system.

Code Confidentiality

In our design, we have not provided code confidentiality as there is no need for code confidentiality in the case of KNN code as it is simple and known to everyone. But code confidentiality can be added to Enclave E1 without worrying about Enclave E1 (an extension of Data Owner) giving queries to Data Owner or others. Even if it provides the query to the Data Owner or others, they won't be able to link Query User's identity to that query. To provide code confidentiality, the KNN code can be sent in encrypted form to Enclave E1 (along with the database). We will need a loader inside the enclave to decrypt and load the code. The concept is similar to the use of Shield module decrypting and loading the encrypted code in Baumann *et al.* (2015).

Data Privacy

Enclave E1 belongs to the Data Owner. The code is installed in it by Data Owner and has been verified through software attestation. So it is the trusted code. Data Owner can send the database in encrypted form to enclave E1. The enclave E1 will decrypt the code inside itself (as only it has the key). Once the data is decrypted inside the enclave, it will be present inside the enclave. No one from outside can see it. So, data privacy is guaranteed by the secure enclave.

Key Confidentiality

The data is encrypted by an AES key K_{DO} of the data owner. This key is sent to the enclave E1 in encrypted form (encrypted by its public key). So only the enclave E1 can decrypt and get the AES key with which the Database is encrypted. So Data Owner's key is confidential.

Query Controllability

In earlier works like Zhu *et al.* (2016) and Singh *et al.* (2018) work, Data Owner decides whether to allow the query or not. But the criteria of decision is what? Data Owner cannot see the query. He has to allow the Query User's query based on his credential (or some payment information). This task can be given to Enclave E2. Enclave E2 can decide whether to allow the query or not based on some payment information.

Query Privacy

Enclave E2 receives the query from the query user in encrypted form. It cannot decrypt the query. It sends the encrypted query to Enclave E1, which decrypts and performs the KNN computation. Enclave E1 gets only the encrypted query and not the credential of the query user from Enclave E2. So E1 cannot link Query User's identity to their query. In other words, Query Privacy is preserved.

Access Pattern Hiding

The KNN computation happens inside the enclave E1. So Cloud Server or the attacker cannot see the result of the KNN computation. In other words, he cannot use the KNN results to figure out corresponding encrypted database tuples. Also, the result is sent in encrypted form to the query user. Only Query User can decrypt the final result. So no one else can see the result as well.

Per Query Key

Each query is encrypted with a different AES key. So even if one key is lost, the privacy of other queries is preserved.

Data Privacy and Query Privacy against collusion attack

Enclave E1 belongs to the data owner. So it won't collude with query users or cloud servers. So data privacy is guaranteed against collusion attacks. Enclave E2 has open-source minimal interfacing code installed and verified by various trusted parties. So E2 won't collude with anyone else. Thus Query Privacy is guaranteed against collusion attacks.

Data Owner need not be online

The task of KNN computation is taken care of by Enclave E1. And the query controllability is handled by Enclave E2. So Data Owner need not be online.

Paillier Encryption multiple matrix multiplications are not needed

The beauty of our scheme is that it uses the AES scheme, which is very fast and secure. We need not worry about the cache-based side-channel attacks because, in recent processors, AES is done on hardware without needing the cache. Also, we don't use fancy matrix multiplications like those used in earlier works.

5.6.2 Cons of the Proposed Design

Couldn't find any security, performance or availability related cons.

5.6.3 Other considerations

Our design is not limited to Intel SGX. We can use any other secure enclave as well. We have used Intel SGX because it came free on our Intel core processor. So, if we are using Intel SGX then:

1. Intel SGX support needed on the cloud server.

2. We have to trust Intel for software attestation (Till SGX version 1). In SGX version 2, we can perform remote attestation without going to Intel Attestation Centre. But for this, extra infrastructure will be needed on the client side providing attestation service.

3. Intel SGX is considered vulnerable to cache-based side-channel attacks. Several successful cache-based side-channel attacks have been launched on AES encryption happening inside SGX Enclaves. But, nowadays, AES encryption/decryption are performed on hardware without need of cache. So, our design is not vulnerable to cache-based side-channel attack on AES encryption inside SGX enclave.

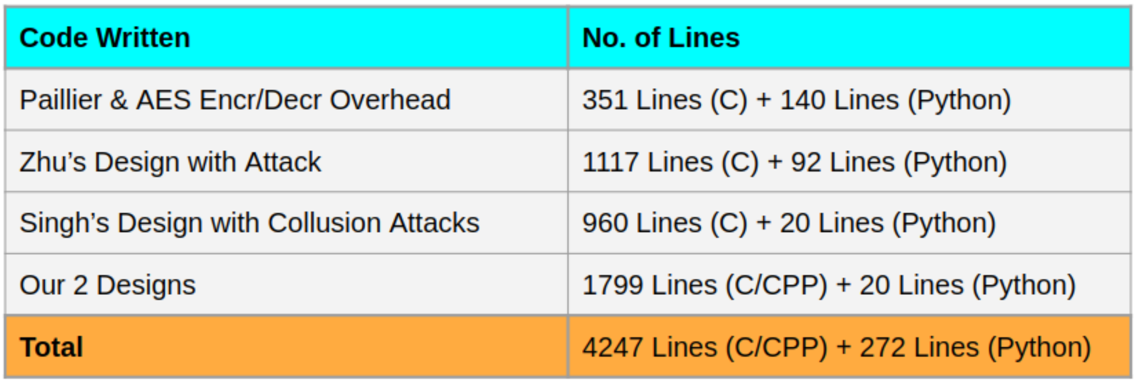
4. SGX reserves only 128 or 256 MB for enclaves as protected space because Intel developed SGX to allow small and sensitive computation inside isolated regions called enclaves. The issue is what happens when we need more space. E.g., our design requires much more space than just 128 MB. Windows don't support more than 128 or 256 MB. But, Linux systems can support arbitrarily large memory space for enclaves by using the paging concept, similar to the paging concept in OS (E.g., we set around 3GB space for enclaves in our Linux system). Whenever the data size exceeds the protected region size in RAM, the encrypted pages are swapped out to the unprotected space. But they are still in encrypted form. Whenever needed, they are swapped into the protected area (in RAM) and checked by the processor

(for integrity). If it is okay, then the processor can use it. So, the Linux system supports arbitrary large enclaves, but there is also a downside. This swapping in and swapping out of pages deteriorates performance. We want to make it clear again that our design is not restricted to using Intel SGX. We could also use other enclaves like AMD's SEV, etc. Or a better enclave may be developed in the future, providing ample protected memory space. Thus, our design is still a practical and powerful one.

Chapter 6

Performance Evaluation

We implemented our proposed scheme and two earlier schemes: Zhu *et al.* (2016) and Singh *et al.* (2018), using C language. To implement Zhu *et al.* (2016) and Singh *et al.* (2018) work, we used the GNU MP library and Paillier library with a 1024 bits key size. We selected the security parameters c = 2, $\epsilon = 2$ for Zhu *et al.* (2016) Scheme. Similarly, we selected the security parameters c = 2, $\epsilon = 2$ and l = 5 for Singh *et al.* (2018) Scheme. To implement our scheme, we used the GNU MP library, AES library, and Intel SGX (for secure enclaves). We performed all our experiments on a machine having Ubuntu 20.04 with Intel Core i5 2.5 GHz Processor and 8 GB RAM. We also implemented attacks on Zhu *et al.* (2016) scheme and collusion attacks on Singh *et al.* (2018) scheme. (Link to my code: https://github.com/aditya059/MTP_Code)



Code Written	No. of Lines
Paillier & AES Encr/Decr Overhead	351 Lines (C) + 140 Lines (Python)
Zhu's Design with Attack	1117 Lines (C) + 92 Lines (Python)
Singh's Design with Collusion Attacks	960 Lines (C) + 20 Lines (Python)
Our 2 Designs	1799 Lines (C/CPP) + 20 Lines (Python)
Total	4247 Lines (C/CPP) + 272 Lines (Python)

Figure 6.1: Code Written

6.1 Key Generation

Here, Key Generation Time refers to the time required for generating the key(s) with which the database of the Data Owner is encrypted.

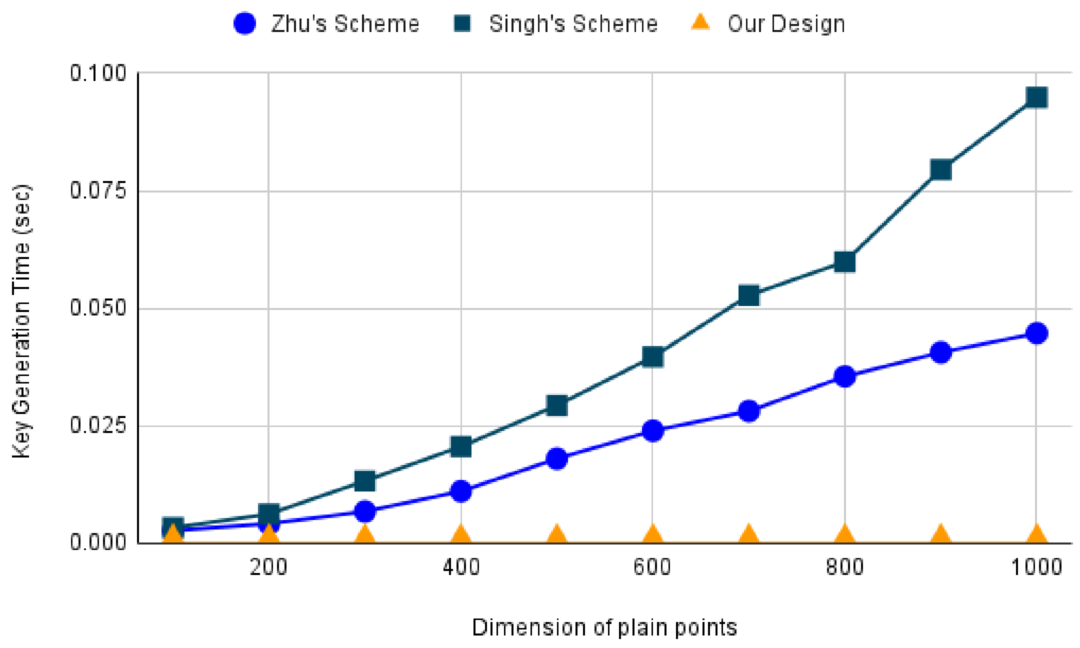


Figure 6.2: Average Key Generation Time (in sec)

6.1.1 Computational Complexity

In our scheme, the computational complexity for Data Owner during Key Generation is O(1) because Data Owner generates an AES Key only for encrypting the database. In Zhu *et al.* (2016) scheme, the computational complexity for Data Owner during this stage is $O(\eta^2)$ because the costliest step is the generation of $\eta \times \eta$ invertible matrix M. Here, $\eta = d + 1 + c + \epsilon$. In Singh *et al.* (2018) scheme, the computational complexity for Data Owner during this stage is $O(n^2)$ because the costliest step is the generation of $n \times n$ invertible matrix W. Here, $n = \eta + l$.

6.1.2 Experiment Results

We evaluated the time for key generation in our design, Zhu *et al.* (2016), and Singh *et al.* (2018) schemes. Fig. 6.2 shows the average key generation time over ten runs, with the dimension of data points varying from 100 to 1000. Our scheme needs to generate a random AES key, so our design will take constant and negligible time. On the other hand, key generation time in Zhu *et al.* (2016) and Singh *et al.* (2018) schemes increases with the dimension of data points. Since Singh *et al.* (2018) scheme generates two large matrices compared to one in Zhu *et al.* (2016) scheme, it takes longer.

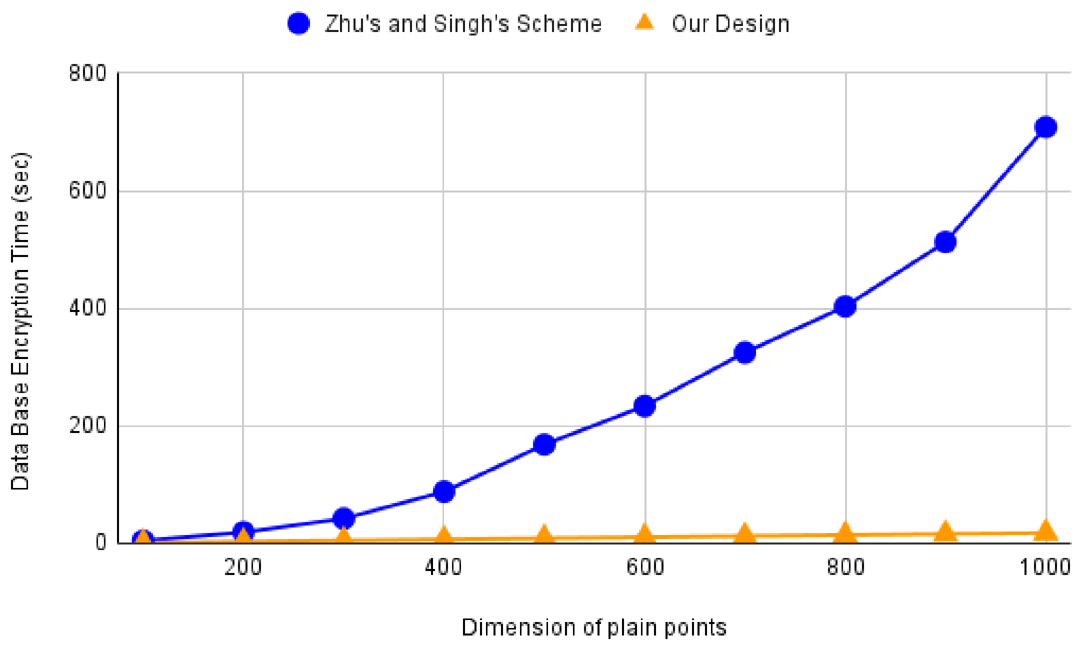


Figure 6.3: DataBase Encryption Time (in sec) when number of tuples = 10000

6.2 Database Encryption

6.2.1 Computational Complexity

In our scheme, the computational complexity for Data Owner during Database Encryption is O(md) because Data Owner encrypts each database point (m points) having d features each, using AES encryption. In Zhu *et al.* (2016) and Singh *et al.* (2018) scheme, the computational complexity for the Data Owner during this stage is O($m\eta^2$). The costliest step is the multiplication of m data points with an $\eta * \eta$ invertible matrix M.

6.2.2 Experiment Results

We evaluated the time to encrypt a random database consisting of 10000 tuples. We compared our design with both Zhu *et al.* (2016) and Singh *et al.* (2018) schemes. Fig. 6.3 shows the database encryption time, with the dimension of data points varying from 100 to 1000. This stage is the same for both Zhu *et al.* (2016) and Singh *et al.* (2018) schemes. So their line graphs match. Since our design uses AES encryption instead of matrix multiplication, ours take far less time than their design. Our design takes just 18 sec compared to 708 sec in their scheme when the dimension of data points is 1000.

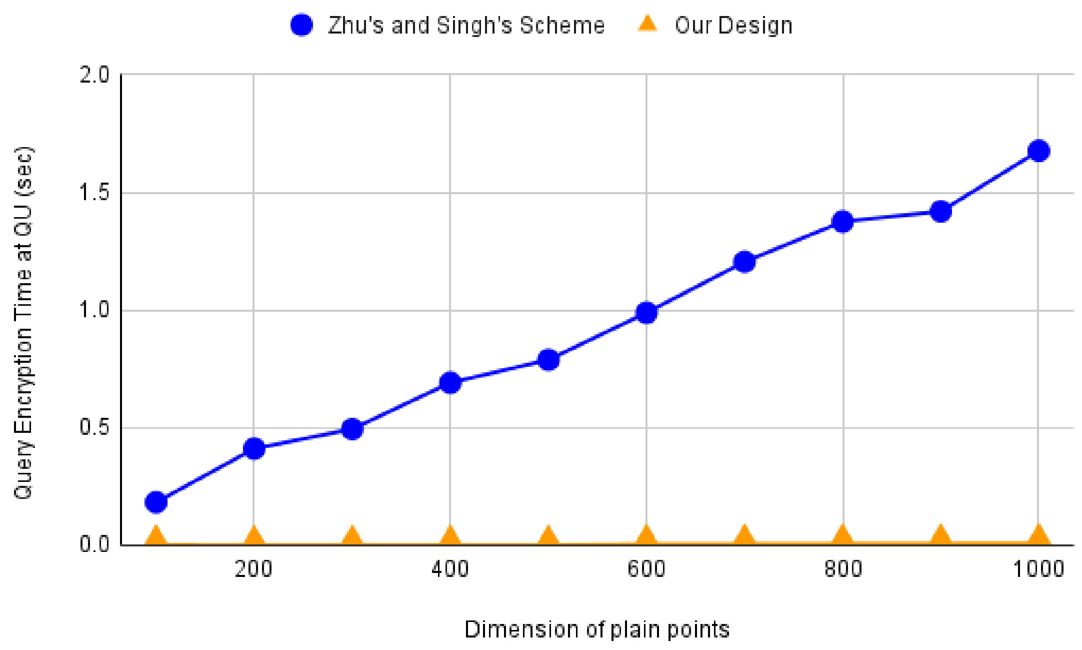


Figure 6.4: Average Query Encryption Time at QU's side (in sec)

6.3 Query Encryption at Query User

6.3.1 Computational Complexity

In our scheme, the computational complexity for Query User during Query Encryption is O(d) because the costliest step is the encryption (using AES) of query point with d dimensions. In Paillier Cryptosystem, one encryption takes O(log N) computation time (where N = pq). So, in Zhu *et al.* (2016) and Singh *et al.* (2018) scheme, the computational complexity for Query Encryption during this stage is O(d log N) because the costliest step is the encryption of the query point with d dimensions.

6.3.2 Experiment Results

We evaluated the Query Encryption time (at the Query User side) in our design, Zhu *et al.* (2016), and Singh *et al.* (2018) schemes. Fig. 6.4 shows the average Query Encryption time over ten runs, with the dimension of query points varying from 100 to 1000. In Zhu *et al.* (2016) and Singh *et al.* (2018) scheme, the encryption time is more as Paillier encryption takes more time. Our scheme needs to generate a random AES key and then encrypt the query point with this AES key. Our design takes very little time, as is visible in fig 6.4.

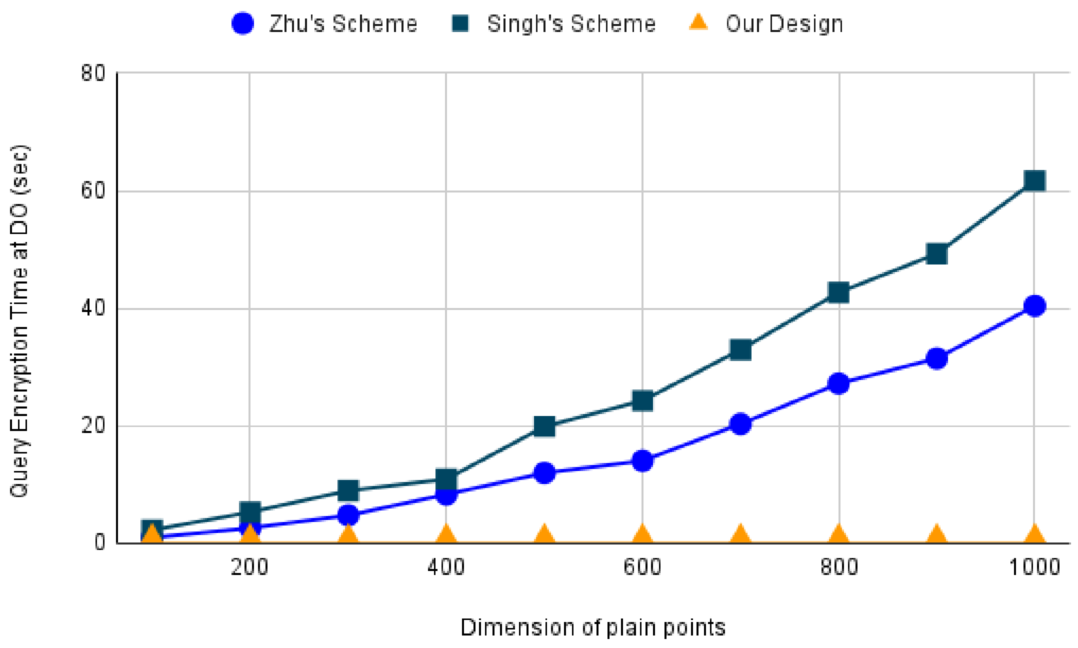


Figure 6.5: Average Query Encryption Time at DO's side (in sec)

6.4 Query Encryption at Data Owner

6.4.1 Computational Complexity

Our scheme doesn't require Data Owner for Query Encryption. In Zhu *et al.* (2016) scheme, the computational complexity of Query Encryption during this stage is O($\eta^2 + c\eta log\, N$) because the costliest step is the calculation of $A^{(q)}$, which requires O(η^2) multiplications and O(cη) encryptions (Paillier). In Singh *et al.* (2018) scheme, the computational complexity of Query Encryption during this stage is O($n^2 + c\eta\, log\, N + ln\, log\, N$) because the costliest step is the calculation of $A^{(q)}$ and then $B^{(q)}$, which requires O(η^2 + n^2) multiplications and O(cη + ln) encryption (Paillier).

6.4.2 Experiment Results

We evaluated the Query Encryption time (at the Data Owner side) in Zhu *et al.* (2016) and Singh *et al.* (2018) schemes. Fig. 6.5 shows the average Query Encryption time over ten runs, with the dimension of query points varying from 100 to 1000. In Zhu *et al.* (2016) scheme, $A^{(q)}$ calculation takes too much time. Since Singh *et al.* (2018) scheme calculates $A^{(q)}$ and then $B^{(q)}$ from $A^{(q)}$, it requires a lot more time. This puts too much pressure on the Data Owner. Data Owner has to spend 40 sec and 60 sec per query (per query user) in Zhu *et al.* (2016) and Singh *et al.* (2018) scheme, respectively. In our

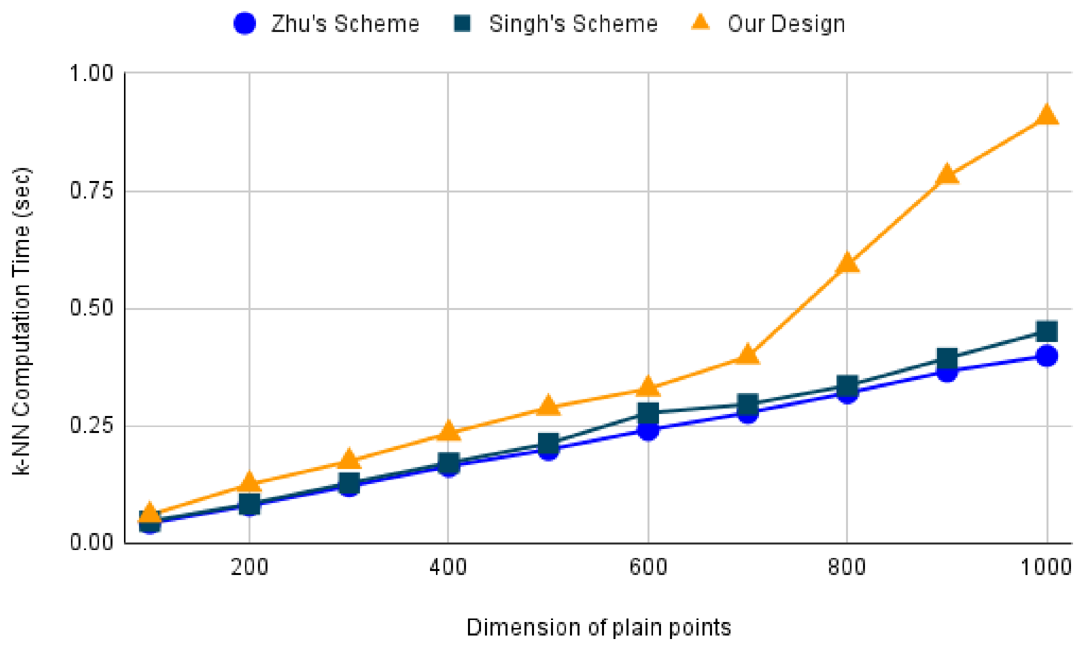


Figure 6.6: Average k-NN Computation Time (in sec) when number of tuples = 10000

design, Data Owner is not involved. So in our design time required on Data Owner's side per query is 0.

6.5 k-NN Computation

6.5.1 Computational Complexity

In our scheme, the computational complexity for Enclave E1 during k-NN computation is O(dm + m log k). The dot-product distance calculation requires O(d) for the dot product between each data and query point (total m data tuples need O(dm)). Calculating k-NN indexes requires O(m log k) as it can be done using modified heap sort. Similarly, in Zhu *et al.* (2016) scheme, the computational complexity at Cloud Server for this stage is O(mη + m log k). In Singh *et al.* (2018) scheme, the computational complexity at Cloud Server for this stage is O(mη + m log k + n^2) because of the extra multiplication of query point with matrix W at the start.

6.5.2 Experiment Results

We evaluated the time for k-NN computation in our design, Zhu *et al.* (2016), and Singh *et al.* (2018) schemes. We set k = 10 and m = 10000 (tuples). Fig. 6.6 shows the average k-NN computation time over ten runs, with the dimension of data points varying from 100 to 1000. Singh *et al.* (2018) scheme takes a longer computation time than Zhu *et al.* (2016) scheme because it involves multiplication with extra matrix W at the start. The time complexity of our scheme is lower than theirs. But our design takes a bit more time because of Intel SGX. Intel SGX allocates only 128 MB for secure enclaves. Since we require more, more time is wasted in paging the database in and out of the reserved 128 MB. We don't need to worry about the security as the pages are encrypted while moving out of the reserved memory. Our scheme takes 0.9 sec compared to 0.4 sec and 0.45 sec in the case of Zhu *et al.* (2016) and Singh *et al.* (2018) scheme, respectively, when the dimension of data points is 1000. So our design is still practical.

6.6 Other Overheads in our Design

In our design, we used Intel SGX for secure enclaves. We need to perform remote attestation between Data Owner and Enclave E1, Enclave E1 and Enclave E2, and Enclave E2 and Query Users. During this attestation process, RSA key pairs are generated and shared through the report (or quote) and used later to encrypt the messages between the involved parties for extra security. These attestation steps are one-time tasks only. For each query, an additional cost will be involved for AES and RSA decryption at Enclave E1 and Enclave E2. Similarly, the k-NN result will be encrypted and returned. These overheads will be far less compared to the overhead of Paillier encryption and, at the same time, will provide extra security.

Chapter 7

Comparison with Previous Works

Requirements	**Zhu's**	**Parampalli's**	**Singh's**	**Our Design**
Ensure Data Privacy	No	Yes	Yes	Yes
Ensure Query Privacy	No	Yes	Yes	Yes
Ensure Key Confidentiality	No	Yes	Yes	Yes
Ensure Query Controllability	No	No	Yes	Yes
Paillier Encr. Overhead	Moderate	No	High	No
Collusion Attack (Database Privacy)	Lost	Secure	Lost	Secure
Collusion Attack (Query Privacy)	Lost	Lost	Lost	Secure
Matrix Multiplication (Overhead)	Moderate	High	High	Low
Availability (If DO system down)	No	Yes	No	Yes
Ensure Code Confidentiality	No	No	No	Can be added
Ensure Code Integrity	No	No	No	Yes
Access Pattern Hiding	No	Yes	No	Yes
Same Key Used (Vulnerability)	Yes	Yes	Yes	No
1 or 2 Clouds (or Enclaves)	1	2	1	2

Table 7.1: Comparison with Previous Works

Chapter 8

Trials and Tribulations

8.1 MTP Stage 1

1. Intel SGX wasn't supported in the system allocated to me by the institute. It was released in 6th gen Intel Core processors, but my system (assigned) was 4th gen.

2. Intel has stopped providing SGX in 11th gen core processors.

3. Compared to most other applications, installation of SGX is non-trivial & complex.

4. It took me several days to install it on my laptop.

5. One of the papers I read Baumann *et al.* (2015) was challenging to understand as it contained many background details.

8.2 MTP Stage 2

1. I had to read and understand six different libraries for implementing various codes. These libraries weren't documented properly. So it wasn't easy to install and use these libraries.

2. Very few resources were available on the internet, making it difficult to debug my code in case of errors.

3. There was no library for Paillier Encryption in SGX Enclave. So I couldn't implement my design 3.

4. GMP (GNU Multiple Precision) library was not working inside the enclave. So I had to use standard C/CPP data types in my proposed design implementation, which could not handle data and query points of more than 64 bits. I wrote data owner's

code and query user's code using the GMP library but had to use standard C/CPP data types for kNN computation code in the enclave (on Cloud Server).

5. The code I wrote for Intel SGX was difficult to debug (in case of errors) as the error message was difficult to understand.

Chapter 9

Summary and Conclusions

Previous works (using homomorphic encryption alone) hadn't considered collusion between the cloud server and query user (or between the cloud server and data owner). They also failed to provide Code Confidentiality and Integrity. Their scheme needs Paillier encryption to simultaneously provide Query Controllability and Key Confidentiality and requires many matrix multiplications. We identified these vulnerabilities and proposed a new design (using secure enclaves) for performing secure k-NN computation on the cloud. Our design removes previous works' vulnerabilities and provides data privacy, query privacy, key confidentiality, query controllability, data integrity, code integrity, and access pattern hiding. Our scheme also improves performance by eliminating Paillier encryption and multiple matrix multiplications, which takes too much time. Our design uses AES encryption to improve performance without compromising security. Other benefits include preventing collusion attacks and using different keys per query. Also, Data Owner need not be online. Our scheme can also be extended to provide code confidentiality.

Appendix A

Programming in SGX

SGX applications are divided into two parts:

1. **Trusted Part:** Part of the application dealing with sensitive data is called Trusted Part and is executed inside an enclave. ECALL functions are defined here.

2. **Untrusted Part:** Remaining part of the application is called the Untrusted Part. This part creates an enclave and invokes them. OCALL functions are defined here.

Given below are some simple programs to show how codes are written for Intel SGX.

A.1 Enclave Interface comprising of ECALLs and OCALLs (Enclave.edl File)

```
/* Enclave.edl - Top EDL file. */
enclave {
    /* trusted - contains ECALL definitions
     * untrusted - contain OCALL definitions
     */
    trusted {
        public int ecall_add([in]int *a, [in]int *b);
        public int ecall_sumArray([in, count = len]int *num, size_t
    len);
        public void ecall_printPattern([in, out, count = len]char *
   pattern, size_t len);
    };
    untrusted {
        void ocall_print_string([in, string] const char *str);
        void ocall_ask_n([out]int *n);
    };
};
```

Code Snippet A.1: Enclave.edl

A.2 Creating and Destroying Enclave

```
/* Create the enclave */
ret = sgx_create_enclave(ENCLAVE_FILENAME, SGX_DEBUG_FLAG, NULL,
   NULL, &global_eid, NULL);
    if (ret != SGX_SUCCESS)
    {
        print_error_message(ret);
        return -1;
    }
/* Destroy the enclave */
// global_eid was the ID of created enclave
    sgx_destroy_enclave(global_eid);
```

Code Snippet A.2: Creating and Destroying Enclave

A.3 Defining and Invoking ECALLs

```
// ECALL definition in Enclave.cpp
// Take 2 nos. as input and return their sum
int ecall_add(int *a, int *b)
{
    return *a + *b;
}
// Take an array of nos. as input and return their sum
int ecall_sumArray(int num[], size_t n)
{
    int sum = 0;
    for (size_t i = 0; i < n; i++)
    {
        sum += num[i];
    }
    return sum;
}
// Take a string array as input and fill it with a pattern of '*'
// It makes an OCALL to untrusted part to get 'n'
void ecall_printPattern(char *pattern, size_t len)
{
    int i, j, k = 0, n;
    ocall_ask_n(&n);
    for (i = 0; i < n; i++)
    {
        for (j = 0; j <= i; j++)
        {
            pattern[k++] = '*';
        }
        pattern[k++] = '\n';
    }
}

// ECALL invocation from App.cpp
// Send 2 nos. and get back their sum in 'ans'
ecall_add(global_eid, &ans, &a, &b);
printf("a␣+␣b␣=␣%d\n", ans);

// Send array of n nos. and get back their sum in 'ans'
ecall_sumArray(global_eid, &ans, num, n);
printf("Sum␣of␣all␣nos.␣=␣%d\n", ans);

// Sent a string with garbage value and get a pattern in it
ecall_printPattern(global_eid, pattern, len);
printf("Pattern\n%s\n", pattern);
```

Code Snippet A.3: Defining and Invoking ECALLs

A.4 Defining and Invoking OCALLs

```
// OCALL definition in App.cpp
// Takes a string as argument and print it
void ocall_print_string(const char *str)
{
    /* Proxy/Bridge will check the length and null-terminate
     * the input string to prevent buffer overflow.
     */
    printf("%s", str);
}

// Enclave call it to get input from untrusted part
void ocall_ask_n(int *n)
{
    cout << "Enter␣the␣no.of␣lines" << "\n";
    cin >> *n;
}

// OCALL invocation from Enclave.cpp
// Ask untrusted part to send user input
ocall_ask_n(&n);

char buf[BUFSIZ] = {'\0'};
// Sends a string to untrusted part to print it
ocall_print_string(buf);
```

Code Snippet A.4: Defining and Invoking OCALLs

Appendix B

Attacks on Zhu *et al.* (2016) Scheme

B.1 Attack 1: Obtaining the secret β_q

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#define num_of_features 100
#define C 2
#define epsilon 2
#define eta (num_of_features + 1 + C + epsilon)

mpz_t decrypted_query[eta];

mpz_t betaQ;

void printBetaGenByDO()
{
    FILE *file;
    file = fopen("BetaGeneratedByDO.txt", "r");
    mpz_init(betaQ);
    gmp_fscanf(file, "%Zd", betaQ);
    gmp_printf("%s␣=␣%Zd\n", "Beta␣generated␣by␣DO␣=␣", betaQ);
    fclose(file);
}

void readDecryptedQuery()
```

```
{
    FILE *file;
    file = fopen("finalDecryptedQuery.txt", "r");
    for (int i = 0; i < eta; i++)
    {
        mpz_init(decrypted_query[i]);
        gmp_fscanf(file, "%Zd", decrypted_query[i]);
    }
    fclose(file);
}

void calcBeta()
{
    mpz_t gcd;
    mpz_init_set(gcd, decrypted_query[0]);
    for (int i = 1; i < eta; i++)
    {
        mpz_gcd(gcd, gcd, decrypted_query[i]);
    }
    gmp_printf("%s␣=␣%Zd\n", "Calculated␣Beta␣at␣QU␣=␣", gcd);
    mpz_clear(gcd);
}

void garbageCollection()
{
    mpz_clear(betaQ);
    for (int i = 0; i < eta; i++)
    {
        mpz_clear(decrypted_query[i]);
    }
}

int main()
{
    printBetaGenByDO();
    readDecryptedQuery();
    calcBeta();

    garbageCollection();
    return 0;
}
```

Code Snippet B.1: Obtaining the secret β_q

B.2 Attack 2: Obtaining the secret M

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include "paillier.h"
#define num_of_features 100
#define C 2
#define epsilon 2
#define eta (num_of_features + 1 + C + epsilon)

paillier_plaintext_t *query[num_of_features];
paillier_ciphertext_t *encrypted_query[num_of_features];
mpz_t M[eta][eta];
mpz_t calcM[eta][eta];
mpz_t Rq[C];
paillier_ciphertext_t *Aq[eta];
paillier_plaintext_t *decrypted_query[eta];
paillier_plaintext_t *zero, *plainPhi;
paillier_ciphertext_t *temp, *encryptedPhi;

mpz_t maxRange, phi, betaQ, gcd, BigInt;
// initilze the state object for the random generator functions
gmp_randstate_t state;

paillier_pubkey_t *PK;
paillier_prvkey_t *SK;
```

```
//-----------------------------------------------------------------
// QU's Work
void genKeysByQU()
{
    void *buf = malloc(128);
    void (*get_rand)(void *, int) = paillier_get_rand_devurandom;
    get_rand(buf, 128);
    paillier_keygen(1024, &PK, &SK, get_rand);
    free(buf);
}

void initialization()
{
    mpz_init(gcd);
    mpz_init(betaQ);
    mpz_init(phi);
    zero = (paillier_plaintext_t *)malloc(sizeof(
   paillier_plaintext_t));
    mpz_init_set_si(zero->m, 0);
    plainPhi = (paillier_plaintext_t *)malloc(sizeof(
   paillier_plaintext_t));
    mpz_init(plainPhi->m);
    temp = (paillier_ciphertext_t *)malloc(sizeof(
   paillier_ciphertext_t));
    mpz_init(temp->c);
    for (int i = 0; i < num_of_features; i++)
    {
        query[i] = (paillier_plaintext_t *)malloc(sizeof(
   paillier_plaintext_t));
        encrypted_query[i] = (paillier_ciphertext_t *)malloc(sizeof
   (paillier_ciphertext_t));
        mpz_init(query[i]->m);
        mpz_init(encrypted_query[i]->c);
    }
    for (int i = 0; i < eta; i++)
    {
        decrypted_query[i] = (paillier_plaintext_t *)malloc(sizeof(
   paillier_plaintext_t));
        Aq[i] = (paillier_ciphertext_t *)malloc(sizeof(
   paillier_ciphertext_t));
        mpz_init(Aq[i]->c);
        mpz_init(decrypted_query[i]->m);
    }
    for (int i = 0; i < eta; i++)
    {
        for (int j = 0; j < eta; j++)
        {
            mpz_init_set_si(calcM[i][j], 0);
        }
    }
    mpz_init_set_si(BigInt, 1);
    for (int i = 0; i < num_of_features; i++)
    {
        mpz_mul_si(BigInt, BigInt, 1000);
    }
}

void genQueryByQU(int index)
{
    for (int i = 0; i < num_of_features; i++)
    {
        mpz_set_si(query[i]->m, 0);
    }
    mpz_set(query[index]->m, BigInt);
}

void encryptQueryByQU()
{
    void *buf = malloc(128);
    void (*get_rand)(void *, int) = paillier_get_rand_devurandom;
    get_rand(buf, 128);
    for (int i = 0; i < num_of_features; i++)
    {
        paillier_enc(encrypted_query[i], PK, query[i], get_rand);
    }
    free(buf);
}
//----------------------------------------------------------------
// DO's Work
void read_M_by_DO()
{
    FILE *file;
    file = fopen("M.txt", "r");
    for (int i = 0; i < eta; i++)
    {
        for (int j = 0; j < eta; j++)
        {
            gmp_fscanf(file, "%Zd", M[i][j]);
        }
    }
    fclose(file);
}

void genBetaByDO()
{
    mpz_urandomm(betaQ, state, maxRange);
    gmp_printf("%s␣=␣%Zd\n", "Beta␣generated␣by␣DO", betaQ);
    // pi = TODO
    for (int i = 0; i < C; i++)
    {
        mpz_init(Rq[i]);
        mpz_urandomm(Rq[i], state, maxRange);
    }
}

void encryptQueryByDO()
{
    void *buf = malloc(128);
    void (*get_rand)(void *, int) = paillier_get_rand_devurandom;
    get_rand(buf, 128);

    for (int i = 0; i < eta; i++)
    {
        paillier_enc(Aq[i], PK, zero, get_rand);
        for (int j = 0; j < eta; j++)
        {
            int t = j;
            mpz_mul(phi, M[i][j], betaQ);
            if (t < num_of_features)
            {
                mpz_set(plainPhi->m, phi);
                paillier_exp(PK, temp, encrypted_query[t], plainPhi
);
                paillier_mul(PK, Aq[i], Aq[i], temp);
            }
            else if (t == num_of_features)
            {
                mpz_set(plainPhi->m, phi);
                encryptedPhi = paillier_enc(NULL, PK, plainPhi,
get_rand);
                paillier_mul(PK, Aq[i], Aq[i], encryptedPhi);
                paillier_freeciphertext(encryptedPhi);
            }
            else if (t <= num_of_features + C)
            {
                mpz_mul(phi, phi, Rq[t - num_of_features - 1]);
                mpz_set(plainPhi->m, phi);
                encryptedPhi = paillier_enc(NULL, PK, plainPhi,
get_rand);
                paillier_mul(PK, Aq[i], Aq[i], encryptedPhi);
                paillier_freeciphertext(encryptedPhi);
            }
        }
    }
    free(buf);
}

//---------------------------------------------------------------
// QU’s Work
void decryptQueryByQU(int index)
{
    for (int i = 0; i < eta; i++)
    {
        paillier_dec(decrypted_query[i], PK, SK, Aq[i]);
    }
    mpz_set(gcd, decrypted_query[0]->m);
    for (int i = 1; i < eta; i++)
    {
        mpz_gcd(gcd, gcd, decrypted_query[i]->m);
    }
    gmp_printf("%s␣=␣%Zd\n", "Calculated␣Beta␣at␣QU", gcd);
    for (int i = 0; i < eta; i++)
    {
        mpz_divexact(decrypted_query[i]->m, decrypted_query[i]->m,
   gcd);
    }
    for (int i = 0; i < eta; i++)
    {
        mpz_t a;
        mpz_init(a);
        mpz_mod(a, decrypted_query[i]->m, BigInt);
        mpz_sub(decrypted_query[i]->m, decrypted_query[i]->m, a);
        mpz_divexact(calcM[i][index], decrypted_query[i]->m, BigInt
   );
    }
}

void garbageCollection()
{
    // *********************** GARBAGE COLLECTION
   *********************** //
    // empty the memory location for the random generator state
    gmp_randclear(state);
    // clear the memory locations for the variables used to avoid
   leaks
    mpz_clear(maxRange);
    mpz_clear(phi);
    mpz_clear(betaQ);
    mpz_clear(gcd);
    mpz_clear(BigInt);

    paillier_freepubkey(PK);
    paillier_freeprvkey(SK);
    paillier_freeplaintext(zero);
    paillier_freeplaintext(plainPhi);
    paillier_freeciphertext(temp);

    for (int i = 0; i < num_of_features; i++)
    {
        paillier_freeplaintext(query[i]);
        paillier_freeciphertext(encrypted_query[i]);
    }
    for (int i = 0; i < eta; i++)
    {
        paillier_freeplaintext(decrypted_query[i]);
        paillier_freeciphertext(Aq[i]);
    }
    for (int i = 0; i < C; i++)
    {
        mpz_clear(Rq[i]);
    }
    for (int i = 0; i < eta; i++)
    {
        for (int j = 0; j < eta; j++)
        {
            mpz_clear(M[i][j]);
            mpz_clear(calcM[i][j]);
        }
    }
}

void attack()
{
    for (int i = 0; i < num_of_features; i++)
    {
        genQueryByQU(i);
        encryptQueryByQU();
        genBetaByDO();
        encryptQueryByDO();
        decryptQueryByQU(i);
    }
    FILE *file;
    file = fopen("calcMbyQU.txt", "w");
    for (int i = 0; i < eta; i++)
    {
        for (int j = 0; j < eta; j++)
        {
            gmp_fprintf(file, "%Zd␣", calcM[i][j]);
            // if (mpz_cmp(M[i][j], calcM[i][j]))
            //     gmp_printf("%d %d ", i, j);
        }
        // gmp_printf("\n");
        gmp_fprintf(file, "\n");
    }
    fclose(file);
}

int main()
{
    unsigned long seed;
    mpz_init_set_si(maxRange, 999999);
    // initialize state for a Mersenne Twister algorithm. This
   algorithm is fast and has good randomness properties.
    gmp_randinit_mt(state);
    // create the generator seed for the random engine to reference
    gmp_randseed_ui(state, seed);

    initialization();
    read_M_by_DO();
    genKeysByQU();
    attack();
    garbageCollection();

    return 0;
}
```

Code Snippet B.2: Obtaining the secret M

B.3 Attack 3: Obtaining the DataBase in clear

```
import numpy as np
from numpy.linalg import inv
num_of_features = 100
c = 2
epsilon = 2
eta = num_of_features + 1 + c + epsilon

# Calculating Encrypted DB Tuple
file = open('ModifiedDataBase.txt')
modified_DB = file.readlines()
modified_tuple1 = [float(x) for x in modified_DB[0].split("␣")
   [:-1]]
modified_tuple1 = np.array(modified_tuple1)
actual_M = np.loadtxt("M.txt", dtype='int')
M_inverse = inv(actual_M)
encrypted_tuple1 = np.dot(modified_tuple1, M_inverse)
```

```
file.close()

# Getting PlainText Query
file = open('DataBase.txt')
DataBase = file.readlines()
plain_tuple1 = [float(x) for x in DataBase[0].split("␣")[:-1]]
for i in range(eta - num_of_features):
    plain_tuple1.append(0.0)
plain_tuple1 = np.array(plain_tuple1)
file.close()

# Performing Attack to get s_vector

plain_tuple1 = np.dot(2, plain_tuple1)
calc_M = np.loadtxt("calcMbyQU.txt", dtype='int')
temp = encrypted_tuple1.dot(calc_M)
s_vector = np.add(plain_tuple1, temp)
np.savetxt('Calculated_s_vector_by_QU.txt', s_vector)

# Performing Attack to get DataBase in clear
modified_DB = np.loadtxt('ModifiedDataBase.txt', dtype='float')
encrypted_DB = np.matmul(modified_DB, M_inverse)

temp = np.matmul(encrypted_DB, calc_M)
temp = temp - s_vector
calc_DB_by_QU = np.divide(temp, -2)
np.savetxt('Calculated_DataBase_by_QU.txt', calc_DB_by_QU)
```

Code Snippet B.3: Obtaining the DataBase in clear

B.4 Attack 4: Obtaining the query in clear

```
from operator import matmul
import numpy as np
from numpy.linalg import inv

```

```
num_of_features = 100

M = np.loadtxt('M.txt', dtype='float')
inv_M = inv(M)

# Assumption: QU have (num_of_features + 1) plaintext [i.e., A and
    B]
plainDB = np.loadtxt('DataBase.txt', dtype='float')
A = plainDB[0:num_of_features, 0:num_of_features]
B = plainDB[1:num_of_features+1, 0:num_of_features]
C = 2 * np.subtract(A, B)
inv_C = inv(C)

D = np.linalg.norm(A, axis=1)
E = np.linalg.norm(B, axis=1)

F = np.square(D) - np.square(E)

# Assumption: QU have (num_of_features + 1) ciphertext [i.e.,
    encrypted_DB]
DB_cap = np.loadtxt('ModifiedDataBase.txt', dtype='float')
DB_cap = DB_cap[0:num_of_features + 1, 0:]
encrypted_DB = matmul(DB_cap, inv_M)

encrypted_query = np.loadtxt('finalDecryptedQuery.txt', dtype='int'
    )

betaQ = np.gcd.reduce(encrypted_query)

G = np.dot(encrypted_DB, np.transpose(encrypted_query))
G = np.divide(G, betaQ)
H = G[0:num_of_features]
I = G[1:num_of_features + 1]
J = np.subtract(H, I)

K = np.subtract(F, J)
```

```
calc_query = np.dot(inv_C, K)
print(calc_query)
```

Code Snippet B.4: Obtaining the query in clear

Appendix C

Our Design

C.1 kNN code inside enclave E1

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <time.h>
#define num_of_data_points 10000
#define num_of_features 1000
#define c 2
#define epsilon 2
#define eta (num_of_features + 1 + c + epsilon)
int k = 10;

typedef struct
{
    int index;
    mpf_t distance;
} Node;

mpf_t Encrypted_DB[num_of_data_points][eta];
mpf_t Encrypted_Query[eta];

mpf_t dist;
mpz_t maxRange;
// initilze the state object for the random generator functions
```

```
gmp_randstate_t state;

Node node[num_of_data_points];

void readEncryptedDB()
{
    FILE *file;
    file = fopen("EncryptedDataBase.txt", "r");
    for (int i = 0; i < num_of_data_points; i++)
    {
        for (int j = 0; j < eta; j++)
        {
            mpf_init(Encrypted_DB[i][j]);
            gmp_fscanf(file, "%Ff", Encrypted_DB[i][j]);
        }
    }
    fclose(file);
}

void readEncryptedQuery()
{
    FILE *file;
    file = fopen("finalDecryptedQuery.txt", "r");
    for (int i = 0; i < eta; i++)
    {
        mpf_init(Encrypted_Query[i]);
        gmp_fscanf(file, "%Ff", Encrypted_Query[i]);
    }
    fclose(file);
}

void calc_distance()
{
    mpf_t temp;
    mpf_init(temp);
    for (int i = 0; i < num_of_data_points; i++)
    {
        node[i].index = i;
        mpf_init_set_si(node[i].distance, 0.0);
        for (int j = 0; j < eta; j++)
        {
            mpf_mul(temp, Encrypted_DB[i][j], Encrypted_Query[j]);
            mpf_add(node[i].distance, node[i].distance, temp);
        }
    }
    mpf_clear(temp);
}

// Function to swap the the position of two elements
void swap(Node *a, Node *b)
{
    int data = a->index;
    a->index = b->index;
    b->index = data;
    mpf_set(dist, a->distance);
    mpf_set(a->distance, b->distance);
    mpf_set(b->distance, dist);
}

void heapify(int n, int i)
{
    // Find smallest among root, left child and right child
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && mpf_cmp(node[left].distance, node[smallest].
   distance) < 0)
        smallest = left;

    if (right < n && mpf_cmp(node[right].distance, node[smallest].
   distance) < 0)
```

```
        smallest = right;

    // Swap and continue heapifying if root is not smallest
    if (smallest != i)
    {
        swap(&node[i], &node[smallest]);
        heapify(n, smallest);
    }
}

// Main function to do heap sort
void heapSort()
{
    // Build min heap
    for (int i = num_of_data_points / 2 - 1; i >= 0; i--)
        heapify(num_of_data_points, i);

    // Heap sort
    int count = 0;
    for (int i = num_of_data_points - 1; count < k && i >= 0; i--)
    {
        swap(&node[0], &node[i]);

        // Heapify root element to get smallest element at root
   again
        heapify(i, 0);
        count++;
    }
}

void returnTopK()
{
    FILE *file;
    file = fopen("TopK.txt", "w");
    for (int i = 0; i < k; i++)
    {
        gmp_fprintf(file, "%d\n", node[num_of_data_points - i - 1].
    index);
        gmp_fprintf(file, "%Ff␣%d\n", node[num_of_data_points - i -
     1].distance, node[num_of_data_points - i - 1].index);
    }
    fclose(file);
}

void garbageCollection()
{
    // *********************** GARBAGE COLLECTION
    *********************** //
    // empty the memory location for the random generator state
    gmp_randclear(state);
    // clear the memory locations for the variables used to avoid
    leaks
    mpz_clear(maxRange);
    for (int i = 0; i < num_of_data_points; i++)
    {
        for (int j = 0; j < eta; j++)
        {
            mpf_clear(Encrypted_DB[i][j]);
        }
    }
    for (int i = 0; i < eta; i++)
    {
        mpf_clear(Encrypted_Query[i]);
    }
    for (int i = 0; i < num_of_data_points; i++)
    {
        mpf_clear(node[i].distance);
    }
    mpf_clear(dist);
}
```

```
// Reference: https://stackoverflow.com/questions/30942413/c-gmp-
   generating-random-number
int main()
{
    unsigned long seed;
    mpz_init_set_si(maxRange, 99);
    mpf_init(dist);

    // initialize state for a Mersenne Twister algorithm. This
   algorithm is fast and has good randomness properties.
    gmp_randinit_mt(state);
    // create the generator seed for the random engine to reference
    gmp_randseed_ui(state, seed);

    readEncryptedDB();
    readEncryptedQuery();
    clock_t begin = clock();
    calc_distance();
    heapSort();
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("%f\n", time_spent);
    returnTopK();
    garbageCollection();

    return 0;
}
```

Code Snippet C.1: kNN code inside enclave E1

C.2 Untrusted App code

```

#include <stdio.h>
#include <fstream>
#include <iostream>
```

```
#include <string.h>
#include <stdarg.h>
#include <assert.h>
#include <stdlib.h>

#include <unistd.h>
#include <pwd.h>
#define MAX_PATH FILENAME_MAX

#include "sgx_urts.h"
#include "App.h"
#include "Enclave_u.h"

using namespace std;

/* Global EID shared by multiple threads */
sgx_enclave_id_t global_eid = 0;

typedef struct _sgx_errlist_t
{
    sgx_status_t err;
    const char *msg;
    const char *sug; /* Suggestion */
} sgx_errlist_t;

/* Error code returned by sgx_create_enclave */
static sgx_errlist_t sgx_errlist[] = {
    {SGX_ERROR_UNEXPECTED,
     "Unexpected␣error␣occurred.",
     NULL},
    {SGX_ERROR_INVALID_PARAMETER,
     "Invalid␣parameter.",
     NULL},
    {SGX_ERROR_OUT_OF_MEMORY,
     "Out␣of␣memory.",
     NULL},
    {SGX_ERROR_ENCLAVE_LOST,
     "Power␣transition␣occurred.",
     "Please␣refer␣to␣the␣sample␣\"PowerTransition\"␣for␣details."
   },
    {SGX_ERROR_INVALID_ENCLAVE,
     "Invalid␣enclave␣image.",
     NULL},
    {SGX_ERROR_INVALID_ENCLAVE_ID,
     "Invalid␣enclave␣identification.",
     NULL},
    {SGX_ERROR_INVALID_SIGNATURE,
     "Invalid␣enclave␣signature.",
     NULL},
    {SGX_ERROR_OUT_OF_EPC,
     "Out␣of␣EPC␣memory.",
     NULL},
    {SGX_ERROR_NO_DEVICE,
     "Invalid␣SGX␣device.",
     "Please␣make␣sure␣SGX␣module␣is␣enabled␣in␣the␣BIOS,␣and␣
   install␣SGX␣driver␣afterwards."},
    {SGX_ERROR_MEMORY_MAP_CONFLICT,
     "Memory␣map␣conflicted.",
     NULL},
    {SGX_ERROR_INVALID_METADATA,
     "Invalid␣enclave␣metadata.",
     NULL},
    {SGX_ERROR_DEVICE_BUSY,
     "SGX␣device␣was␣busy.",
     NULL},
    {SGX_ERROR_INVALID_VERSION,
     "Enclave␣version␣was␣invalid.",
     NULL},
    {SGX_ERROR_INVALID_ATTRIBUTE,
     "Enclave␣was␣not␣authorized.",
     NULL},
    {SGX_ERROR_ENCLAVE_FILE_ACCESS,
     "Can’t␣open␣enclave␣file.",
     NULL},
};

/* Check error conditions for loading enclave */
void print_error_message(sgx_status_t ret)
{
    size_t idx = 0;
    size_t ttl = sizeof sgx_errlist / sizeof sgx_errlist[0];

    for (idx = 0; idx < ttl; idx++)
    {
        if (ret == sgx_errlist[idx].err)
        {
            if (NULL != sgx_errlist[idx].sug)
                printf("Info:␣%s\n", sgx_errlist[idx].sug);
            printf("Error:␣%s\n", sgx_errlist[idx].msg);
            break;
        }
    }

    if (idx == ttl)
        printf("Error␣code␣is␣0x%X.␣Please␣refer␣to␣the␣\"Intel␣SGX
   ␣SDK␣Developer␣Reference\"␣for␣more␣details.\n", ret);
}

/* Initialize the enclave:
 *   Call sgx_create_enclave to initialize an enclave instance
 */
int initialize_enclave(void)
{
    sgx_status_t ret = SGX_ERROR_UNEXPECTED;

    /* Call sgx_create_enclave to initialize an enclave instance */
    /* Debug Support: set 2nd parameter to 1 */
    ret = sgx_create_enclave(ENCLAVE_FILENAME, SGX_DEBUG_FLAG, NULL
   , NULL, &global_eid, NULL);
    if (ret != SGX_SUCCESS)
    {
        print_error_message(ret);
        return -1;
    }

    return 0;
}

/* OCall functions */
void ocall_print_string(const char *str)
{
    /* Proxy/Bridge will check the length and null-terminate
     * the input string to prevent buffer overflow.
     */
    printf("%s", str);
}

#define num_of_data_points 10000
#define num_of_features 1000
#define eta (num_of_features + 1)
int k = 10;

double Encrypted_Query[eta];

#define DB_size (num_of_data_points * eta)

double Encrypted_DB[DB_size];

void readEncryptedDB()
{
    ifstream fin("EncryptedDataBase.txt");
    for (int i = 0; i < DB_size; i++)
    {
        fin >> Encrypted_DB[i];
    }
    fin.close();
}

void readEncryptedQuery()
{
    ifstream fin("finalDecryptedQuery.txt");
    for (int i = 0; i < eta; i++)
    {
        fin >> Encrypted_Query[i];
    }
    fin.close();
}

/* Application entry */
int SGX_CDECL main(int argc, char *argv[])
{
    (void)(argc);
    (void)(argv);

    /* Initialize the enclave */
    if (initialize_enclave() < 0)
    {
        printf("Enter␣a␣character␣before␣exit␣...\n");
        getchar();
        return -1;
    }

    readEncryptedDB();
    readEncryptedQuery();

    int *result = (int *)malloc(k * sizeof(int));

    clock_t begin = clock();
    ecall_computeKNN(global_eid, result, Encrypted_DB,
   Encrypted_Query, DB_size, eta, k);
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Encryption␣Time␣=␣%f\n", time_spent);

    ofstream fout("TopK.txt");
    for (int i = 0; i < k; i++)
    {
        fout << result[i] << "\n";
        printf("%d\n", result[i]);
    }
    fout.close();

    /* Utilize edger8r attributes */
    edger8r_array_attributes();
    edger8r_pointer_attributes();
    edger8r_type_attributes();
    edger8r_function_attributes();

    /* Utilize trusted libraries */
    ecall_libc_functions();
    ecall_libcxx_functions();
    ecall_thread_functions();

    /* Destroy the enclave */
    sgx_destroy_enclave(global_eid);

    printf("Info:␣SampleEnclave␣successfully␣returned.\n");

    printf("Enter␣a␣character␣before␣exit␣...\n");
    getchar();
    return 0;
}
```

Code Snippet C.2: Untrusted App code

C.3 .EDL file defining ecalls and ocalls

```
/* Enclave.edl - Top EDL file. */

enclave {

    include "user_types.h" /* buffer_t */

    /* Import ECALL/OCALL from sub-directory EDLs.
     *  [from]: specifies the location of EDL file.
     *  [import]: specifies the functions to import,
     *  [*]: implies to import all functions.
     */

    from "Edger8rSyntax/Types.edl" import *;
    from "Edger8rSyntax/Pointers.edl" import *;
    from "Edger8rSyntax/Arrays.edl" import *;
    from "Edger8rSyntax/Functions.edl" import *;

    from "TrustedLibrary/Libc.edl" import *;
    from "TrustedLibrary/Libcxx.edl" import ecall_exception,
   ecall_map;
    from "TrustedLibrary/Thread.edl" import *;

    trusted{
        public void ecall_computeKNN([in, out, count = k]int *
   result, [in, count = DB_size]double *A, [in, count = n]double *B
   , size_t DB_size, size_t n, size_t k);
    };

    /*
     * ocall_print_string - invokes OCALL to display string buffer
   inside the enclave.
     *  [in]: copy the string buffer to App outside.
     *  [string]: specifies 'str' is a NULL terminated buffer.
     */
    untrusted {
        void ocall_print_string([in, string] const char *str);
    };

};
```

Code Snippet C.3: .EDL file defining ecalls and ocalls

References

Baumann, A., Peinado, M., and Hunt, G., 2015 Aug., “Shielding Applications from an Untrusted Cloud with Haven,” *ACM Trans. Comput. Syst.* **33**

Cao, N., Wang, C., Li, M., Ren, K., and Lou, W., 2014, “Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data,” *IEEE Transactions on Parallel and Distributed Systems* **25**, 222–233.

Costan, V., and Devadas, S., 2016, “Intel SGX Explained,” Cryptology ePrint Archive, Paper 2016/086, `https://eprint.iacr.org/2016/086`.

insujang, 2017, “Intel sgx protection mechanism,”

Intel, 2015, “Intel® software guard extensions(intel® sgx),”

Li, J., Ma, J., Miao, Y., Ruikang, Y., Liu, X., and Choo, K.-K. R., 2020, “Practical Multi-keyword Ranked Search with Access Control over Encrypted Cloud Data,” *IEEE Transactions on Cloud Computing*, 1–1.

McKeen, F., 2015, “Intel® software guard extensions(intel® sgx),”

Paillier, P., 1999, “Public-key cryptosystems based on composite degree residuosity classes,” in *Advances in Cryptology — EUROCRYPT ’99*, edited by Stern, J. (Springer Berlin Heidelberg, Berlin, Heidelberg). pp. 223–238.

Parampalli, U., Wu, W., Liu, J., and Xian, M., 2019, “Privacy preserving k-nearest neighbor classification over encrypted database in outsourced cloud environments,” *World Wide Web* **22**, 101–123.

Rebeiro, C., 2018, “Trusted execution environments,”

SGX101, 2019, “Sgx 101,”

Singh, G., Kaul, A., and Mehta, S., 2018, “Secure k-NN as a Service over Encrypted Data in Multi-User Setting,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 154–161.

Zhu, Y., Huang, Z., and Takagi, T., 2016, “Secure and controllable k-NN query over encrypted cloud data with key confidentiality,” *Journal of Parallel and Distributed Computing* **89**, 1–12.

Acknowledgements

I would like to thank Professor Bernard Menezes, CSE Dept, IIT Bombay, for his guidance throughout the M. Tech Project.

Aditya Jain
IIT Bombay
29 June 2022