Lab No. 7: Partial Homomorphic Encryption

Objectives

- Implement and Understand Additive Homomorphic Encryption
- Implement and Understand Multiplicative Homomorphic Encryption
- Evaluate and Apply Partial Homomorphic Encryption

Introduction to Homomorphic Encryption

Homomorphic encryption allows computations on encrypted data without decryption. This enables secure analysis of sensitive information while preserving privacy. There are two main types of HE:

- **Partially Homomorphic Encryption (PHE):** Supports either addition or multiplication homomorphically.
- **Fully Homomorphic Encryption (FHE):** Supports both addition and multiplication homomorphically.

Partially Homomorphic Encryption (PHE)

PHE allows performing either addition or multiplication on encrypted data. Some evident PHE schemes include Paillier and ElGamal.

Homomorphic Addition (Paillier Cryptosystem)

This exercise demonstrates homomorphic addition using the Paillier Cryptosystem.

Problem: Add two encrypted integers (a and b).

This program demonstrates how to perform homomorphic addition on encrypted integers using the Paillier Cryptosystem.

Key functionalities and Explanations:

- generate_keypair: Generates a public/private key pair for encryption and

decryption.

- encrypt: Encrypts a message (integer) using the public key and random blinding for security.
- decrypt: Decrypts a ciphertext using the private key (optional, for demonstration only).
- homomorphic_add: Performs homomorphic addition on two encrypted messages. This works because multiplying encrypted messages under Paillier corresponds to adding the original messages.

Python Code:

Python

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes

def generate_keypair(nlength=1024):
  """Generates a public/private key pair"""
  key = RSA.generate(nlength)
  pub_key = key.publickey()
  return pub_key, key

def encrypt(pub_key, message):
  """Encrypts a message using the public key"""
  random_bytes = get_random_bytes(16)
  p, q = pub_key.n // 2, pub_key.n // 2 + 1
  while math.gcd(p, q) != 1:
    p, q = pub_key.n // 2, pub_key.n // 2 + 1
  m_dot = pow(message, 2, pub_key.n)
  r_dot = pow(int.from_bytes(random_bytes, byteorder='big'), 2, pub_key.n)
  ciphertext = m_dot * r_dot % pub_key.n
  return ciphertext

def decrypt(priv_key, ciphertext):
  """Decrypts a ciphertext using the private key"""
  p = priv_key.n // 2
```

```
  l = (ciphertext - 1) // pub_key.n
  message = math.floor(l * pow(p, -1, priv_key.n))
  return message

def homomorphic_add(ciphertext1, ciphertext2, pub_key):
  """Performs homomorphic addition on ciphertexts"""
  return ciphertext1 * ciphertext2 % pub_key.n

# Generate key pair
pub_key, priv_key = generate_keypair()

# Encrypt integers
a = 5
b = 10
ciphertext_a = encrypt(pub_key, a)
ciphertext_b = encrypt(pub_key, b)

# Homomorphic addition
ciphertext_sum = homomorphic_add(ciphertext_a, ciphertext_b, pub_key)

# Decrypt the sum (optional)
# decrypted_sum = decrypt(priv_key, ciphertext_sum)
# print(f"Decrypted sum: {decrypted_sum}")

print(f"Ciphertext of a: {ciphertext_a}")
print(f"Ciphertext of b: {ciphertext_b}")
print(f"Ciphertext of a + b: {ciphertext_sum}")
```

Secure Medical Diagnosis (ElGamal Cryptosystem)

Explanation:

ElGamal uses a public key for encryption and a private key for decryption.

encrypt creates a random blinding factor and encrypts the message.

homomorphic_comparison multiplies ciphertexts with additional manipulation to achieve encrypted comparison (>).

Decrypting the comparison result (optional) reveals if the first message was greater than the second.

The diagnosis is made based on the decrypted comparison result (demonstration only, not recommended in practice).

Problem: Perform a secure diagnosis on encrypted patient data (blood pressure).

- A doctor wants to diagnose patients with high blood pressure (> 130) without decrypting their actual blood pressure readings.
- Use ElGamal encryption, which supports homomorphic comparison.

Python Code:

Python

```
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def generate_keypair(p=1024):
  """Generates a public/private key pair"""
  while True:
    x = int.from_bytes(get_random_bytes(p // 8), byteorder='big')
    if pow(2, p - 1, p) == 1 and 1 < x < p-1:
      break
  g = 2
  y = pow(g, x, p)
  return ((p, g, y), x)  # Public key, private key

def encrypt(pub_key, message):
  """Encrypts a message using the public key"""
  p, g, y = pub_key
```

```
  r = int.from_bytes(get_random_bytes(p // 8), byteorder='big')
  while math.gcd(r, p) != 1:
    r = int.from_bytes(get_random_bytes(p // 8), byteorder='big')
  a = pow(g, r, p)
  b = (message * pow(y, r, p)) % p
  return (a, b)

def decrypt(priv_key, ciphertext):
  """Decrypts a ciphertext using the private key"""
  p, g, _ = priv_key
  a, b = ciphertext
  x = priv_key
  message = (b * pow(a, -x, p)) % p
  return message

def homomorphic_comparison(ciphertext1, ciphertext2, pub_key):
  """Performs homomorphic comparison on ciphertexts (greater than)"""
  p, g, y = pub_key
  a1, b1 = ciphertext1
  a2, b2 = ciphertext2
  return (a1 * a2) % p, (b1 * b2 * pow(y, 1, p)) % p  # Encrypted result (m1 > m2)

# Generate key pair
pub_key, priv_key = generate_keypair()

# Blood pressure readings (already encrypted)
blood_pressure1 = encrypt(pub_key, 120)
blood_pressure2 = encrypt(pub_key, 140)

# Homomorphic comparison (encrypted result)
ciphertext_comparison = homomorphic_comparison(blood_pressure1, blood_pressure2,
pub_key)

# Decrypt the comparison result (optional - for demonstration only)
```

```
# decrypted_comparison = decrypt(priv_key, ciphertext_comparison)
# print(f"Decrypted comparison: {decrypted_comparison} (True if blood pressure 1 > blood pressure 2)")

# Diagnosis based on the encrypted comparison result
diagnosis = ciphertext_comparison[0] * pow(ciphertext_comparison[1], -1, pub_key[0]) % pub_key[0]
if diagnosis > 1:
  print("Diagnosis: High Blood Pressure detected.")
else:
  print("Diagnosis: Normal Blood Pressure.")
```

Key functionalities:

- generate_keypair: Generates a public/private key pair for ElGamal encryption.
- encrypt: Encrypts a message (blood pressure reading) using the public key and a random factor.
- homomorphic_comparison: Performs homomorphic comparison on two encrypted messages. This leverages ElGamal's properties to create an encrypted result indicating if one message is greater than the other.
- decrypt (not recommended): While included for demonstration, decrypting the comparison result in a real-world scenario would reveal sensitive information. Diagnosis should rely on the encrypted comparison itself.
- Explore additional libraries like PALISADE (https://palisade-crypto.org/) for more advanced PHE functionalities.

Software Requirements:

- Python 3.x (https://www.python.org/downloads/)
- NumPy library (https://numpy.org/)

Lab Exercises

1. Implement the Paillier encryption scheme in Python. Encrypt two integers (e.g., 15 and 25) using your implementation of the Paillier encryption scheme. Print the ciphertexts. Perform an addition operation on the encrypted integers without decrypting them. Print the result of the addition in encrypted form. Decrypt the result of the addition and verify that it matches the sum of the original integers.

2. Utilize the multiplicative homomorphic property of RSA encryption. Implement a basic RSA encryption scheme in Python. Encrypt two integers (e.g., 7 and 3) using your implementation of the RSA encryption scheme. Print the ciphertexts. Perform a multiplication operation on the encrypted integers without decrypting them. Print the result of the multiplication in encrypted form. Decrypt the result of the multiplication and verify that it matches the product of the original integers.

Additional Questions

Implement similar exercise for other PHE operations (like homomorphic multiplication using ElGamal) or explore different functionalities within Paillier.

1a: Homomorphic Multiplication (ElGamal Cryptosystem): Implement ElGamal encryption and demonstrate homomorphic multiplication on encrypted messages. (ElGamal supports multiplication but not homomorphic addition.)
1b: Secure Data Sharing (Paillier): Simulate a scenario where two parties share encrypted data and perform calculations on the combined data without decryption.
1c: Secure Thresholding (PHE): Explore how PHE can be used for secure multi-party computation, where a certain number of parties need to collaborate on a computation without revealing their individual data.
1d: Performance Analysis (Benchmarking): Compare the performance of different PHE schemes (Paillier and ElGamal) for various operations.

Lab No. 8: Searchable Encryption

Objectives

Understand the Fundamentals of Searchable Encryption (SE)

Implement and Perform Encrypted Data Searches

Analyze the Security and Efficiency Trade-offs

Searchable encryption (SE)

It is a cryptographic technique that enables efficient search operations on encrypted data without compromising its confidentiality. [1] This paradigm is essential in modern data management, where data privacy and utility are often conflicting objectives. SE offers a solution by allowing authorized users to search for specific information within encrypted datasets without revealing any underlying data.

Core Concepts used in the SE

- **Encryption:** The foundational layer, where data is transformed into an unreadable format using cryptographic algorithms. Standard symmetric or asymmetric encryption techniques can be employed.
- **Indexing:** A crucial component for enabling search. Data is indexed using cryptographic primitives to generate searchable representations, often termed "encrypted indexes." These indexes must allow for efficient search operations without exposing the underlying data.
- **Query Processing:** When a search query is issued, it is encrypted and compared against the encrypted indexes. Matching results are identified, and their encrypted locations or identifiers are returned. The actual data remains encrypted until accessed by an authorized user with the decryption key.

Categories of SE

Symmetric Searchable Encryption (SSE) is a cryptographic primitive designed to reconcile the conflicting objectives of data confidentiality and searchability. In SSE, a shared secret key is employed to encrypt data and construct searchable indexes. This approach enables authorized parties to perform search operations over encrypted data without compromising the underlying information. While SSE offers advantages in terms of efficiency compared to its public-key counterpart, it necessitates careful design to prevent information leakage through the search process and address challenges related to key management and distribution.
Major Points

- SSE employs a shared secret key for both encryption and decryption.
- SSE offers relatively efficient search performance but suffers from crucial management challenges.
- Some of the SSE schemes include the Boneh-Goh-Nissim (BGN) and the Cuccioletta-Damiani-Di-Crescenzo (CDD) schemes.

Public-key searchable Encryption (PKSE) is a cryptographic technique that enables search operations on encrypted data while maintaining data confidentiality using asymmetric cryptography. Unlike Symmetric Searchable Encryption (SSE), PKSE employs a public-key/private-key pair for encryption and decryption. PKSE allows for flexible access control, as multiple users can encrypt data using the public key, while only the holder of the corresponding private key can decrypt and search the data. PKSE offers greater flexibility in key management than SSE but often incurs higher computational overhead due to public-key operations.
Major Points

- PKSE uses public-key cryptography, where data is encrypted using a public key and decrypted with the corresponding private key.
- PKSE provides greater flexibility in key management and access control.
- Typically, less efficient than SSE due to the computational overhead of public-key operations.

SSE sample code

```
import random

from Crypto.Cipher import AES

from Crypto.Random import get_random_bytes
```

```
from Crypto.Util.Padding import pad, unpad
import hashlib

def encrypt_data(key, data):
  cipher = AES.new(key, AES.MODE_CBC)
  iv = cipher.iv
  ciphertext = cipher.encrypt(pad(data.encode(), AES.block_size))
  return iv, ciphertext

def decrypt_data(key, iv, ciphertext):
  cipher = AES.new(key, AES.MODE_CBC, iv)
  plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
  return plaintext.decode()

def create_index(documents, key):
  index = {}
  for doc_id, doc in documents.items():
    for word in doc.split():
      word_hash = hashlib.sha256(word.encode()).digest()
      if word_hash not in index:
        index[word_hash] = []
      index[word_hash].append(doc_id)
  # Encrypt the index
  encrypted_index = {}
  for word_hash, doc_ids in index.items():
    encrypted_index[encrypt_data(key, word_hash)[1]] = [encrypt_data(key, str(doc_id))[1]
for doc_id in doc_ids]
```

```
  return encrypted_index

def search(encrypted_index, query, key):
  query_hash = hashlib.sha256(query.encode()).digest()
  encrypted_query_hash = encrypt_data(key, query_hash)[1]
  if encrypted_query_hash in encrypted_index:
    return [decrypt_data(key, *encrypt_data(key, doc_id))[0] for doc_id in
encrypted_index[encrypted_query_hash]]
  else:
    return []

# Example usage
documents = {
  "doc1": "this is a document with some words",
  "doc2": "another document with different words",
  "doc3": "yet another document with some common words"
}

key = get_random_bytes(16)

encrypted_index = create_index(documents, key)

query = "document"
results = search(encrypted_index, query, key)
print(results)
```

PKSE Sample Code:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Hash import SHA256
import random
def generate_keys():
keyPair = RSA.generate(2048)
pubKey = keyPair.publickey()
privKey = keyPair
return pubKey, privKey
def encrypt_data(pubKey, data):
cipher = PKCS1_OAEP.new(pubKey)
ciphertext = cipher.encrypt(data.encode())
return ciphertext
def decrypt_data(privKey, ciphertext):
cipher = PKCS1_OAEP.new(privKey)
plaintext = cipher.decrypt(ciphertext)
return plaintext.decode()
def create_index(documents, pubKey):
index = {}
for doc_id, doc in documents.items():
for word in doc.split():
word_hash = SHA256.new(word.encode()).digest()
if word_hash not in index:
index[word_hash] = []
index[word_hash].append(doc_id)
```

```
# Encrypt the index
encrypted_index = {}
for word_hash, doc_ids in index.items():
encrypted_index[encrypt_data(pubKey, word_hash)] = [encrypt_data(pubKey, str(doc_id)) for doc_id in doc_ids]
return encrypted_index
def search(encrypted_index, query, pubKey, privKey):
query_hash = SHA256.new(query.encode()).digest()
encrypted_query_hash = encrypt_data(pubKey, query_hash)
if encrypted_query_hash in encrypted_index:
encrypted_doc_ids = encrypted_index[encrypted_query_hash]
doc_ids = [decrypt_data(privKey, doc_id) for doc_id in encrypted_doc_ids]
return doc_ids
else:
return []
# Example usage
documents = {
"doc1": "this is a document with some words",
"doc2": "another document with different words",
"doc3": "yet another document with some common words"
}
pubKey, privKey = generate_keys()
encrypted_index = create_index(documents, pubKey)
query = "document"
results = search(encrypted_index, query, pubKey, privKey)
print(results)
```

_____

Lab Exercise 1: Execute the following for SSE:

1a. Create a dataset: Generate a text corpus of at least ten documents. Each document should contain multiple words.

1b. Implement encryption and decryption functions: Use the AES encryption and decryption functions.

1c. Create an inverted index: Build an inverted index mapping word to the list of document IDs containing those words.

- Encrypt the index using the provided encryption function.

1d. Implement the search function:

- Take a search query as input.
- Encrypt the query.
- Search the encrypted index for matching terms.
- Decrypt the returned document IDs and display the corresponding documents

Lab Exercise 2: Execute the following for PKSE:

2a. Create a dataset:

- Generate a text corpus of at least ten documents. Each document should contain multiple words.

2b. Implement encryption and decryption functions:

- Use the Paillier cryptosystem for encryption and decryption.

2c. Create an encrypted index:

- Build an inverted index mapping word to the list of document IDs containing those words.

- Encrypt the index using the Paillier cryptosystem.

2d. Implement the search function:

- Take a search query as input.
- Encrypt the query using the public key.
- Search the encrypted index for matching terms.
- Decrypt the returned document IDs using the private key.

Additional Questions:

1. Demonstrate how to securely store and transmit data using GnuPG. Additionally, show how to create a digital signature for the data and verify the signature after transmission.
2. Configure and use Snort as a Network Intrusion Detection System (NIDS) to monitor real-time network traffic. Capture network traffic, apply Snort rules, and analyze the logs to identify any potential intrusions.