Cryptography

Lab Assignment 4

Github Link :

https://github.com/AviatrixK/Uni-Assignments/tree/main/Crypto-assignments/Advanced_Crypto_Assignment_Kiruthika blockchain_sim.py

```
"""
Q4: Blockchain Cryptography Simulation
Demonstrates hashing, digital signatures, block
chaining, and tamper-proofing
"""


import hashlib
import json
import time
from cryptography.hazmat.primitives.asymmetric
import rsa, padding
from cryptography.hazmat.primitives import hashes,
serialization
from cryptography.hazmat.backends import
default_backend
import base64


class MerkleTree:
    """Implements Merkle Tree for efficient
transaction verification"""


    @staticmethod
    def hash_data(data):
        """Hash a piece of data"""
```

```
        return hashlib.sha256(data.encode() if
isinstance(data, str) else data).hexdigest()

    @staticmethod
    def build_tree(transactions):
        """Build Merkle tree from transactions"""
        if not transactions:
            return None

        # Hash all transactions
        current_level = [MerkleTree.hash_data(tx)
for tx in transactions]

        # Build tree bottom-up
        tree = [current_level[:]]

        while len(current_level) > 1:
            next_level = []

            # Process pairs
            for i in range(0, len(current_level),
2):
                left = current_level[i]
                right = current_level[i + 1] if i +
1 < len(current_level) else left
```

```
                combined = left + right
                parent_hash =
MerkleTree.hash_data(combined)
                next_level.append(parent_hash)

            tree.append(next_level)
            current_level = next_level

        return tree

    @staticmethod
    def get_root(tree):
        """Get Merkle root (top of tree)"""
        return tree[-1][0] if tree else None

    @staticmethod
    def get_proof(tree, transaction_index):
        """Get Merkle proof for a transaction"""
        proof = []
        index = transaction_index

        for level in tree[:-1]:
            if index % 2 == 0:
                sibling_index = index + 1
                if sibling_index < len(level):
```

```
                    proof.append(('right',
level[sibling_index]))
            else:
                sibling_index = index - 1
                proof.append(('left',
level[sibling_index]))

            index = index // 2

        return proof

class Block:
    """Represents a single block in the
blockchain"""

    def __init__(self, index, transactions,
previous_hash, timestamp=None):
        self.index = index
        self.timestamp = timestamp or time.time()
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.nonce = 0
        self.merkle_root =
self._calculate_merkle_root()
        self.hash = self.calculate_hash()
```

```
    def _calculate_merkle_root(self):
        """Calculate Merkle root of transactions"""
        if not self.transactions:
            return "0" * 64

        tree =
MerkleTree.build_tree([json.dumps(tx,
sort_keys=True) for tx in self.transactions])
        return MerkleTree.get_root(tree)

    def calculate_hash(self):
        """Calculate block hash"""
        block_string = json.dumps({
            "index": self.index,
            "timestamp": self.timestamp,
            "transactions": self.transactions,
            "previous_hash": self.previous_hash,
            "merkle_root": self.merkle_root,
            "nonce": self.nonce
        }, sort_keys=True)

        return
hashlib.sha256(block_string.encode()).hexdigest()

    def mine_block(self, difficulty):
```

```
        """Proof of Work mining"""
        target = "0" * difficulty

        print(f"  Mining block {self.index}...", 
end="")
        start_time = time.time()

        while self.hash[:difficulty] != target:
            self.nonce += 1
            self.hash = self.calculate_hash()

        elapsed = time.time() - start_time
        print(f" Mined! Hash: {self.hash} (Nonce: 
{self.nonce}, Time: {elapsed:.2f}s)")

    def to_dict(self):
        """Convert block to dictionary"""
        return {
            "index": self.index,
            "timestamp": self.timestamp,
            "transactions": self.transactions,
            "previous_hash": self.previous_hash,
            "merkle_root": self.merkle_root,
            "nonce": self.nonce,
            "hash": self.hash
```

```
        }


class Wallet:
    """Cryptocurrency wallet with public/private
key pair"""


    def __init__(self, name):
        self.name = name
        self.private_key =
rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        self.public_key =
self.private_key.public_key()
        self.address = self._generate_address()


    def _generate_address(self):
        """Generate wallet address from public
key"""
        public_bytes =
self.public_key.public_bytes(
            encoding=serialization.Encoding.DER,
            format=serialization.PublicFormat.Subje
ctPublicKeyInfo
```

```
        )
        return
hashlib.sha256(public_bytes).hexdigest()[:40]

    def sign_transaction(self, transaction):
        """Sign a transaction"""
        tx_string = json.dumps(transaction,
sort_keys=True)
        signature = self.private_key.sign(
            tx_string.encode(),
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return base64.b64encode(signature).decode()

    @staticmethod
    def verify_signature(public_key, transaction,
signature):
        """Verify transaction signature"""
        try:
            tx_string = json.dumps(transaction,
sort_keys=True)
            public_key.verify(
```

```
                base64.b64decode(signature),
                tx_string.encode(),
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256(
)),
                    salt_length=padding.PSS.MAX_LEN
GTH
                ),
                hashes.SHA256()
            )
            return True
        except:
            return False

class Blockchain:
    """Simple blockchain implementation"""

    def __init__(self, difficulty=4):
        self.chain = []
        self.difficulty = difficulty
        self.pending_transactions = []
        self.mining_reward = 50
        self.wallets = {}

        # Create genesis block
```

```
        self.create_genesis_block()

    def create_genesis_block(self):
        """Create the first block"""
        genesis_block = Block(0, [{"type": "genesis", "data": "Genesis Block"}], "0")
        genesis_block.mine_block(self.difficulty)
        self.chain.append(genesis_block)
        print(f" Genesis block created: {genesis_block.hash}")

    def get_latest_block(self):
        """Get the most recent block"""
        return self.chain[-1]

    def add_transaction(self, transaction):
        """Add a transaction to pending transactions"""
        self.pending_transactions.append(transaction)
        print(f" Transaction added: {transaction['from'][:10]} → {transaction['to'][:10]} ({transaction['amount']} coins)")

    def mine_pending_transactions(self, miner_address):
```

```
        """Mine pending transactions into a new
block"""
        # Create new block with pending
transactions
        block = Block(
            index=len(self.chain),
            transactions=self.pending_transactions,
            previous_hash=self.get_latest_block().h
ash
        )

        # Mine the block
        block.mine_block(self.difficulty)

        # Add to chain
        self.chain.append(block)

        # Reset pending transactions and add mining
reward
        self.pending_transactions = [
            {
                "from": "NETWORK",
                "to": miner_address,
                "amount": self.mining_reward,
                "type": "mining_reward"
            }
```

```
        ]

        print(f" Block {block.index} added to
chain")

    def is_chain_valid(self):
        """Validate entire blockchain"""
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Check hash integrity
            if current_block.hash !=
current_block.calculate_hash():
                print(f" Block {i} has been
tampered with!")
                return False

            # Check chain linkage
            if current_block.previous_hash !=
previous_block.hash:
                print(f" Block {i} previous_hash
doesn't match!")
                return False

            # Check proof of work
```

```
            if not
current_block.hash.startswith("0" *
self.difficulty):
                print(f" Block {i} doesn't meet
difficulty requirement!")
                return False

        print(" Blockchain is valid!")
        return True

    def get_balance(self, address):
        """Calculate balance for an address"""
        balance = 0

        for block in self.chain:
            for transaction in block.transactions:
                if transaction.get('from') ==
address:
                    balance -=
transaction.get('amount', 0)
                if transaction.get('to') ==
address:
                    balance +=
transaction.get('amount', 0)

        return balance
```

```
    def demonstrate_tampering(self):
        """Show what happens when blockchain is
tampered with"""
        print("\n" + "=" * 70)
        print("TAMPER DETECTION DEMONSTRATION")
        print("=" * 70)

        print("\n Original blockchain state:")
        self.print_chain()

        print("\n Tampering with block 1...")
        # Try to change a transaction in block 1
        if len(self.chain) > 1:
            original_tx =
self.chain[1].transactions[0].copy()
            self.chain[1].transactions[0]['amount']
= 999999

            print(f"   Changed amount from
{original_tx.get('amount')} to 999999")
            print("\n Validating blockchain after
tampering...")

            self.is_chain_valid()

            # Restore
```

```
            self.chain[1].transactions[0] =
original_tx

    def print_chain(self):
        """Print blockchain summary"""
        print(f"\n{'Block':<8} {'Hash':<20}
{'Previous Hash':<20} {'Transactions':<15}")
        print("-" * 70)
        for block in self.chain:
            print(f"{block.index:<8}
{block.hash[:16]:<20}
{block.previous_hash[:16]:<20}
{len(block.transactions):<15}")


def demonstrate_blockchain():
    """Main blockchain demonstration"""

    print("=" * 70)
    print("BLOCKCHAIN CRYPTOGRAPHY DEMONSTRATION")
    print("=" * 70)

    # Create blockchain with difficulty 3 (for
faster demo)
    blockchain = Blockchain(difficulty=3)

    # Create wallets
```

```
    alice = Wallet("Alice")
    bob = Wallet("Bob")
    miner = Wallet("Miner")

    print(f"\n Alice's address: {alice.address}")
    print(f" Bob's address: {bob.address}")
    print(f" Miner's address: {miner.address}")

    print("\n" + "=" * 70)
    print("TRANSACTION CREATION AND SIGNING")
    print("=" * 70)

    # Create and sign transaction
    transaction1 = {
        "from": alice.address,
        "to": bob.address,
        "amount": 30,
        "timestamp": time.time()
    }

    signature1 =
alice.sign_transaction(transaction1)
    transaction1["signature"] = signature1

    print(f"  Alice signed transaction")
```

```
    print(f"   Signature: {signature1[:50]}...")

    # Verify signature
    verified = 
Wallet.verify_signature(alice.public_key,
                                                   {k: v for k, 
v in transaction1.items() if k != 'signature'},
                                                   signature1)
    print(f"   Verification: {' Valid' if verified 
else ' Invalid'}")

    # Add transactions
    blockchain.add_transaction(transaction1)

    transaction2 = {
        "from": alice.address,
        "to": bob.address,
        "amount": 20,
        "timestamp": time.time()
    }
    blockchain.add_transaction(transaction2)

    # Mine block
    print("\n" + "=" * 70)
    print("MINING BLOCK")
```

```
    print("=" * 70)
    blockchain.mine_pending_transactions(miner.addr
ess)

    # Add more transactions
    transaction3 = {
        "from": bob.address,
        "to": alice.address,
        "amount": 10,
        "timestamp": time.time()
    }
    blockchain.add_transaction(transaction3)

    blockchain.mine_pending_transactions(miner.addr
ess)

    # Print blockchain
    print("\n" + "=" * 70)
    print("BLOCKCHAIN STATE")
    print("=" * 70)
    blockchain.print_chain()

    # Check balances
    print("\n" + "=" * 70)
    print("ACCOUNT BALANCES")
```

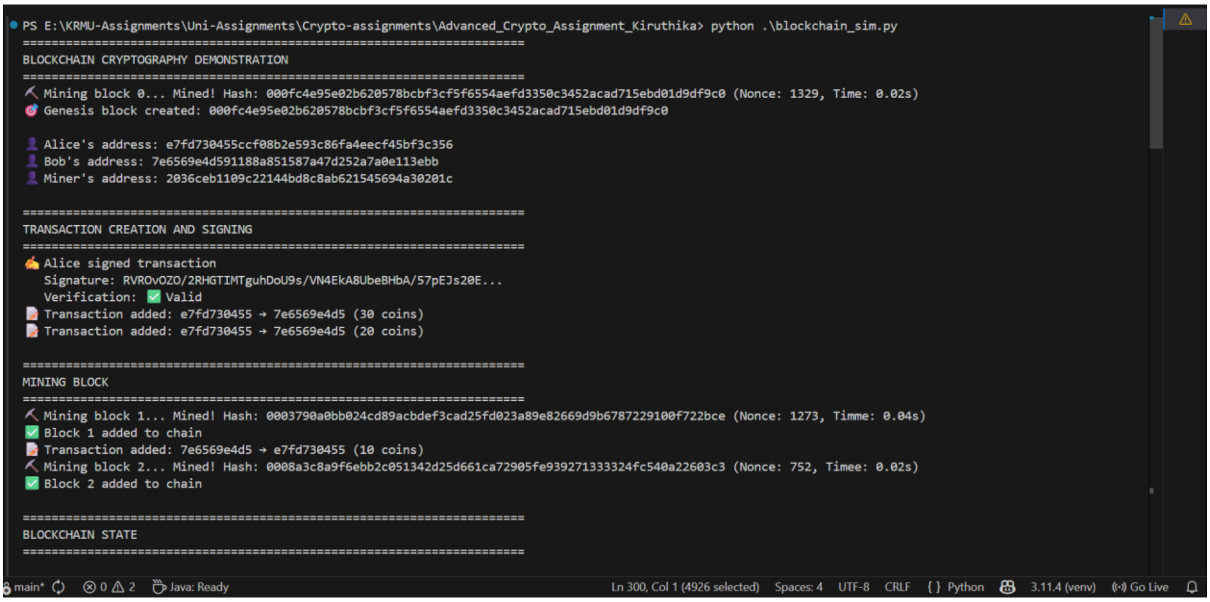
```
    print("=" * 70)
    print(f"Alice:
{blockchain.get_balance(alice.address)} coins")
    print(f"Bob:
{blockchain.get_balance(bob.address)} coins")
    print(f"Miner:
{blockchain.get_balance(miner.address)} coins")

    # Validate blockchain
    print("\n" + "=" * 70)
    print("BLOCKCHAIN VALIDATION")
    print("=" * 70)
    blockchain.is_chain_valid()

    # Demonstrate tampering
    blockchain.demonstrate_tampering()

    # Merkle Tree demonstration
    print("\n" + "=" * 70)
    print("MERKLE TREE DEMONSTRATION")
    print("=" * 70)

    transactions = ["tx1", "tx2", "tx3", "tx4",
"tx5"]
    tree = MerkleTree.build_tree(transactions)
    root = MerkleTree.get_root(tree)
```

```
    print(f"Transactions: {transactions}")
    print(f"Merkle Root: {root}")
    print(f"\nMerkle Tree Structure:")
    for i, level in enumerate(reversed(tree)):
        print(f"  Level {len(tree) - i - 1}:
{level}")

    # Get proof for transaction
    proof = MerkleTree.get_proof(tree, 0)
    print(f"\nMerkle Proof for tx1: {proof}")

    print("\n" + "=" * 70)
    print("CONSENSUS MECHANISMS")
    print("=" * 70)

    consensus_info = """
    1. PROOF OF WORK (PoW) - Used by Bitcoin
       - Miners compete to solve cryptographic
puzzle
       - First to solve gets to add block and
receive reward
       - Difficulty adjusts to maintain block time
       - Pros: Secure, battle-tested
       - Cons: Energy intensive
```

```
    2. PROOF OF STAKE (PoS) - Used by Ethereum 2.0
       - Validators chosen based on stake amount
       - More energy efficient
       - Risk of "rich get richer"
       - Pros: Energy efficient, faster
       - Cons: Less proven, potential
centralization

    3. DELEGATED PROOF OF STAKE (DPoS)
       - Token holders vote for delegates
       - Delegates validate blocks
       - Faster but more centralized

    4. PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT)
       - Used in permissioned blockchains
       - Nodes reach consensus through voting
       - Fast but requires known validators
    """

    print(consensus_info)

    # Save blockchain to file
    print("\n Saving blockchain to file...")
    blockchain_data = {
```

```
        "chain": [block.to_dict() for block in
blockchain.chain],
        "difficulty": blockchain.difficulty
    }

    with open("blockchain_data.json", "w") as f:
        json.dump(blockchain_data, f, indent=2)

    print(" Blockchain saved to
blockchain_data.json")

if __name__ == "__main__":
    demonstrate_blockchain()
```

Output:



```
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\Advanced_Crypto_Assignment_Kiruthika> python .\blockchain_sim.py
======================================================================
BLOCKCHAIN CRYPTOGRAPHY DEMONSTRATION
======================================================================
⛏ Mining block 0... Mined! Hash: 000fc4e95e02b620578bcbf3cf5f6554aefd3350c3452acad715ebd01d9df9c0 (Nonce: 1329, Time: 0.02s)
🎯 Genesis block created: 000fc4e95e02b620578bcbf3cf5f6554aefd3350c3452acad715ebd01d9df9c0

👤 Alice's address: e7fd730455ccf08b2e593c86fa4eecf45bf3c356
👤 Bob's address: 7e6569e4d591188a851587a47d252a7a0e113ebb
👤 Miner's address: 2036ceb1109c22144bd8c8ab621545694a30201c

======================================================================
TRANSACTION CREATION AND SIGNING
======================================================================
✍ Alice signed transaction
   Signature: RVROvOZO/2RHGTIMTguhDoU9s/VN4EkA8UbeBHbA/57pEJs20E...
   Verification: ✅ Valid
📝 Transaction added: e7fd730455 → 7e6569e4d5 (30 coins)
📝 Transaction added: e7fd730455 → 7e6569e4d5 (20 coins)

======================================================================
MINING BLOCK
======================================================================
⛏ Mining block 1... Mined! Hash: 0003790a0bb024cd89acbdef3cad25fd023a89e82669d9b6787229100f722bce (Nonce: 1273, Timme: 0.04s)
✅ Block 1 added to chain
📝 Transaction added: 7e6569e4d5 → e7fd730455 (10 coins)
⛏ Mining block 2... Mined! Hash: 0008a3c8a9f6ebb2c051342d25d661ca72905fe939271333324fc540a22603c3 (Nonce: 752, Timee: 0.02s)
✅ Block 2 added to chain

======================================================================
BLOCKCHAIN STATE
======================================================================
```

```
Block    Hash                 Previous Hash        Transactions
----------------------------------------------------------------------
0        000fc4e95e02b620     0                    1
1        0003790a0bb024cd     000fc4e95e02b620     2
2        0008a3c8a9f6ebb2     0003790a0bb024cd     2

======================================================================
ACCOUNT BALANCES
======================================================================
Alice: -40 coins
Bob: 40 coins
Miner: 50 coins

======================================================================
BLOCKCHAIN VALIDATION
======================================================================
✅ Blockchain is valid!

======================================================================
TAMPER DETECTION DEMONSTRATION
======================================================================

📊 Original blockchain state:

Block    Hash                 Previous Hash        Transactions
----------------------------------------------------------------------
0        000fc4e95e02b620     0                    1
1        0003790a0bb024cd     000fc4e95e02b620     2
2        0008a3c8a9f6ebb2     0003790a0bb024cd     2

🔧 Tampering with block 1...
   Changed amount from 30 to 999999
```

```
🔍 Validating blockchain after tampering...
❌ Block 1 has been tampered with!

======================================================================
MERKLE TREE DEMONSTRATION
======================================================================
Transactions: ['tx1', 'tx2', 'tx3', 'tx4', 'tx5']
Merkle Root: db60ce68c3176600258a40668f6e1a54198cf1d9239e7748276cb84d73d7a5ff

Merkle Tree Structure:
  Level 3: ['db60ce68c3176600258a40668f6e1a54198cf1d9239e7748276cb84d73d7a5ff']
  Level 2: ['773bc304a3b0a626a520a8d6eacc36809ac18c0b174f3ff3cdaf0a4e9c64433d', 'c30d1c12d240decad5d2e5921647bebd24954de45f8f03de4e79694dc94c1ba1']
  Level 1: ['f8f28ede979567036d801ad6cf58b551c7d8530bba005c48e46d39c73ab52664', '850cf301915d09ebcfa84e2ee4087025e17a6fca7e4149ce02cff94cd3db55de', 'bcbdf12a
6a4fa31e81924aa9e4b1c6b5e06b7611e08e5f2f2254739623378b83']
  Level 0: ['709b55bd3da0f5a838125bd0ee20c5bfdd7caba173912d4281cae816b79a201b', '27ca64c092a959c7edc525ed45e845b1de6a7590d173fd2fad9133c8a779a1e3', '1f3cb18e
896256d7d6bb8c11a6ec71f005c75de05e39beae5d93bbd1e2c8b7a9', '41b637cfd9eb3e2f60f734f9ca44e5c1559c6f481d49d6ed6891f3e9a086ac78', 'a8c0cce8bb067e91cf2766c26be4e
5d7cfba3d3323dc19d08a834391a1ce5acf']

Merkle Proof for tx1: [('right', '27ca64c092a959c7edc525ed45e845b1de6a7590d173fd2fad9133c8a779a1e3'), ('right', '850cf301915d09ebcfa84e2ee4087025e17a6fca7e41
49ce02cff94cd3db55de'), ('right', 'c30d1c12d240decad5d2e5921647bebd24954de45f8f03de4e79694dc94c1ba1')]

======================================================================
CONSENSUS MECHANISMS
======================================================================

    1. PROOF OF WORK (PoW) - Used by Bitcoin
       - Miners compete to solve cryptographic puzzle
       - First to solve gets to add block and receive reward
       - Difficulty adjusts to maintain block time
       - Pros: Secure, battle-tested
       - Cons: Energy intensive

    2. PROOF OF STAKE (PoS) - Used by Ethereum 2.0
```

```
    2. PROOF OF STAKE (PoS) - Used by Ethereum 2.0
       - Validators chosen based on stake amount
       - More energy efficient
       - Risk of "rich get richer"
       - Pros: Energy efficient, faster
       - Cons: Less proven, potential centralization

    3. DELEGATED PROOF OF STAKE (DPoS)
       - Token holders vote for delegates
       - Delegates validate blocks
       - Faster but more centralized

    4. PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT)
       - Used in permissioned blockchains
       - Nodes reach consensus through voting
       - Fast but requires known validators


💾 Saving blockchain to file...
✅ Blockchain saved to blockchain_data.json
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\Advanced_Crypto_Assignment_Kiruthika>
```

key_distribution.py

```
"""
Q1: Key Management and Distribution System
Simulates a Key Distribution Center (KDC) for
symmetric key distribution
"""


import os
import json
import time
from cryptography.hazmat.primitives.ciphers import
Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes,
serialization
from cryptography.hazmat.primitives.asymmetric
import rsa, padding
from cryptography.hazmat.backends import
default_backend
import base64


class KeyDistributionCenter:
    """Simulates a trusted KDC for key
management"""


    def __init__(self):
        self.registered_users = {}
```

```
        self.session_keys = {}
        self.key_versions = {}  # For key rotation
        print(" Key Distribution Center 
Initialized")

    def register_user(self, user_id, master_key):
        """Register a user with their master key"""
        self.registered_users[user_id] = master_key
        self.key_versions[user_id] = 1
        print(f" User '{user_id}' registered with 
KDC")

    def generate_session_key(self, user_a, user_b):
        """Generate a session key for communication 
between two users"""
        if user_a not in self.registered_users or 
user_b not in self.registered_users:
            raise ValueError("One or both users not 
registered!")

        # Generate random session key
        session_key = os.urandom(32)  # 256-bit key
        session_id = 
f"{user_a}_{user_b}_{int(time.time())}"

        # Encrypt session key with each user's 
master key
```

```
        encrypted_for_a =
self.encrypt_with_master_key(session_key, user_a)
        encrypted_for_b =
self.encrypt_with_master_key(session_key, user_b)

        self.session_keys[session_id] = {
            'session_key': session_key,
            'created_at': time.time(),
            'users': [user_a, user_b]
        }

        print(f" Session key generated for {user_a}
↔ {user_b}")
        return {
            'session_id': session_id,
            'key_for_a': encrypted_for_a,
            'key_for_b': encrypted_for_b
        }

    def encrypt_with_master_key(self, data,
user_id):
        """Encrypt data using user's master key
(AES-256)"""
        master_key = self.registered_users[user_id]
        iv = os.urandom(16)
```

```
        cipher = Cipher(algorithms.AES(master_key),
modes.CBC(iv), backend=default_backend())
        encryptor = cipher.encryptor()

        # Pad data to 16-byte boundary
        padded_data = data + b'\x00' * (16 -
len(data) % 16)
        encrypted = encryptor.update(padded_data) +
encryptor.finalize()

        return base64.b64encode(iv +
encrypted).decode()

    def decrypt_with_master_key(self,
encrypted_data, user_id):
        """Decrypt data using user's master key"""
        master_key = self.registered_users[user_id]
        decoded = base64.b64decode(encrypted_data)
        iv = decoded[:16]
        ciphertext = decoded[16:]

        cipher = Cipher(algorithms.AES(master_key),
modes.CBC(iv), backend=default_backend())
        decryptor = cipher.decryptor()
        decrypted = decryptor.update(ciphertext) +
decryptor.finalize()
```

```
        return decrypted.rstrip(b'\x00')

    def rotate_key(self, user_id):
        """Simulate key rotation for a user"""
        if user_id not in self.registered_users:
            raise ValueError("User not registered!")

        old_version = self.key_versions[user_id]
        new_master_key = os.urandom(32)

        # Archive old key (in real system, you'd have proper key archival)
        old_key = self.registered_users[user_id]

        self.registered_users[user_id] = new_master_key
        self.key_versions[user_id] += 1

        print(f" Key rotated for '{user_id}': v{old_version} → v{self.key_versions[user_id]}")
        return new_master_key

class AsymmetricKeyDistribution:
    """Demonstrates asymmetric key distribution (PKI model)"""
```

```
    def __init__(self):
        self.public_keys = {}
        self.private_keys = {}

    def generate_key_pair(self, user_id):
        """Generate RSA key pair for a user"""
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        public_key = private_key.public_key()

        self.private_keys[user_id] = private_key
        self.public_keys[user_id] = public_key

        print(f" RSA key pair generated for
'{user_id}'")
        return public_key

    def encrypt_message(self, message,
recipient_id):
        """Encrypt message using recipient's public
key"""
```

```
        if recipient_id not in self.public_keys:
            raise ValueError("Recipient's public
key not available!")

        public_key = self.public_keys[recipient_id]
        encrypted = public_key.encrypt(
            message.encode(),
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.S
HA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return base64.b64encode(encrypted).decode()

    def decrypt_message(self, encrypted_message,
recipient_id):
        """Decrypt message using recipient's
private key"""
        private_key =
self.private_keys[recipient_id]
        encrypted =
base64.b64decode(encrypted_message)

        decrypted = private_key.decrypt(
```

```
            encrypted,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.S
HA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return decrypted.decode()

def demonstrate_key_distribution():
    """Main demonstration of key distribution
systems"""

    print("=" * 70)
    print("SYMMETRIC KEY DISTRIBUTION WITH KDC")
    print("=" * 70)

    # Initialize KDC
    kdc = KeyDistributionCenter()

    # Register users with master keys
    alice_master = os.urandom(32)
    bob_master = os.urandom(32)
```

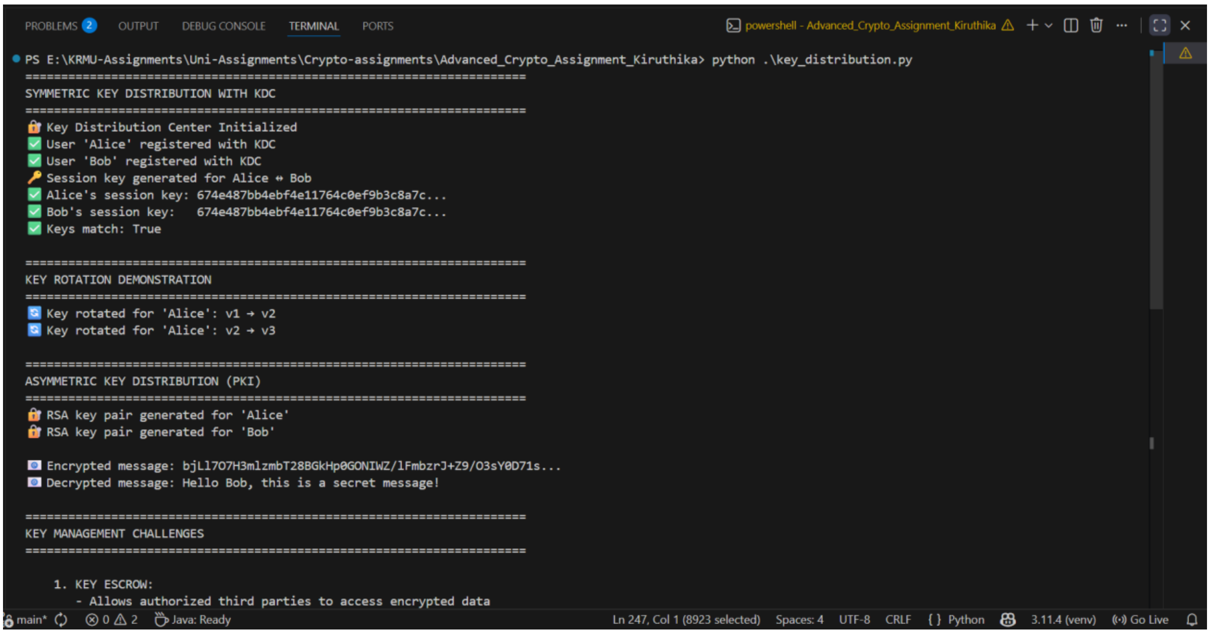
```
    kdc.register_user("Alice", alice_master)
    kdc.register_user("Bob", bob_master)

    # Generate session key
    session_info =
kdc.generate_session_key("Alice", "Bob")

    # Alice decrypts her copy of the session key
    alice_session_key =
kdc.decrypt_with_master_key(session_info['key_for_a
'], "Alice")
    bob_session_key =
kdc.decrypt_with_master_key(session_info['key_for_b
'], "Bob")

    print(f" Alice's session key:
{alice_session_key.hex()[:32]}...")
    print(f" Bob's session
key:   {bob_session_key.hex()[:32]}...")
    print(f" Keys match: {alice_session_key ==
bob_session_key}")

    # Demonstrate key rotation
    print("\n" + "=" * 70)
    print("KEY ROTATION DEMONSTRATION")
    print("=" * 70)
    kdc.rotate_key("Alice")
```

```
    kdc.rotate_key("Alice")  # Rotate twice

    print("\n" + "=" * 70)
    print("ASYMMETRIC KEY DISTRIBUTION (PKI)")
    print("=" * 70)

    # Asymmetric key distribution
    pki = AsymmetricKeyDistribution()

    # Generate key pairs
    pki.generate_key_pair("Alice")
    pki.generate_key_pair("Bob")

    # Alice sends encrypted message to Bob
    message = "Hello Bob, this is a secret
message!"
    encrypted = pki.encrypt_message(message, "Bob")
    print(f"\n Encrypted message:
{encrypted[:50]}...")

    decrypted = pki.decrypt_message(encrypted,
"Bob")
    print(f" Decrypted message: {decrypted}")

    print("\n" + "=" * 70)
```

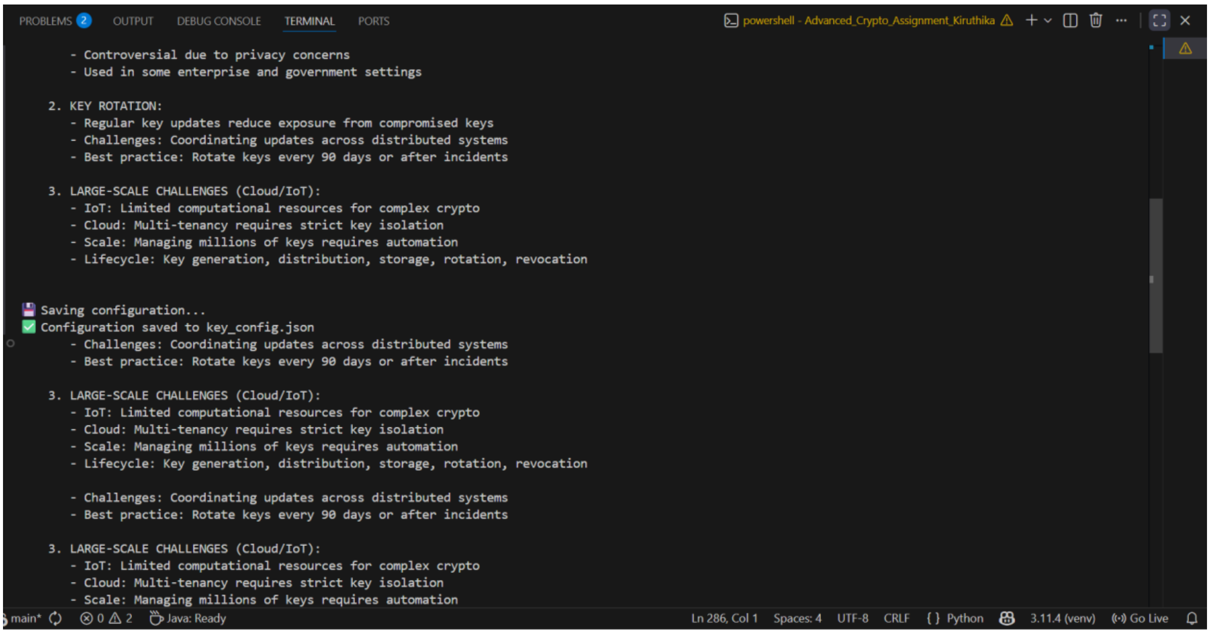
```
    print("KEY MANAGEMENT CHALLENGES")
    print("=" * 70)
    print("""
    1. KEY ESCROW:
       - Allows authorized third parties to access encrypted data
       - Controversial due to privacy concerns
       - Used in some enterprise and government settings

    2. KEY ROTATION:
       - Regular key updates reduce exposure from compromised keys
       - Challenges: Coordinating updates across distributed systems
       - Best practice: Rotate keys every 90 days or after incidents

    3. LARGE-SCALE CHALLENGES (Cloud/IoT):
       - IoT: Limited computational resources for complex crypto
       - Cloud: Multi-tenancy requires strict key isolation
       - Scale: Managing millions of keys requires automation
       - Lifecycle: Key generation, distribution, storage, rotation, revocation
```

```
    """)

if __name__ == "__main__":
    demonstrate_key_distribution()

    print("\n Saving configuration...")
    config = {
        "kdc_type": "symmetric",
        "key_algorithm": "AES-256",
        "asymmetric_algorithm": "RSA-2048",
        "key_rotation_period": "90 days"
    }

    with open("key_config.json", "w") as f:
        json.dump(config, f, indent=2)

    print(" Configuration saved to
key_config.json")
```

Output:



PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell - Advanced_Crypto_Assignment_Kiruthika
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\Advanced_Crypto_Assignment_Kiruthika> python .\key_distribution.py
==
SYMMETRIC KEY DISTRIBUTION WITH KDC
==
Key Distribution Center Initialized
User 'Alice' registered with KDC
User 'Bob' registered with KDC
Session key generated for Alice ↔ Bob
Alice's session key: 674e487bb4ebf4e11764c0ef9b3c8a7c...
Bob's session key: 674e487bb4ebf4e11764c0ef9b3c8a7c...
Keys match: True
==
KEY ROTATION DEMONSTRATION
==
Key rotated for 'Alice': v1 → v2
Key rotated for 'Alice': v2 → v3
==
ASYMMETRIC KEY DISTRIBUTION (PKI)
==
RSA key pair generated for 'Alice'
RSA key pair generated for 'Bob'
Encrypted message: bjLl7O7H3mlzmbT28BGkHp0GONIWZ/lFmbzrJ+Z9/O3sY0D71s...
Decrypted message: Hello Bob, this is a secret message!
==
KEY MANAGEMENT CHALLENGES
==
1. KEY ESCROW:
- Allows authorized third parties to access encrypted data
main* ⊗ 0 ⚠ 2 Java: Ready
Ln 247, Col 1 (8923 selected) Spaces: 4 UTF-8 CRLF { } Python 3.11.4 (venv) Go Live



PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell - Advanced_Crypto_Assignment_Kiruthika
- Controversial due to privacy concerns
- Used in some enterprise and government settings
2. KEY ROTATION:
- Regular key updates reduce exposure from compromised keys
- Challenges: Coordinating updates across distributed systems
- Best practice: Rotate keys every 90 days or after incidents
3. LARGE-SCALE CHALLENGES (Cloud/IoT):
- IoT: Limited computational resources for complex crypto
- Cloud: Multi-tenancy requires strict key isolation
- Scale: Managing millions of keys requires automation
- Lifecycle: Key generation, distribution, storage, rotation, revocation
Saving configuration...
Configuration saved to key_config.json
- Challenges: Coordinating updates across distributed systems
- Best practice: Rotate keys every 90 days or after incidents
3. LARGE-SCALE CHALLENGES (Cloud/IoT):
- IoT: Limited computational resources for complex crypto
- Cloud: Multi-tenancy requires strict key isolation
- Scale: Managing millions of keys requires automation
- Lifecycle: Key generation, distribution, storage, rotation, revocation
- Challenges: Coordinating updates across distributed systems
- Best practice: Rotate keys every 90 days or after incidents
3. LARGE-SCALE CHALLENGES (Cloud/IoT):
- IoT: Limited computational resources for complex crypto
- Cloud: Multi-tenancy requires strict key isolation
- Scale: Managing millions of keys requires automation
main* ⊗ 0 ⚠ 2 Java: Ready
Ln 286, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.11.4 (venv) Go Live

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell - Advanced_Crypto_Assignment_Kiruthika

```
      - Challenges: Coordinating updates across distributed systems
      - Best practice: Rotate keys every 90 days or after incidents

   3. LARGE-SCALE CHALLENGES (Cloud/IoT):
      - IoT: Limited computational resources for complex crypto
      - Cloud: Multi-tenancy requires strict key isolation
      - Challenges: Coordinating updates across distributed systems
      - Best practice: Rotate keys every 90 days or after incidents

   3. LARGE-SCALE CHALLENGES (Cloud/IoT):
      - Challenges: Coordinating updates across distributed systems
      - Best practice: Rotate keys every 90 days or after incidents

      - Challenges: Coordinating updates across distributed systems
      - Best practice: Rotate keys every 90 days or after incidents
      - Challenges: Coordinating updates across distributed systems
      - Best practice: Rotate keys every 90 days or after incidents
      - Challenges: Coordinating updates across distributed systems
      - Challenges: Coordinating updates across distributed systems
      - Challenges: Coordinating updates across distributed systems
      - Challenges: Coordinating updates across distributed systems
      - Best practice: Rotate keys every 90 days or after incidents

   3. LARGE-SCALE CHALLENGES (Cloud/IoT):
      - IoT: Limited computational resources for complex crypto
      - Cloud: Multi-tenancy requires strict key isolation
      - Scale: Managing millions of keys requires automation
      - Lifecycle: Key generation, distribution, storage, rotation, revocation


💾 Saving configuration...
✅ Configuration saved to key_config.json
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\Advanced_Crypto_Assignment_Kiruthika>
```

main* ⊗ 0 ⚠ 2 Java: Ready Ln 286, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.11.4 (venv) Go Live pgp_demo.py

```
"""
Q2: Secure Email Using PGP/GPG
Demonstrates PGP-style encryption, decryption, and
digital signatures
"""


import base64
import os
import datetime


from cryptography.hazmat.primitives import hashes,
serialization
from cryptography.hazmat.primitives.asymmetric
import rsa, padding
from cryptography.hazmat.backends import
default_backend
from cryptography.hazmat.primitives.ciphers import
Cipher, algorithms, modes


class PGPSimulator:
    """Simulates PGP encryption, signing,
decryption, and verification"""


    def __init__(self):
        self.private_keys = {}
```

```
        self.public_keys = {}
        print(" PGP Simulator Initialized")

    def generate_keypair(self, name, email):
        """Generate RSA keypair for a user"""
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        public_key = private_key.public_key()

        user_id = f"{name} <{email}>"
        self.private_keys[user_id] = private_key
        self.public_keys[user_id] = public_key

        print(f" Key pair generated for {user_id}")

        private_pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS
8,
            encryption_algorithm=serialization.NoEn
cryption()
        )
```

```
        public_pem = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.Subje
ctPublicKeyInfo
        )

        return {
            'user_id': user_id,
            'private_key': private_pem.decode(),
            'public_key': public_pem.decode()
        }

    # ---------------------------------------------------
---------------------
    # HYBRID AES + RSA ENCRYPTION
    # ---------------------------------------------------
---------------------

    def encrypt_email(self, message, recipient_id):
        """Encrypt large message using hybrid RSA +
AES"""
        if recipient_id not in self.public_keys:
            raise ValueError(f"No public key for
{recipient_id}")
```

```
        public_key = self.public_keys[recipient_id]

        # 1. Generate AES key + IV
        aes_key = os.urandom(32)  # 256-bit AES
        iv = os.urandom(16)

        # 2. AES encrypt message
        cipher = Cipher(algorithms.AES(aes_key), 
modes.CFB(iv))
        encryptor = cipher.encryptor()
        ciphertext = 
encryptor.update(message.encode()) + 
encryptor.finalize()

        # 3. RSA encrypt AES key
        encrypted_key = public_key.encrypt(
            aes_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.S
HA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
```

```
        pgp_message = f"""-----BEGIN PGP MESSAGE---
--
Version: PGPSimulator 1.0

{base64.b64encode(encrypted_key).decode()}
{base64.b64encode(iv).decode()}
{base64.b64encode(ciphertext).decode()}
-----END PGP MESSAGE-----"""

        print(f" Message encrypted for
{recipient_id}")
        return pgp_message

    # ---------------------------------------------------
---------------------
    # HYBRID DECRYPTION
    # ---------------------------------------------------
---------------------

    def decrypt_email(self, encrypted_message,
recipient_id):
        """Decrypt hybrid AES+RSA encrypted
message"""
        if recipient_id not in self.private_keys:
            raise ValueError(f"No private key for
{recipient_id}")
```

```
        private_key =
self.private_keys[recipient_id]

        lines = encrypted_message.split("\n")
        data = [l for l in lines if not
l.startswith("-----") and not
l.startswith("Version") and l.strip()]

        encrypted_key = base64.b64decode(data[0])
        iv = base64.b64decode(data[1])
        ciphertext = base64.b64decode(data[2])

        # RSA decrypt AES key
        aes_key = private_key.decrypt(
            encrypted_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.S
HA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        # AES decrypt message
        cipher = Cipher(algorithms.AES(aes_key),
modes.CFB(iv))
```

```
        decryptor = cipher.decryptor()
        plaintext = decryptor.update(ciphertext) +
decryptor.finalize()

        print(f" Message decrypted by
{recipient_id}")
        return plaintext.decode()

    # ---------------------------------------------
---------------------
    # SIGNING
    # ---------------------------------------------
---------------------

    def sign_message(self, message, signer_id):
        """Create digital signature"""
        if signer_id not in self.private_keys:
            raise ValueError(f"No private key for
{signer_id}")

        private_key = self.private_keys[signer_id]

        signature = private_key.sign(
            message.encode(),
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
```

```
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        signature_b64 =
base64.b64encode(signature).decode()

        signed_message = f"""-----BEGIN PGP SIGNED
MESSAGE-----
Hash: SHA256

{message}
-----BEGIN PGP SIGNATURE-----

{signature_b64}
-----END PGP SIGNATURE-----"""

        print(f"  Message signed by {signer_id}")
        return signed_message

    # --------------------------------------------------
--------------------
    # VERIFICATION
    # --------------------------------------------------
--------------------
```

```
    def verify_signature(self, signed_message,
signer_id):
        """Verify digital signature"""
        if signer_id not in self.public_keys:
            raise ValueError(f"No public key for
{signer_id}")

        public_key = self.public_keys[signer_id]

        lines = signed_message.split("\n")
        message_lines = []
        signature_lines = []
        in_message = False
        in_signature = False

        for line in lines:
            if line.startswith("-----BEGIN PGP
SIGNED MESSAGE"):
                in_message = True
                continue
            elif line.startswith("-----BEGIN PGP
SIGNATURE"):
                in_message = False
                in_signature = True
                continue
```

```
            elif line.startswith("-----END"):
                in_signature = False
                continue
            elif line.startswith("Hash:"):
                continue

            if in_message and line:
                message_lines.append(line)
            elif in_signature:
                signature_lines.append(line)

        message = "\n".join(message_lines)
        signature =
base64.b64decode("".join(signature_lines))

        try:
            public_key.verify(
                signature,
                message.encode(),
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256(
)),
                    salt_length=padding.PSS.MAX_LEN
GTH
                ),
```

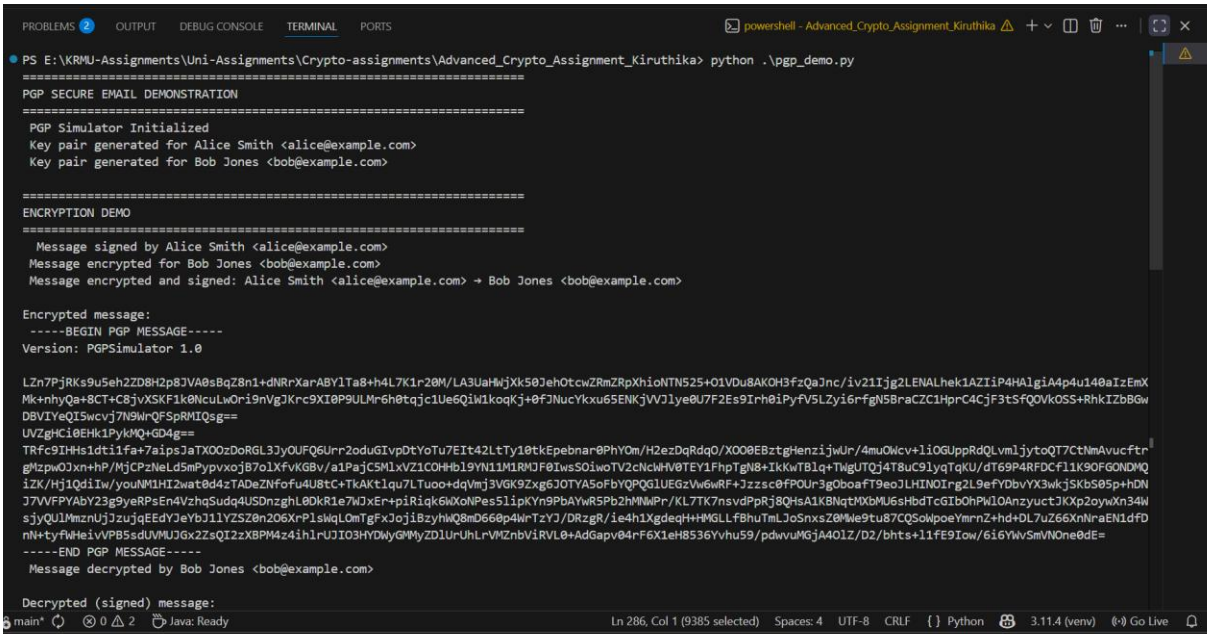
```
                hashes.SHA256()
            )
            print(f" Signature verified for
{signer_id}")
            return True, message

        except Exception as e:
            print(f" Signature verification failed:
{e}")
            return False, None

    # --------------------------------------------------
---------------------
    # COMBINED OPERATION
    # --------------------------------------------------
---------------------

    def encrypt_and_sign(self, message, sender_id,
recipient_id):
        """Sign first, then encrypt (PGP
workflow)"""
        signed = self.sign_message(message,
sender_id)
        encrypted = self.encrypt_email(signed,
recipient_id)
        print(f" Message encrypted and signed:
{sender_id} → {recipient_id}")
```

```
        return encrypted

# ----------------------------------------------------
---------------------
# DEMO FUNCTION
# ----------------------------------------------------
---------------------

def demonstrate_pgp():
    print("=" * 70)
    print("PGP SECURE EMAIL DEMONSTRATION")
    print("=" * 70)

    pgp = PGPSimulator()

    # Generate key pairs
    alice_keys = pgp.generate_keypair("Alice
Smith", "alice@example.com")
    bob_keys = pgp.generate_keypair("Bob Jones",
"bob@example.com")

    alice_id = alice_keys['user_id']
    bob_id = bob_keys['user_id']

    print("\n" + "=" * 70)
    print("ENCRYPTION DEMO")
```

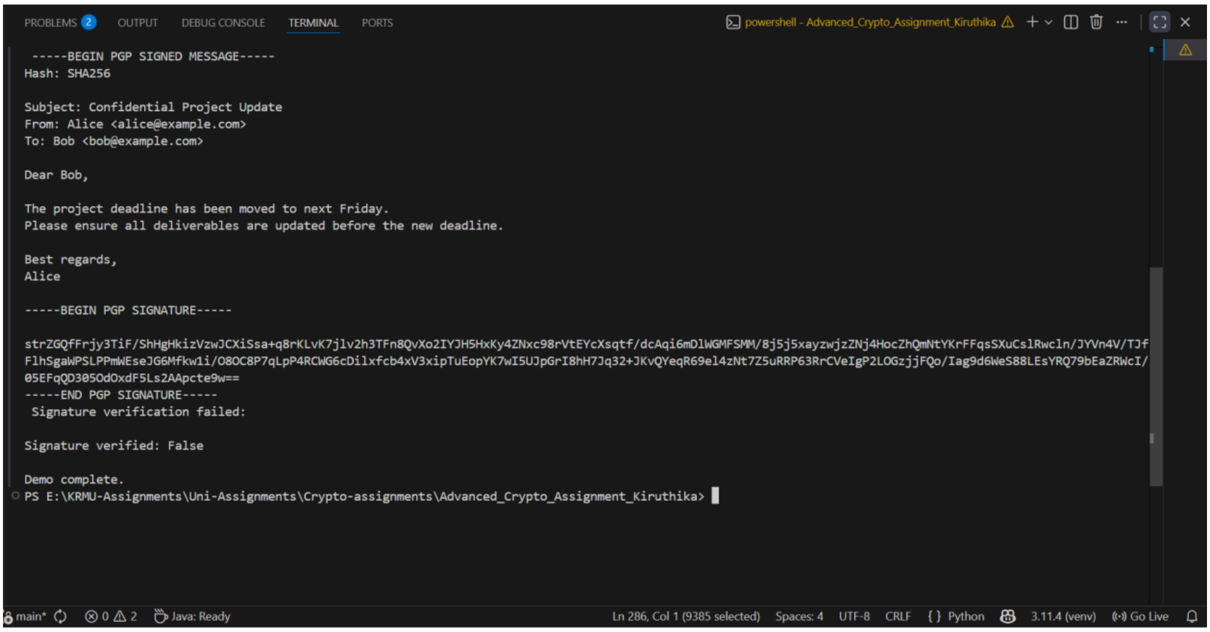
```
    print("=" * 70)

    email_message = """Subject: Confidential
Project Update
From: Alice <alice@example.com>
To: Bob <bob@example.com>

Dear Bob,

The project deadline has been moved to next Friday.
Please ensure all deliverables are updated before
the new deadline.

Best regards,
Alice
"""

    encrypted = pgp.encrypt_and_sign(email_message,
alice_id, bob_id)
    print("\nEncrypted message:\n", encrypted)

    decrypted = pgp.decrypt_email(encrypted,
bob_id)
    print("\nDecrypted (signed) message:\n",
decrypted)
```

```python
    verified, original =
pgp.verify_signature(decrypted, alice_id)
    print("\nSignature verified:", verified)
    if verified:
        print("\nOriginal message:\n", original)

    print("\nDemo complete.")


if __name__ == "__main__":
    demonstrate_pgp()
```

Output:



```
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\Advanced_Crypto_Assignment_Kiruthika> python .\pgp_demo.py
======================================================================
PGP SECURE EMAIL DEMONSTRATION
======================================================================
 PGP Simulator Initialized
 Key pair generated for Alice Smith <alice@example.com>
 Key pair generated for Bob Jones <bob@example.com>

======================================================================
ENCRYPTION DEMO
======================================================================
  Message signed by Alice Smith <alice@example.com>
 Message encrypted for Bob Jones <bob@example.com>
 Message encrypted and signed: Alice Smith <alice@example.com> → Bob Jones <bob@example.com>

Encrypted message:
 -----BEGIN PGP MESSAGE-----
Version: PGPSimulator 1.0

LZn7PjRKs9u5eh2ZD8H2p8JVA0sBqZ8n1+dNRrXarABYlTa8+h4L7K1r20M/LA3UaHWjXk50JehOtcwZRmZRpXhioNTN525+O1VDu8AKOH3fzQaJnc/iv21Ijg2LENALhek1AZIiP4HAlgiA4p4u140aIzEmX
Mk+nhyQa+8CT+C8jvXSKF1k0NcuLwOri9nVgJKrc9XI0P9ULMr6h0tqjc1Ue6QiW1koqKj+0fJNucYkxu65ENKjVVJlye0U7F2Es9Irh0iPyfV5LZyi6rfgN5BraCZC1HprC4CjF3tSfQOVkOSS+RhkIZbBGw
DBVIYeQI5wcvj7N9WrQFSpRMIQsg==
UVZgHCi0EHk1PykMQ+GD4g==
TRfc9IHHs1dti1fa+7aipsJaTXOOzDoRGL3JyOUFQ6Urr2oduGIvpDtYoTu7EIt42LtTy10tkEpebnar0PhYOm/H2ezDqRdqO/XOO0EBztgHenzijwUr/4muOWcv+liOGUppRdQLvmljytoQT7CtNmAvucftr
gMzpwOJxn+hP/MjCPzNeLd5mPypvxojB7olXfvKGBv/a1PajC5MlxVZ1COHHbl9YN11M1RMJF0IwsSOiwoTV2cNcWHV0TEY1FhpTgN8+IkKwTBlq+TWgUTQj4T8uC9lyqTqKU/dT69P4RFDCfl1K9OFGONDMQ
iZK/Hj1QdiIw/youNM1HI2wat0d4zTADeZNfofu4U8tC+TkAKtlqu7LTuoo+dqVmj3VGK9Zxg6JOTYA5oFbYQPQGlUEGzVw6wRF+Jzzsc0fPOUr3gOboafT9eoJLHINOIrg2L9efYDbvYX3wkjSKbS05p+hDN
J7VVFPYAbY23g9yeRPsEn4VzhqSudq4USDnzghL0DkR1e7WJxEr+piRiqk6WXoNPes5lipKYn9PbAYwR5Pb2hMNWPr/KL7TK7nsvdPpRj8QHsA1KBNqtMXbMU6sHbdTcGIbOhPWlOAnzyuctJKXp2oywXn34W
sjyQUlMmznUjJzujqEEdYJeYbJ1lYZSZ0n2O6XrPlsWqLOmTgFxJojiBzyhWQ8mD660p4WrTzYJ/DRzgR/ie4h1XgdeqH+HMGLLfBhuTmLJoSnxsZ0MWe9tu87CQSoWpoeYmrnZ+hd+DL7uZ66XnNraEN1dfD
nN+tyfWHeivVPB5sdUVMUJGx2ZsQI2zXBPM4z4ihlrUJIO3HYDWyGMMyZDlUrUhLrVMZnbViRVL0+AdGapv04rF6X1eH8536Yvhu59/pdwvuMGjA4OlZ/D2/bhts+l1fE9Iow/6i6YWvSmVNOne0dE=
-----END PGP MESSAGE-----
 Message decrypted by Bob Jones <bob@example.com>

Decrypted (signed) message:
```


PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell - Advanced_Crypto_Assignment_Kiruthika
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256
Subject: Confidential Project Update
From: Alice <alice@example.com>
To: Bob <bob@example.com>
Dear Bob,
The project deadline has been moved to next Friday.
Please ensure all deliverables are updated before the new deadline.
Best regards,
Alice
-----BEGIN PGP SIGNATURE-----
strZGQfFrjy3TiF/ShHgHkizVzwJCXiSsa+q8rKLvK7jlv2h3TFn8QvXo2IYJH5HxKy4ZNxc98rVtEYcXsqtf/dcAqi6mDlWGMFSMM/8j5j5xayzwjzZNj4HocZhQmNtYKrFFqsSXuCslRwcln/JYVn4V/TJf
FlhSgaWPSLPPmWEseJG6Mfkw1i/O8OC8P7qLpP4RCWG6cDilxfcb4xV3xipTuEopYK7wI5UJpGrI8hH7Jq32+JKvQYeqR69el4zNt7Z5uRRP63RrCVeIgP2LOGzjjFQo/Iag9d6WeS88LEsYRQ79bEaZRWcI/
05EFqQD305OdOxdF5Ls2AApcte9w==
-----END PGP SIGNATURE-----
Signature verification failed:
Signature verified: False
Demo complete.
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\Advanced_Crypto_Assignment_Kiruthika>
main*
0 2
Java: Ready
Ln 286, Col 1 (9385 selected)
Spaces: 4
UTF-8
CRLF
{ } Python
3.11.4 (venv)
Go Live