Privacy-Preserving Fraud Detection Using Homomorphic Encryption (Project 2.8.2)

Malani Snowden
Applied Cryptography, 05/9/2025
Professor DiCrescenzo

PROBLEM STATEMENT

USE CASE

ALICE WANTS TO CLASSIFY HER TRANSACTION DATA WITHOUT REVEALING IT TO CAROL.

CORE QUESTION

How can Alice obtain a useful ML prediction without revealing her input data?

PROJECT OBJECTIVE

Implement a homomorphic encryption-based system that enable secure inference of a fraud detection model on encrypted input data. Must ensure to:

1. Preserve input privacy
2. Maintain classifier utility
3. Demonstrate secure end-to-end encrypted inference

WORKFLOW

Step 1: Preparing the data (alice.py)

1: Alice transaction data stored in ‘alice_data.json’

2: Alice loads transactions from file

3: Alice generates her keys and stores them appropriately

```json
{
    "Name": "Alice",
    "Account_Number": "123456789",
    "Transactions": [1200.0, -400.0, 350.0, -150.0, -200.0, 300.0]
}
```

```python
# Load ALice's transaction history
with open("alice_data.json", 'r') as file:
    data = json.load(file)
transactions = np.array(data["Transactions"], dtype=np.float32)
print(f"[ALICE] Transaction vector: {transactions}")
```

```python
# Create encryption context --> Generate context and keys
context = ts.context(
    ts.SCHEME_TYPE.CKKS, # use CKKS since it uses real numbers for calculations
    poly_modulus_degree=8192,
    coeff_mod_bit_sizes=[60, 40, 40, 60]
)
context.generate_galois_keys() # generate context keys
context.global_scale = 2 ** 40 # set gloabl scale

# Save keys
utils.write_data("keys/secret.txt", context.serialize(save_secret_key=True)) #
extract and store secret key
utils.write_data("carol_function/keys/public.txt", context.serialize()) # store
public key
```

Step 2: Encrypting the data and sending it to Carol (alice.py)

4: Alice encrypts her transactions

5: Then, she sends it to Carol through function '**upload_to_gcs()**'

>>> This concludes Alice's part for now

```python
# Encrypt and save transaction vector
enc_txn_vector = ts.ckks_vector(context, transactions)
utils.write_data("inputs/encrypted_transactions.txt", enc_txn_vector.serialize
())
```

```python
# Upload to Carol
upload_to_gcs(BUCKET_NAME, "inputs/encrypted_transactions.txt", "inputs/
encrypted_transactions.txt")
upload_to_gcs("alice_data", "carol_function/keys/public.txt", "keys/public.txt")

print(f"[ALICE] Encrypted data and key uploaded. Waiting for result...")
```

Step 3: Carol receives encrypted data, performs an inference (carol_listener.py)

6: Carol downloads the data she receives from Alice and reads it in.

7: Carol then initializes the weights and biases her neural network

```
print("[CAROL] Downloading public key and encrypted transactions from
bucket...")
download_blob(BUCKET_NAME, PUBLIC_KEY_BLOB, LOCAL_KEY_FILE)
download_blob(BUCKET_NAME, INPUT_BLOB, LOCAL_INPUT)

print("[CAROL] Loading public context...")
context = ts.context_from(utils.read_data(LOCAL_KEY_FILE))

# Load encrypted input (transaction vector)
txn_proto = utils.read_data(LOCAL_INPUT)
enc_txn = ts.ckks_vector_from(context, txn_proto)
```

```
# Pretrained nerual network (6 inputs (txn) -> 3 hidden -> 1 output)
# Weights for hidden layer (3 neurons)
print("[DEBUG] Reading encrypted transaction vector...")
w1 = [ # smaller weights and biases to prevent score inflation due to
encrypted math
    [0.01, 0.02, -0.03, 0.05, 0.01, -0.02],
    [-0.04, 0.03, 0.01, 0.01, 0.02, -0.01],
    [0.02, -0.02, 0.05, -0.03, 0.04, 0.01]
]
b1 = [0.01, -0.01, 0.005]

# Encode W1 rows and perform dot product for each neuron
print("[DEBUG] Computing hidden layer outputs...")
hidden_layer_outputs = []
for i in range(3):
    z = enc_txn.dot(w1[i]) # homomorphic dot product
    # a = z * z # must avoid squared activation so as to not increase scale
    hidden_layer_outputs.append(z)

# Output layer (1 neuron)
w2 = [0.1, -0.1, 0.15]
b2 = [0.05]
```

Step 3: Carol receives encrypted data, performs an inference (carol_listener.py)

8: Carol calculates a final encrypted fraud score and sends it back to Alice

```
# Compute final risk calculation score
print("[CAROL] Computing output score...")
weighted_terms = []
for i in range(3):
    term = hidden_layer_outputs[i] * w2[i]
    weighted_terms.append(term)
score = sum(weighted_terms) + b2[0]
# Save and upload
utils.write_data(LOCAL_OUTPUT, score.serialize())
upload_blob(BUCKET_NAME, LOCAL_OUTPUT, OUTPUT_BLOB)
```

Step 4: Alice receives the encrypted result and decrypts it (alice.py)

9: Alice downloads encrypted inference from Carol

10: Alice decrypts her final fraud risk score

>>> ALL DONE NOW!

```python
# Wait for Carol to respond
for _ in range(10):
    if download_from_gcs(BUCKET_NAME, result_blob, local_result_file):
        print("[ALICE] Decrypting result...")
        result_proto = utils.read_data(local_result_file)
        enc_result = ts.lazy_ckks_vector_from(result_proto)
        enc_result.link_context(context)
        score = int(min(max(enc_result.decrypt()[0], 0), 100))  # clamp between
        0 and 100
        print(f"[ALICE] Encrypted fraud risk score decrypted: {score}")
        if 0 <= score <= 20:
            print("\tScore Range: 0-20\n\tLow-Risk: Transactions normal. Faund
            unlikely :)")
        elif 21 <= score <= 50:
            print("\tScore Range: 20-50\n\tLow-Risk: Transactions are
            suspicious. Further investigation is recommended.")
        elif 51 <= score:
            print("\tScore Range: 51-100+\n\tLow-Risk: Transactions are likely
            fraudulent or strange.")
        break
    print("[ALICE] Result not ready, waiting 5s...")
    time.sleep(5)
else:
    print(f"[ALICE] Timed out waiting for result...")
    # print(f"[ALICE] Awaiting encrypted score from Carol...")
```

CLASSIFIER DESIGN

CLASSIFIER: Simple neural network (NN)

LAYERS

1. **Input layer:**

   6 features (float values)

2. **Hidden layer:**

   3 neurons

3. **Output layer:**

   1 neuron -> Fraud risk score

WHY A NEURAL NETWORK?

Neural networks work with arithmetic circuits. They tend to be expressive, but are simple enough for homomorphic evaluation.

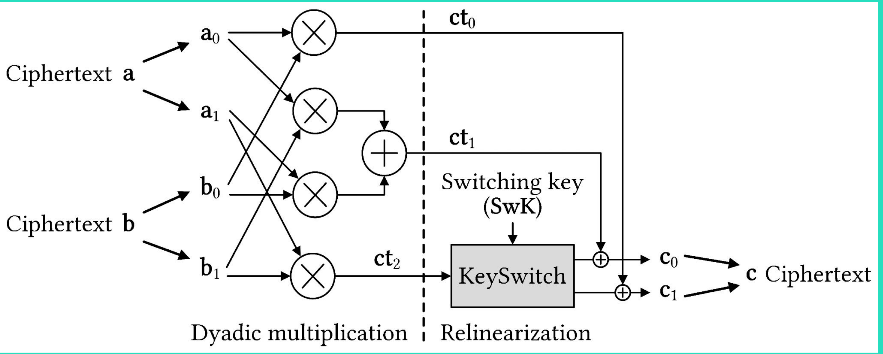
ENCRYPTION SCHEME: Cheon-Kim-Kim-Song (CKKS)

Overview:

- Supports real number operations
- Approximate arithmetic
- Ideal for ML inference

Parameters:

- ‘poly_modulus_degree = 8192’
- ‘Coeff_mod_bit_sizes = [60,40,40,60]’
- ‘Global_scale = 2**40’



Implementation Overview

Libraries

Code written in **Python** using:

1. **TenSEAL**
2. **Google Cloud Storage** (for communication)

Components

There are two components:

1. **ALICE (Local)**: Encrypt, upload, decrypt
2. **Carol (Cloud)**: Perform computation on encrypted data

Deployment

Every file used by Carol stored in '**carol_function/**' to allow for easy deployment

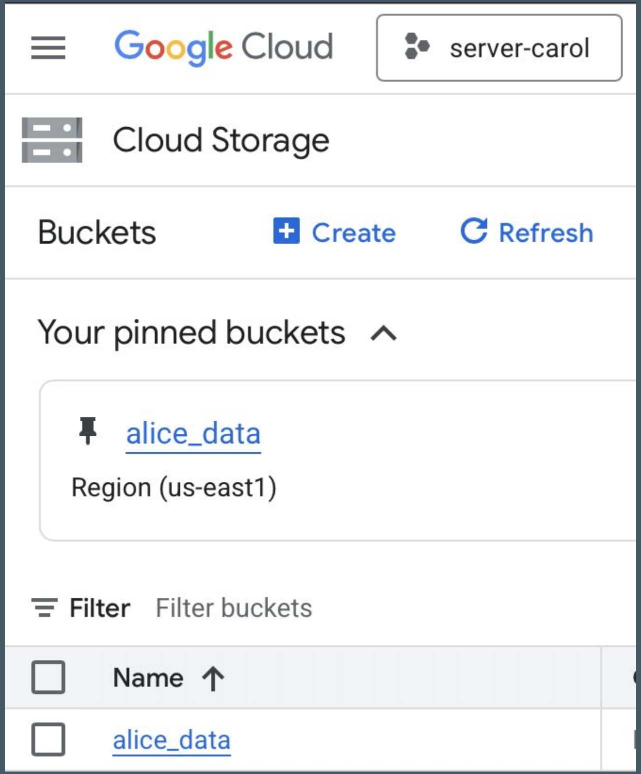
DEMO

STEP 1: Setup GCLOUD

In the google cloud engine, GCP:

1. Create project '**server-carol**'
2. Under '**server-carol**', create bucket '**alice_data**'

>>>> Your gcloud storage interface should look something like the image to the left when completed successfully:



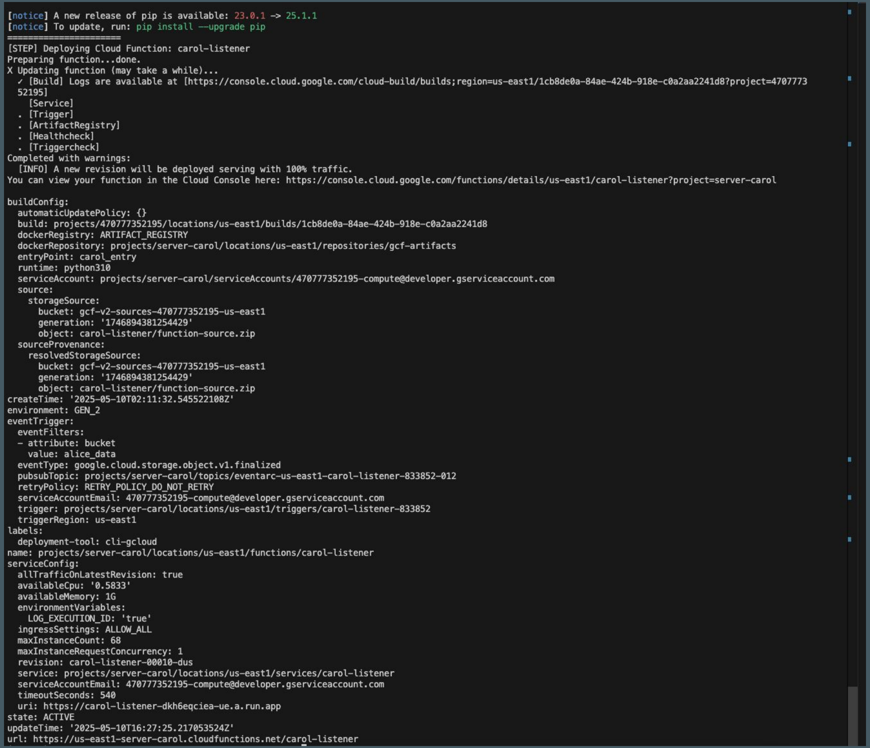
STEP 2: Deploy Carol to GCP

Then, in your terminal,run:

- chmod +x deploy.sh
- ./deploy.sh

NOTE: The deployment file takes care of a lot. Very important in this implementation.

>>> When the function is being deployed, your terminal should look something like this:



STEP 3: Run the Program

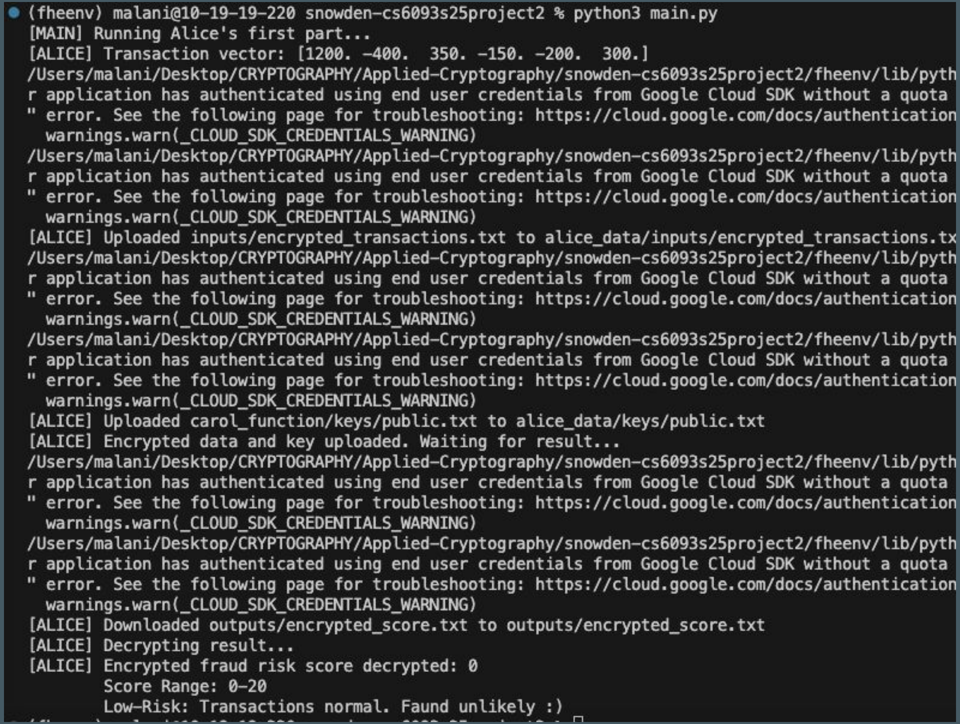
After the ‘**carol-listener**’ function is deployed to gcloud, you are all set to run the program.

To do so, in your terminal, again under the main project file, enter:

python3 main.py

Once the program is complete, Alice should have her inferred fraud risk score based on the transactions store in her file ‘**alice_data.json**’

>>> Your terminal should look like this in the end:



PERFORMANCE CONSIDERATIONS

- Runtime, latency trade-off (high, low)
- Homomorphic encryption overhead vs.pricey benefits
- **Optimization**:
  1. Avoided non-linear activations
  2. Used simple weights and small model

INFERENCE COMPARISON

	Plaintext	Encrypted
Accuracy	More likely to be closer to true value	Operations performed on encrypted values need more work to ensure correctness of result (numerical stability required)
Time	Simple, straight-forward -> Faster	Encryption overhead requires additional calculations -> Slower
Benefit	Easier to reverse inference to get the input data	Confidentiality of input data guaranteed

Program achieved:

1. Confidential ML inference
2. Secure fraud detection

Real-World Relevance:

Privacy-preserving AI in finance, healthcare, etc.

Improvements:

- Extend to larger models
- Tune performance
- Hybrid encryption

CONCLUSION