Configuration Manual

MSc Research Project
MSc in Cloud Computing

Prajwal Seethur Raveendra
Student ID: 22228811

School of Computing
National College of Ireland

Supervisor: Aqeel Kazmi

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Prajwal Seethur Raveendra		
Student ID:	22228811		
Programme:	MSc in Cloud computing	**Year:**	2024
Module:	MSc Research Project		
Lecturer:	Aqeel Kazmi		
Submission Due Date:	12/08/2024		
Project Title:	Effective Optimization strategies to utilize Fully Homomorphic Encryption in Cloud Platforms		

Word Count:1142 **Page Count:** 9

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Prajwal Seethur Raveendra

Date: 12/08/2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Prajwal Seethur Raveendra
Student ID: 22228811

1 Introduction

Fully Homomorphic encryption is a type of encryption mechanism that allows computation on encrypted data without the need for decryption. FHE can be quite useful in cloud environments because it can be used to perform secure calculations. Businesses that prioritize data privacy can leverage this technology. FHE is impractical because of its high computational demands and researchers are actively trying to come up with new strategies to optimize its performance. In this research, a novel framework was developed using data splitting and parallel processing. Two lambda functions were created, one lambda function to handle homomorphic calculations and the other to handle normal function. AWS API Gateway was used to invoke the lambda functions from the client end. The results show a drastic 94% CPU time reduction when compared to other FHE schemes and a significant 82.88% reduction.

2 Environment Set-up

1. Most FHE libraries are compatible with Linux Operating System. So, the first step will be to set up a Linux environment. Implementation and the experiments (case studies) were performed on WSL (Windows Subsystem for Linux).
2. Create a python environment (Preferably python version 3.10).
3. Update the Linux OS using the command sudo apt-get update
4. Install all the necessary libraries

```
import pandas as pd
import tenseal as ts
import time
import numpy as np
import psutil
from memory_profiler import memory_usage
import gc
import json
from Pyfhel import Pyfhel
```

These are all the libraries used throughout the research (including case study codes).

```
sensitive_columns = ['age', 'blood_pressure', 'cholesterol']
    operation = 'average'  # Choose the operation: 'average', 'addition', or 'multiplication'
```

In the sensitive columns are specified in the sensitive_columns list and the type of operation to be performed has to be specified as well. The framework only supports addition, multiplication and average(mean) operations. More meaningful operations can be integrated into the framework.

3 AWS Lambda function

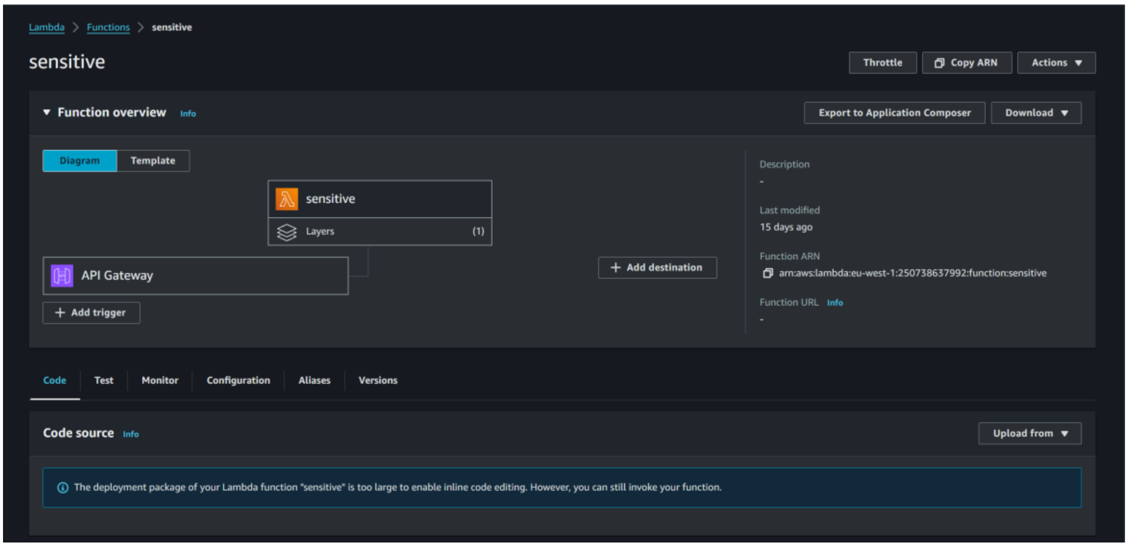


Figure 1: Sensitive lambda handler dashboard

The sensitive lambda code was uploaded as a zip file. The sensitive lambda handler is running on a python 3.10 version Linux environment. AWS lambda environment supports all of the basic python libraries. The zip file contains the python code that handles homomorphic calculations and all the dependency files. To install any third-party library into a specific file directory, pip install tenseal -t . should be used. As you can see from the figure that an extra layer is added to the lambda function. The layer's main purpose is only to install the pandas python library.

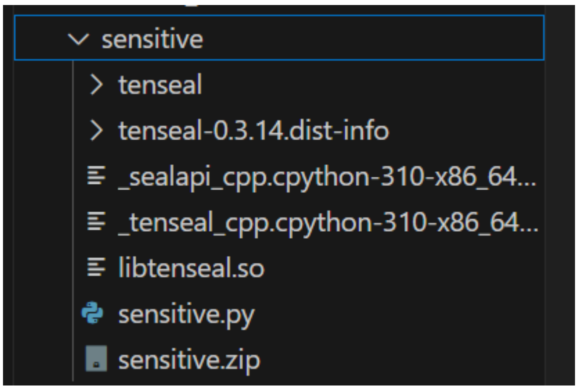


Figure 2: Sensitive.zip files

The tenseal library is a homomorphic encryption library that supports CKKS and BFV Schemes (Benaissa, Retiat, Cebere, & Belfedhal, 2021).

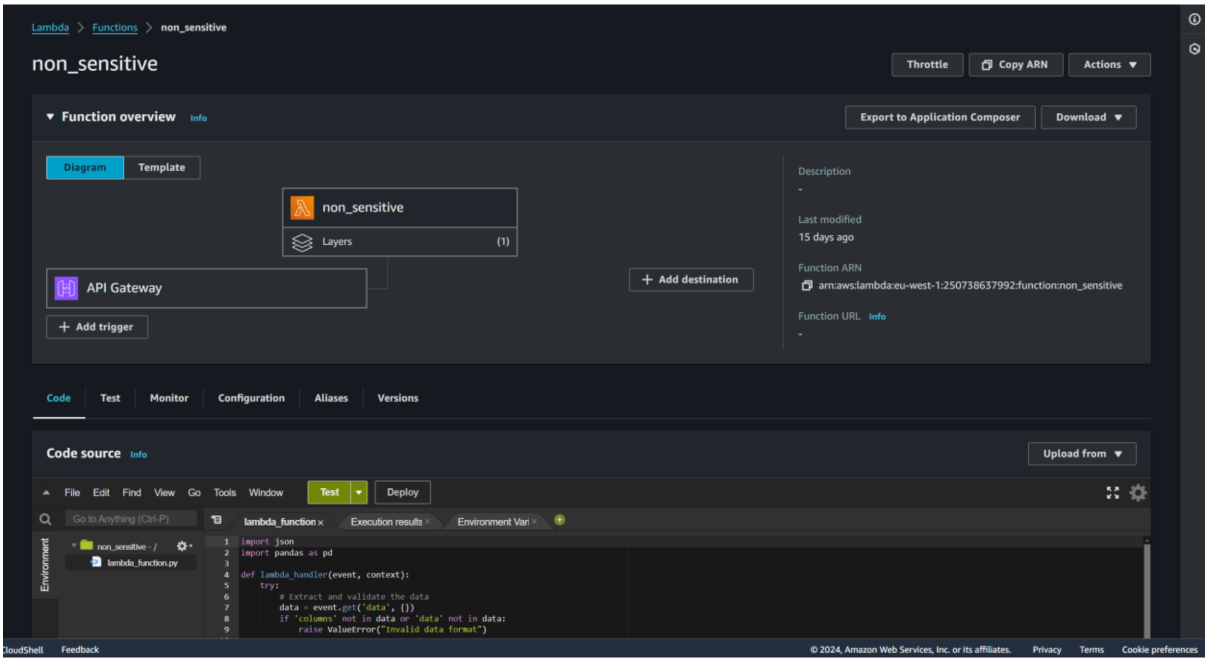
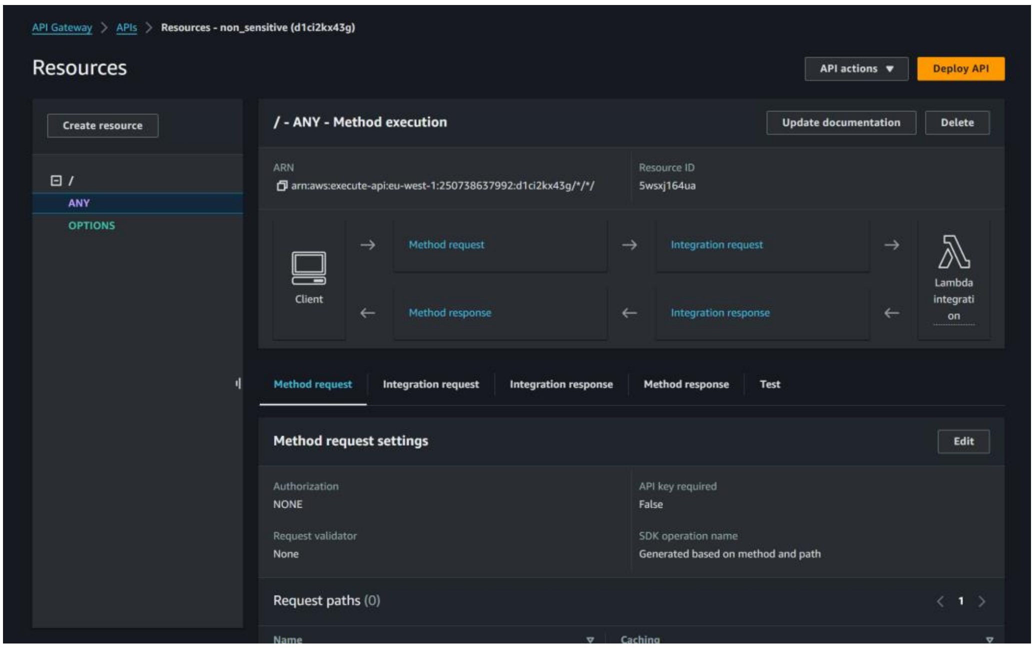


Figure 3: Non sensitive lambda handler

The non sensitive data lambda handler code was directly written on the lambda text editor since this code does not require any third-part library. The padasSDK layer was to this lambda function as well. This lambda function is running on the latest python 3.12 version Linux environment.

4 AWS API Gateway



AWS API Gateway is integrated into the lambda functions so that it can invoked from the client script. CORS (Cross Origin Resource Sharing) is enabled to allow requests from the client script.

```
data = prepare_data_for_lambda(df, sensitive_columns, operation)
    non_sensitive_url = 'https://d1ci2kx43g.execute-api.eu-west-
1.amazonaws.com/non_sensitive/'
    sensitive_url = 'https://1gjf7bj3f0.execute-api.eu-west-
1.amazonaws.com/sensitive/'
```

The API endpoints are entered in the framework client script.

5 Case studies conducted

This section contains all the important code snippets used in all the case study scripts.

5.1 FHE Schemes codes

The evaluation was performed by writing code for all FHE Schemes to do the comparisons. The whole dataset was encrypted for other FHE Schemes while only the sensitive data is encrypted in the framework. A separate program was written for the proposed FHE framework without using AWS lambda. The case studies were performed using a synthetic a dataset.

```
# Sample healthcare dataset
def generate_large_dataset(num_rows):
    data = {
        'patient_id': range(1, num_rows + 1),
        'age': np.random.randint(18, 80, size=num_rows),
        'blood_pressure': np.random.randint(90, 150, size=num_rows),
        'cholesterol': np.random.randint(150, 300, size=num_rows),
        'height': np.random.randint(150, 200, size=num_rows),
        'weight': np.random.randint(50, 100, size=num_rows),
        'heart_rate': np.random.randint(60, 100, size=num_rows),
        'temperature': np.random.uniform(36.0, 37.5, size=num_rows)
    }
    return pd.DataFrame(data)
```

Figure 4: Generates synthetic dataset

```python
# Setup TenSEAL context
def setup_tenseal():
    context = ts.context(
        ts.SCHEME_TYPE.CKKS,
        poly_modulus_degree=8192,
        coeff_mod_bit_sizes=[60, 40, 40, 60]
    )
    context.generate_galois_keys()
    context.global_scale = 2**40
    return context

# Encryption and decryption functions
def encrypt_data(data, context):
    return ts.ckks_vector(context, [data])

def decrypt_data(encrypted_data):
    return encrypted_data.decrypt()[0]

# Process data function
def process_data(df, operation):
    context = setup_tenseal()
    results = df.copy()

    for column in df.columns:
        if df[column].dtype in ['int64', 'float64']:
            for i in range(len(df)):
                encrypted_value = encrypt_data(float(df[column][i]), context)
                if operation == 'addition':
                    processed_value = encrypted_value + encrypted_value  # Example: doubling the value
                elif operation == 'multiplication':
                    processed_value = encrypted_value * 2  # Example: doubling the value
                results[column][i] = decrypt_data(processed_value)

    return results
```

Figure 5: CKKS Scheme context setup, encryption and decryption function functions and encrypted calculations.

```python
# Setup Pyfhel context
def setup_pyfhel():
    HE = Pyfhel()           # Creating empty Pyfhel object

    bgv_params = {
        'scheme': 'BGV',    # can also be 'bgv'
        'n': 2**13,         # Polynomial modulus degree, the num. of slots per plaintext
        't': 65537,         # Plaintext modulus. Encrypted operations happen modulo t
        't_bits': 20,       # Number of bits in t. Used to generate a suitable value for t
        'sec': 128,         # Security parameter. The equivalent length of AES key in bits
    }
    HE.contextGen(**bgv_params)  # Generate context for BGV scheme
    HE.keyGen()                  # Key Generation: generates a pair of public/secret keys
    HE.rotateKeyGen()            # Rotate key generation --> Allows rotation/shifting
    HE.relinKeyGen()             # Relinearization key generation
    return HE
```

Figure 6: BGV Scheme context setup

The Tenseal library only supports the CKKS scheme and the BFV Scheme. The BGV scheme was coded using the python library Pyfhel (Ibarrondo & Viand, 2021). It supports all the basic Homomorphic operations like the tenseal library.

```python
# Setup TenSEAL context
def setup_tenseal():
    context = ts.context(
        ts.SCHEME_TYPE.BFV,
        poly_modulus_degree=8192,
        plain_modulus=1032193
    )
    context.generate_galois_keys()
    return context
```

Figure 7: BFV Context setup

Remaining sections of the code is pretty much the same except for the context setup part.

5.2 Case study framework code

```python
def process_sensitive_data_worker(df_chunk, sensitive_columns, operation):
    context = setup_tenseal()
    results = df_chunk.copy()

    for column in sensitive_columns:
        if df_chunk[column].dtype in ['int64', 'float64']:
            for i in range(len(df_chunk)):
                encrypted_value = encrypt_data(df_chunk[column].iloc[i], context)
                if operation == 'addition':
                    processed_value = encrypted_value + encrypted_value  # Example: doubling the value
                elif operation == 'multiplication':
                    processed_value = encrypted_value * 2  # Example: doubling the value
                results.at[df_chunk.index[i], column] = decrypt_data(processed_value)

    return results

# Worker function for parallel processing of non-sensitive columns
def process_non_sensitive_data_worker(df_chunk, non_sensitive_columns, operation):
    results = df_chunk.copy()

    for column in non_sensitive_columns:
        if df_chunk[column].dtype in ['int64', 'float64']:
            if operation == 'addition':
                results[column] = df_chunk[column] + df_chunk[column]  # Example: doubling the value
            elif operation == 'multiplication':
                results[column] = df_chunk[column] * 2  # Example: doubling the value

    return results

# Main processing function
def process_data_parallel(df, sensitive_columns, operation):
    non_sensitive_columns = [col for col in df.columns if col not in sensitive_columns]

    num_chunks = multiprocessing.cpu_count()  # Number of chunks based on CPU cores
    chunk_size = len(df) // num_chunks
    chunks = [df.iloc[i:i + chunk_size] for i in range(0, len(df), chunk_size)]

    with multiprocessing.Pool(processes=num_chunks) as pool:
        sensitive_results = pool.starmap(process_sensitive_data_worker, [(chunk[sensitive_columns], sensitive_columns, operation) for chunk in chunks])
        non_sensitive_results = pool.starmap(process_non_sensitive_data_worker, [(chunk[non_sensitive_columns], non_sensitive_columns, operation) for chunk in ch

    sensitive_df = pd.concat(sensitive_results, ignore_index=True)
    non_sensitive_df = pd.concat(non_sensitive_results, ignore_index=True)

    processed_df = pd.concat([non_sensitive_df, sensitive_df], axis=1).sort_index(axis=1)

    return processed_df
```

Figure 8: Framework main functions

The framework case study code utilizes the CKKS scheme. The data is split based on sensitivity. The sensitive data worker handles the sensitive data calculations while the non-sensitive data worker handles non sensitive data calculations. These operations are processed parallelly in the process_data_parallel function using the python library called multiprocessing.

6 Framework development

```python
def lambda_handler(event, context):
    try:
        # Extract and validate data
        data = event.get('data', {})
        if 'columns' not in data or 'data' not in data:
            raise ValueError("Invalid data format")

        columns = data['columns']
        rows = data['data']

        # Validate that all rows have the same length
        if not all(len(row) == len(columns) for row in rows):
            raise ValueError("Mismatch between number of columns and data length")

        # Create DataFrame
        df = pd.DataFrame(rows, columns=columns)

        # Extract operation and sensitive columns
        operation = event.get('operation', '')
        sensitive_columns = event.get('sensitive_columns', [])

        # Process sensitive data
        result_df = process_sensitive_data(df, sensitive_columns, operation)

        return {
            'statusCode': 200,
            'body': json.dumps({'data': result_df.to_dict(orient='split')})
        }
    except Exception as e:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': str(e)})
        }
```

Figure 9: Sensitive lambda handler

```python
def lambda_handler(event, context):
    try:
        # Extract and validate the data
        data = event.get('data', {})
        if 'columns' not in data or 'data' not in data:
            raise ValueError("Invalid data format")

        columns = data['columns']
        rows = data['data']

        # Validate that all rows have the same length
        if not all(len(row) == len(columns) for row in rows):
            raise ValueError("Mismatch between number of columns and data length")

        # Create DataFrame
        df = pd.DataFrame(rows, columns=columns)

        # Perform operations
        operation = event.get('operation', '')
        if operation == 'average':
            result = df.mean().to_dict()
        elif operation == 'addition':
            result = df.sum().to_dict()
        elif operation == 'multiplication':
            result = df * 2
        else:
            result = {"error": "Invalid operation"}

        return {
            'statusCode': 200,
            'body': json.dumps({'data': result})
        }
    except Exception as e:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': str(e)})
        }
```

Figure 10: Non-sensitive lambda handler

7 Framework Output

```
Processed Data (first 5 rows):
   patient_id   age  blood_pressure  cholesterol  height  weight  heart_rate  temperature
0         3.0  42.0           127.0        216.0   170.0    75.0        76.6        36.64
1         NaN   NaN             NaN          NaN     NaN     NaN         NaN          NaN
2         NaN   NaN             NaN          NaN     NaN     NaN         NaN          NaN
3         NaN   NaN             NaN          NaN     NaN     NaN         NaN          NaN
4         NaN   NaN             NaN          NaN     NaN     NaN         NaN          NaN
prajwal@Prajwal:~$
```

After processing the data in their individual lambda functions, the processed output is sent back to the client. The data chunks are rejoined and displayed.

References

Benaissa, A., Retiat, B., Cebere, B., & Belfedhal, A. E. (2021). Tenseal: A library for encrypted tensor operations using homomorphic encryption. *arXiv preprint arXiv:2104.03152*.

Ibarrondo, A., & Viand, A. (2021, November). Pyfhel: Python for homomorphic encryption libraries. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (pp. 11-16).