

Cryptography

Lab Assignment – 4

Github:

https://github.com/mayank24000/University_Assignments/tree/main/Crypto_Assignments/Advanced_Crypto_Assignment_Mayank

Q1: Key Management and Distribution System

Simulates a Key Distribution Center (KDC) for symmetric key distribution.

Code:

```
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 symmetric key distribution"""
```

```

def __init__(self):
    self.master_keys = {} # Store master keys for each user
    self.session_keys = {} # Track active session keys
    self.key_escrow = {} # Escrowed keys for recovery

def register_user(self, user_id):
    """Register a user and generate their master key"""
    master_key = os.urandom(32) # 256-bit key
    self.master_keys[user_id] = master_key
    # Key escrow - store encrypted backup
    self.key_escrow[user_id] = {
        'key': base64.b64encode(master_key).decode(),
        'timestamp': time.time(),
        'rotations': 0
    }
    print(f"✓ User '{user_id}' registered with KDC")
    return master_key

def request_session_key(self, user_a, user_b):
    """
    Simulate Needham-Schroeder protocol:
    1. A requests session key for communication with B
    2. KDC generates session key
    3. KDC encrypts session key with A's and B's master keys
    """
    if user_a not in self.master_keys or user_b not in self.master_keys:

```

```

        raise ValueError("Both users must be registered")

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

    # Encrypt session key for User A
    ticket_a = self._encrypt_ticket(session_key, self.master_keys[user_a],
user_b)

    # Encrypt session key for User B (ticket)
    ticket_b = self._encrypt_ticket(session_key, self.master_keys[user_b],
user_a)

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

    print(f"✓ Session key generated for {user_a} <-> {user_b}")
    return ticket_a, ticket_b, session_id

def _encrypt_ticket(self, session_key, master_key, peer_id):
    """Encrypt session key with user's master key"""
    iv = os.urandom(16)
    cipher = Cipher(

```

```

        algorithms.AES(master_key),
        modes.CBC(iv),
        backend=default_backend()
    )

    encryptor = cipher.encryptor()

    # Pad session key to block size
    padded_key = session_key + b'\x00' * (16 - len(session_key) % 16)
    encrypted = encryptor.update(padded_key) + encryptor.finalize()

    return {
        'iv': base64.b64encode(iv).decode(),
        'encrypted_key': base64.b64encode(encrypted).decode(),
        'peer': peer_id
    }

```

```

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

    old_key = self.master_keys[user_id]
    new_key = os.urandom(32)

    # Update master key
    self.master_keys[user_id] = new_key

```

```

# Update escrow

self.key_escrow[user_id]['key'] = base64.b64encode(new_key).decode()

self.key_escrow[user_id]['rotations'] += 1

self.key_escrow[user_id]['timestamp'] = time.time()


    print(f'✓ Key rotated for user '{user_id}' (rotation
#{self.key_escrow[user_id]['rotations']})")

    return new_key

```

```

class AsymmetricKeyManager:

    """Manages asymmetric key pairs and distribution"""

    def __init__(self):

        self.public_keys = {}

        self.private_keys = {}


    def generate_keypair(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 private_key, public_key
```

```
def get_public_key(self, user_id):
```

```
    """Simulate public key directory lookup"""
```

```
    return self.public_keys.get(user_id)
```

```
def encrypt_with_public_key(self, message, recipient_id):
```

```
    """Encrypt message with recipient's public key"""
```

```
    public_key = self.get_public_key(recipient_id)
```

```
    if not public_key:
```

```
        raise ValueError(f"No public key for {recipient_id}")
```

```
    encrypted = public_key.encrypt(
```

```
        message.encode() if isinstance(message, str) else message,
```

```
        padding.OAEP(
```

```
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
```

```
            algorithm=hashes.SHA256(),
```

```
            label=None
```

```
        )
```

```
    )
```

```
    return encrypted
```

```

def demonstrate_key_distribution():
    """Demonstrate both symmetric and asymmetric key distribution"""

    print("=" * 60)
    print("KEY DISTRIBUTION DEMONSTRATION")
    print("=" * 60)

    # Symmetric Key Distribution via KDC
    print("\n[1] SYMMETRIC KEY DISTRIBUTION (KDC Model)")
    print("-" * 60)
    kdc = KeyDistributionCenter()

    # Register users
    kdc.register_user("Alice")
    kdc.register_user("Bob")
    kdc.register_user("Charlie")

    # Request session key
    ticket_a, ticket_b, session_id = kdc.request_session_key("Alice", "Bob")
    print(f"Session ID: {session_id}")

    # Demonstrate key rotation
    print("\n[2] KEY ROTATION")
    print("-" * 60)

```

```

kdc.rotate_key("Alice")
kdc.rotate_key("Alice") # Rotate again


# Asymmetric Key Distribution

print("\n[3] ASYMMETRIC KEY DISTRIBUTION (Public Key
Infrastructure)")

print("-" * 60)

key_manager = AsymmetricKeyManager()


key_manager.generate_keypair("Alice")
key_manager.generate_keypair("Bob")


# Demonstrate encryption

message = "Secret message for Bob"

encrypted = key_manager.encrypt_with_public_key(message, "Bob")

print(f'✓ Message encrypted for Bob (length: {len(encrypted)} bytes)')


# Key Escrow Report

print("\n[4] KEY ESCROW STATUS")

print("-" * 60)

for user, escrow_data in kdc.key_escrow.items():

    print(f"User: {user}")

    print(f" Rotations: {escrow_data['rotations']}")

    print(f" Last Updated: {time.ctime(escrow_data['timestamp'])}")


# Challenges Discussion

```



```
print("\n[5] CHALLENGES IN LARGE-SCALE ENVIRONMENTS")
print("-" * 60)
challenges = {
    "Cloud": [
        "Multi-tenancy key isolation",
        "Geographic key distribution",
        "Compliance with regional regulations (GDPR, etc.)",
        "Key synchronization across data centers"
    ],
    "IoT": [
        "Limited computational resources for key operations",
        "Massive scale (billions of devices)",
        "Secure key storage in constrained devices",
        "Battery-efficient key management protocols"
    ],
    "General": [
        "Key rotation without service disruption",
        "Secure key escrow and recovery mechanisms",
        "Quantum-resistant key exchange migration",
        "Insider threat mitigation"
    ]
}
```

```
for category, items in challenges.items():
    print(f"\n{category}:")
    for item in items:
```

```
print(f' • {item}')
```

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

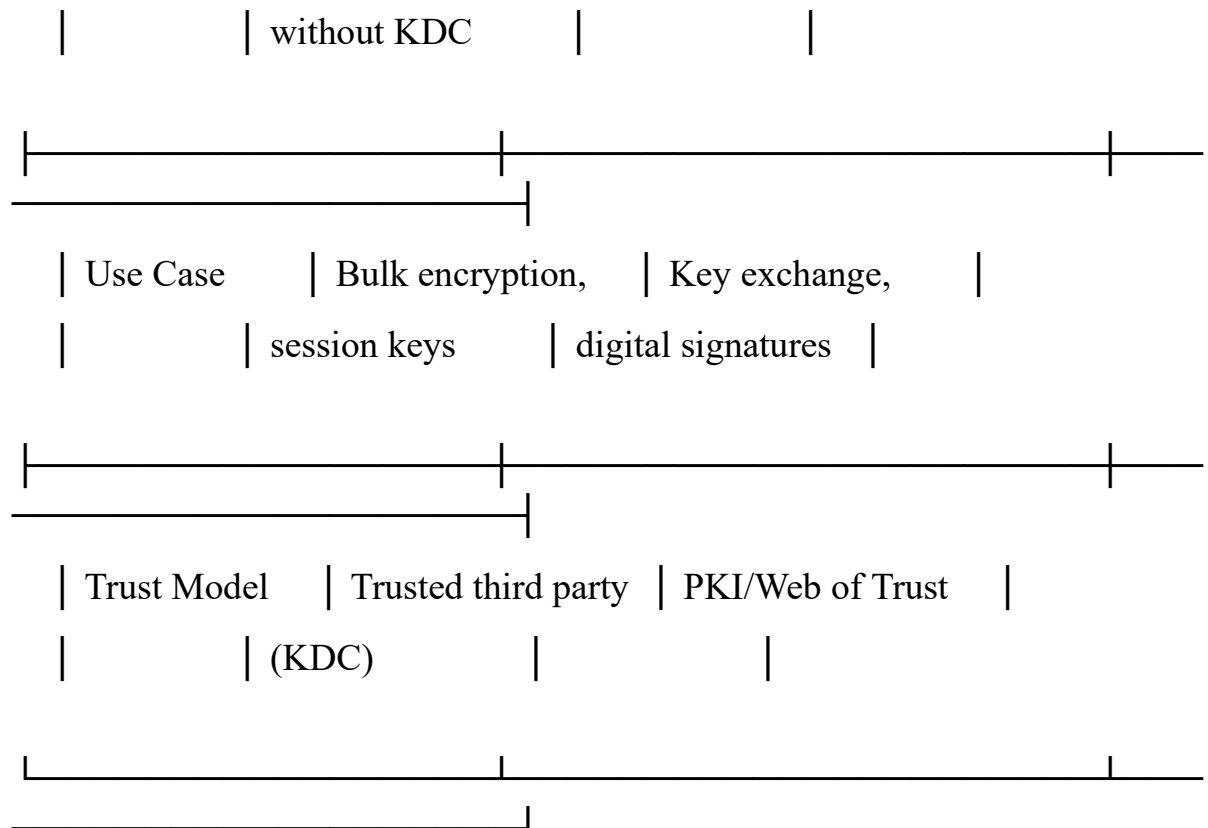
```
print("\n" + "=" * 60)
```

```
print("COMPARISON: Symmetric vs Asymmetric Key Distribution")
```

```
print("=" * 60)
```

```
comparison = ""
```

Symmetric Key Distribution		Asymmetric Key Distribution	
Aspect	Symmetric	Asymmetric	
Key Exchange	Requires secure channel (KDC)	Public key can be openly distributed	
Speed	Fast (AES, 3DES)	Slow (RSA, ECC)	
Key Management	$O(n^2)$ keys needed	$O(n)$ keys needed	



"""

print(comparison)

Output:

```

• (.venv) PS D:\VsCode\University_Assignments\Crypto_Assignments\Advanced_Crypto_Assignment_Mayank> & D:/V
vanced_Crypto_Assignment_Mayank/key_distribution.py
=====
KEY DISTRIBUTION DEMONSTRATION
=====

[1] SYMMETRIC KEY DISTRIBUTION (KDC Model)
-----
✓ User 'Alice' registered with KDC
✓ User 'Bob' registered with KDC
✓ User 'Charlie' registered with KDC
• ✓ Session key generated for Alice <-> Bob
    Session ID: Alice_Bob_1763481516

[2] KEY ROTATION
-----
✓ Key rotated for user 'Alice' (rotation #1)
✓ Key rotated for user 'Alice' (rotation #2)

[3] ASYMMETRIC KEY DISTRIBUTION (Public Key Infrastructure)
-----
✓ RSA key pair generated for 'Alice'
✓ RSA key pair generated for 'Bob'
✓ Message encrypted for Bob (length: 256 bytes)

[4] KEY ESCROW STATUS
-----
User: Alice
    Rotations: 2
    Last Updated: Tue Nov 18 21:28:36 2025
User: Bob
    Rotations: 0
    Last Updated: Tue Nov 18 21:28:36 2025
User: Charlie
    Rotations: 0
    Last Updated: Tue Nov 18 21:28:36 2025

[5] CHALLENGES IN LARGE-SCALE ENVIRONMENTS
-----

Cloud:
  • Multi-tenancy key isolation
  • Geographic key distribution
  • Compliance with regional regulations (GDPR, etc.)
  • Key synchronization across data centers

IoT:
  • Limited computational resources for key operations
  • Massive scale (billions of devices)
  • Secure key storage in constrained devices

```

General:

- Key rotation without service disruption
- Secure key escrow and recovery mechanisms
- Quantum-resistant key exchange migration
- Insider threat mitigation

=====

COMPARISON: Symmetric vs Asymmetric Key Distribution

=====

Aspect	Symmetric	Asymmetric
Key Exchange	Requires secure channel (KDC)	Public key can be openly distributed
Speed	Fast (AES, 3DES)	Slow (RSA, ECC)
Key Management	$O(n^2)$ keys needed without KDC	$O(n)$ keys needed
Use Case	Bulk encryption, session keys	Key exchange, digital signatures
Trust Model	Trusted third party (KDC)	PKI/Web of Trust

○ (.venv) PS D:\VsCode\University_Assignments\Crypto_Assignments\Advanced_Crypto_Assignment_Mayank> █

Q-2 Secure Email Using PGP-like Implementation

Demonstrates Hybrid Encryption (RSA + AES) and Digital Signatures

Code:

```
import os
```

```
import base64
```

```
from datetime import datetime
```

```
from cryptography.hazmat.primitives import hashes, serialization
```

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding as  
asym_padding
```

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

```
from cryptography.hazmat.primitives import padding as sym_padding
```

```
from cryptography.hazmat.backends import default_backend
```

```
class SimplePGPDemo:
```

```
"""
```

PGP demonstration using pure Python cryptography library.

Implements 'Hybrid Encryption':

1. Message encrypted with symmetric key (AES)
2. Symmetric key encrypted with asymmetric public key (RSA)

```
"""
```

```
def __init__(self):
```

```
    self.private_keys = {}
```

```
    self.public_keys = {}
```

```
def generate_key_pair(self, user_id):
```

```
    """Generate RSA key pair for user"""
```

```
    print(f"Generating 2048-bit RSA keys for {user_id}...")
```

```
    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
```

```
# Save keys to files (simulation)
```

```
self._save_private_key(user_id, private_key)
```

```

self._save_public_key(user_id, public_key)

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

return private_key, public_key

def _save_private_key(self, user_id, private_key):
    """Save private key to PEM file"""
    pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )
    # Using sanitize filename to prevent errors
    filename = f"{user_id.replace('@','_').replace('.', '_')} _private.pem"

    # Ensure directory exists
    if not os.path.exists("emails"):
        os.makedirs("emails")

    filepath = os.path.join("emails", filename)
    with open(filepath, 'wb') as f:
        f.write(pem)

def _save_public_key(self, user_id, public_key):
    """Save public key to PEM file"""
    pem = public_key.public_bytes(

```

```

        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    filename = f'{user_id.replace('@','_').replace('.', '_')}_public.pem"

    # Ensure directory exists
    if not os.path.exists("emails"):
        os.makedirs("emails")

    filepath = os.path.join("emails", filename)
    with open(filepath, 'wb') as f:
        f.write(pem)

def encrypt_message(self, message, recipient_id):
    """Encrypt message for recipient using hybrid encryption"""
    if recipient_id not in self.public_keys:
        raise ValueError(f"Public key for {recipient_id} not found!")

    public_key = self.public_keys[recipient_id]

    # 1. Generate random AES key (Session Key) and IV
    aes_key = os.urandom(32) # 256-bit AES key
    iv = os.urandom(16)     # 128-bit IV

    # 2. Pad message to AES block size (PKCS7)
    padder = sym_padding.PKCS7(128).padder()

```



```
message_bytes = message.encode('utf-8')
padded_data = padder.update(message_bytes) + padder.finalize()
```

3. Encrypt message with AES (Symmetric)

```
cipher = Cipher(
    algorithms.AES(aes_key),
    modes.CBC(iv),
    backend=default_backend()
)
encryptor = cipher.encryptor()
encrypted_message = encryptor.update(padded_data) + encryptor.finalize()
```

4. Encrypt AES key with recipient's public key (Asymmetric/RSA)

```
encrypted_key = public_key.encrypt(
    aes_key,
    asym_padding.OAEP(
        mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
```

5. Combine: [Encrypted AES Key (256 bytes)] + [IV (16 bytes)] + [Encrypted Message]

```
combined = encrypted_key + iv + encrypted_message
encrypted_b64 = base64.b64encode(combined).decode('utf-8')
```

```
    print(f"✓ Message encrypted for {recipient_id} (Size: {len(encrypted_b64)} bytes)")
```

```
    return encrypted_b64
```

```
def decrypt_message(self, encrypted_message_b64, recipient_id):
```

```
    """Decrypt message using hybrid decryption"""
```

```
    if recipient_id not in self.private_keys:
```

```
        raise ValueError(f"Private key for {recipient_id} not found!")
```

```
    private_key = self.private_keys[recipient_id]
```

```
    try:
```

```
        combined = base64.b64decode(encrypted_message_b64)
```

```
        # Extract parts (Assuming 2048-bit RSA key = 256 bytes encrypted key)
```

```
        encrypted_key_len = 256
```

```
        iv_len = 16
```

```
        encrypted_key = combined[:encrypted_key_len]
```

```
        iv = combined[encrypted_key_len : encrypted_key_len + iv_len]
```

```
        encrypted_message = combined[encrypted_key_len + iv_len:]
```

```
        # 1. Decrypt AES key with recipient's private key (RSA)
```

```
        aes_key = private_key.decrypt(
```

```
            encrypted_key,
```

```

    asym_padding.OAEP(
        mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

```

2. Decrypt message with AES

```

cipher = Cipher(
    algorithms.AES(aes_key),
    modes.CBC(iv),
    backend=default_backend()
)

decryptor = cipher.decryptor()

padded_message = decryptor.update(encrypted_message) +
decryptor.finalize()

```

3. Remove PKCS7 padding

```

unpadder = sym_padding.PKCS7(128).unpadder()

decrypted_message_bytes = unpadder.update(padded_message) +
unpadder.finalize()

```

```

print(f"✓ Message decrypted by {recipient_id}")

return decrypted_message_bytes.decode('utf-8')

```

except Exception as e:

```

    print(f"Error decrypting: {e}")

```

```
return None
```

```
def sign_message(self, message, sender_id):  
    """Create digital signature"""  
    private_key = self.private_keys[sender_id]  
  
    signature = private_key.sign(  
        message.encode('utf-8'),  
        asym_padding.PSS(  
            mgf=asym_padding.MGF1(hashes.SHA256()),  
            salt_length=asym_padding.PSS.MAX_LENGTH  
        ),  
        hashes.SHA256()  
    )  
  
    signature_b64 = base64.b64encode(signature).decode('utf-8')  
    print(f"✓ Message signed by {sender_id}")  
    return signature_b64
```

```
def verify_signature(self, message, signature_b64, sender_id):  
    """Verify digital signature"""  
    public_key = self.public_keys[sender_id]  
    signature = base64.b64decode(signature_b64)  
  
    try:  
        public_key.verify(  

```

```

        signature,
        message.encode('utf-8'),
        asym_padding.PSS(
            mgf=asym_padding.MGF1(hashes.SHA256()),
            salt_length=asym_padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    print(f'✓ Signature verified from {sender_id}')

    return True
except Exception as e:
    print(f'✗ Signature verification failed: {e}')

    return False

```

```

def create_signed_email(self, message, sender_id, recipient_id):
    """Create encrypted and signed email"""

    # 1. Sign the cleartext message
    signature = self.sign_message(message, sender_id)

    # 2. Combine message and signature
    # In real PGP, this is more complex (compression, packets),
    # but for this demo, we append the signature.
    signed_payload = f'{message}\n\n---SIGNATURE---\n{signature}'

    # 3. Encrypt the combined payload for the recipient
    encrypted = self.encrypt_message(signed_payload, recipient_id)

```

```
return encrypted
```

```
def read_signed_email(self, encrypted_email, recipient_id, sender_id):
```

```
    """Decrypt and verify signed email"""
```

```
    # 1. Decrypt
```

```
    decrypted_payload = self.decrypt_message(encrypted_email, recipient_id)
```

```
    if not decrypted_payload:
```

```
        return None
```

```
    # 2. Split message and signature
```

```
    separator = "\n\n---SIGNATURE---\n"
```

```
    if separator not in decrypted_payload:
```

```
        print("X Invalid email format: Signature block missing")
```

```
        return {'message': decrypted_payload, 'verified': False, 'sender':  
'Unknown'}
```

```
    message, signature = decrypted_payload.split(separator)
```

```
    # 3. Verify signature using the extracted message content
```

```
    verified = self.verify_signature(message, signature, sender_id)
```

```
    return {
```

```
        'message': message,
```

```
        'verified': verified,
```

```
        'sender': sender_id
    }
```

```
def demonstrate_pgp():
```

```
    """Demonstrate PGP email encryption and signing"""
```

```
    print("=" * 70)
```

```
    print("PGP EMAIL DEMONSTRATION")
```

```
    print("=" * 70)
```

```
    pgp = SimplePGPDemo()
```

```
    # Ensure emails directory exists
```

```
    if not os.path.exists("emails"):
```

```
        os.makedirs("emails")
```

```
    # Generate key pairs
```

```
    print("\n[1] KEY GENERATION")
```

```
    print("-" * 70)
```

```
    pgp.generate_key_pair("alice@example.com")
```

```
    pgp.generate_key_pair("bob@example.com")
```

```
    # Create email message
```

```
    print("\n[2] COMPOSING EMAIL")
```

```
    print("-" * 70)
```

```
email_message = """From: alice@example.com
To: bob@example.com
Subject: Confidential Project Update
Date: {}
```

Dear Bob,

This is a confidential message about our secret project.
The launch is scheduled for next month.

Best regards,

Alice""".format(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))

```
print(email_message)
```

```
# Create signed and encrypted email
```

```
print("\n[3] SIGNING AND ENCRYPTING")
```

```
print("-" * 70)
```

```
encrypted_email = pgp.create_signed_email(
```

```
    email_message,
```

```
    "alice@example.com",
```

```
    "bob@example.com"
```

```
)
```

```
# Save encrypted email
```

```
file_path = os.path.join("emails", "signed_email.asc")
```



```

with open(file_path, "w") as f:
    f.write("-----BEGIN PGP MESSAGE-----\n\n")
    f.write(encrypted_email)
    f.write("\n-----END PGP MESSAGE-----\n")

print(f' Encrypted email saved to: {file_path}')

# Bob receives and decrypts
print("\n[4] DECRYPTING AND VERIFYING")
print("-" * 70)

# Simulating reading from file
with open(file_path, "r") as f:
    content = f.read()
    # simplistic parsing for demo
    lines = content.strip().split('\n')
    # Removing header/footer roughly
    clean_encrypted = lines[2] if len(lines) > 3 else encrypted_email

result = pgp.read_signed_email(
    encrypted_email, # Passing the raw base64 string
    "bob@example.com",
    "alice@example.com"
)

if result and result['verified']:

```

```

print("\n✓ Email successfully decrypted and verified!")

print("\nDecrypted message:")

print("-" * 70)

print(result['message'])

else:

    print("\nX Failed to decrypt or verify.")

```

```

# Comparison

```

```

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

print("S/MIME vs PGP COMPARISON")

print("=" * 70)

```

```

comparison = """

```

S/MIME vs PGP COMPARISON			
Feature	S/MIME	PGP	
Trust Model	Hierarchical PKI (Certificate Authority)	Web of Trust (Decentralized)	
Key Distribution	X.509 certificates from trusted CAs	Public keyservers (keys.openpgp.org)	

Integration	Built into email clients (Outlook, iOS)	Requires plugins (Thunderbird, Mailvelo)
Cost	Requires paid CA cert	Free and open source
Revocation	CRL and OCSP	Key revocation certs
Standards	IETF RFC 8551	OpenPGP RFC 4880
Best For	Corporate email	Privacy-focused users
	Enterprise use	Personal communication

```
"""
```

```
print(comparison)
```

```
if __name__ == "__main__":
```

```
    demonstrate_pgp()
```

Output:

```
(.venv) PS D:\VsCode\University_Assignments\Crypto_Assignments\Advanced_Crypto_Assignment_Mayank> & D:/VsCode/University_Assignments/.venv/Scripts/python.exe d:/VsCode/University_Assignments/Crypto_Assignments/Advanced_Crypto_Assignment_Mayank/pgp_demo.py

=====
PGP EMAIL DEMONSTRATION
=====

[1] KEY GENERATION
-----
Generating 2048-bit RSA keys for alice@example.com...
✓ Key pair generated for alice@example.com
Generating 2048-bit RSA keys for bob@example.com...
✓ Key pair generated for bob@example.com

[2] COMPOSING EMAIL
-----
From: alice@example.com
To: bob@example.com
Subject: Confidential Project Update
Date: 2025-11-18 21:32:00

Dear Bob,

This is a confidential message about our secret project.
The launch is scheduled for next month.

Best regards,
Alice

[3] SIGNING AND ENCRYPTING
-----
✓ Message signed by alice@example.com
✓ Message encrypted for bob@example.com (Size: 1176 bytes)
  Encrypted email saved to: emails\signed_email.asc

[4] DECRYPTING AND VERIFYING
-----
✓ Message decrypted by bob@example.com
✓ Signature verified from alice@example.com

✓ Email successfully decrypted and verified!
```

```
✓ Email successfully decrypted and verified!

Decrypted message:
-----
From: alice@example.com
To: bob@example.com
Subject: Confidential Project Update
Date: 2025-11-18 21:32:00

Dear Bob,

This is a confidential message about our secret project.
The launch is scheduled for next month.

Best regards,
Alice

=====
S/MIME vs PGP COMPARISON
=====
```

Feature	S/MIME	PGP
Trust Model	Hierarchical PKI (Certificate Authority)	Web of Trust (Decentralized)
Key Distribution	X.509 certificates from trusted CAs	Public keyservers (keys.openpgp.org)
Integration	Built into email clients (Outlook, iOS)	Requires plugins (Thunderbird, Mailvelo)
Cost	Requires paid CA cert	Free and open source
Revocation	CRL and OCSP	Key revocation certs
Standards	IETF RFC 8551	OpenPGP RFC 4880

Q3: SSL/TLS SECURE COMMUNICATION PROTOCOL ANALYSIS

Solution :

=====

=====

Q3: SSL/TLS SECURE COMMUNICATION PROTOCOL ANALYSIS

=====

=====

OPTION A: SSL/TLS HTTPS CONNECTION ANALYSIS

1. PROTOCOL OVERVIEW

=====

=====

TLS (Transport Layer Security) is the successor to SSL and provides:

- Authentication: Verify server (and optionally client) identity
- Confidentiality: Encrypt data in transit
- Integrity: Detect message tampering

Current Version: TLS 1.3 (RFC 8446)

Legacy Versions: TLS 1.2, TLS 1.1, TLS 1.0, SSL 3.0 (deprecated)

2. TLS HANDSHAKE PROCESS (TLS 1.2)

=====

=====

Step-by-Step Breakdown:

[CLIENT]

[SERVER]

1. ClientHello -->

- TLS version: 1.2
- Cipher suites supported
- Random number (Client Random)
- Session ID
- Supported extensions (SNI, ALPN)

<-- 2. ServerHello

- Selected cipher suite

- Random (Server Random)
- Session ID

<-- 3. Certificate

- Server's X.509 cert
- Certificate chain

<-- 4. ServerKeyExchange

- DH parameters (if using DHE)

<-- 5. ServerHelloDone

6. ClientKeyExchange -->

- Pre-master secret (encrypted with server's public key)

7. ChangeCipherSpec -->

- Switch to encrypted communication

8. Finished -->

- Encrypted handshake verification

<-- 9. ChangeCipherSpec

<-- 10. Finished

[ENCRYPTED APPLICATION DATA EXCHANGE]

3. TLS 1.3 IMPROVEMENTS

Reduced Handshake (1-RTT):

- Combined messages for faster connection
- Forward secrecy by default (ephemeral keys only)
- Removed weak cipher suites (RC4, DES, 3DES, MD5, SHA-1)

Supported Cipher Suites (TLS 1.3):

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

4. CERTIFICATE VALIDATION PROCESS

Certificate Chain Verification:

[End-Entity Certificate]

↓ (signed by)

[Intermediate CA Certificate]

↓ (signed by)

[Root CA Certificate] (in browser trust store)

Validation Steps:

1. Check certificate validity period (Not Before/Not After)
2. Verify certificate signature using issuer's public key
3. Check certificate revocation status (CRL or OCSP)
4. Validate certificate purpose (Extended Key Usage)
5. Verify hostname matches (Subject Alternative Name)
6. Ensure root CA is in trusted store

Example Certificate Fields:

Subject: CN=example.com, O=Example Inc, C=US

Issuer: CN=DigiCert TLS RSA SHA256 2020 CA1

Validity:

Not Before: Jan 1 00:00:00 2024 GMT

Not After : Jan 1 23:59:59 2025 GMT

Subject Alternative Names:

DNS:example.com

DNS:*.example.com

Public Key: RSA 2048 bits

Signature Algorithm: sha256WithRSAEncryption

5. SYMMETRIC KEY NEGOTIATION

=====

=====

Key Derivation Process:

Client Random (32 bytes)

+

Server Random (32 bytes)

+

Pre-Master Secret (48 bytes, encrypted with server's public key)

↓

[PRF - Pseudorandom Function with HMAC]

↓

Master Secret (48 bytes)

↓

[Key Expansion]

↓

- |— Client Write MAC Key
- |— Server Write MAC Key
- |— Client Write Encryption Key
- |— Server Write Encryption Key
- |— Client Write IV
- |— Server Write IV

Example Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

- ECDHE: Elliptic Curve Diffie-Hellman Ephemeral (key exchange)
- RSA: Authentication algorithm
- AES_128_GCM: Encryption (AES 128-bit with Galois/Counter Mode)

- SHA256: Message authentication

6. BROWSER DEVELOPER TOOLS ANALYSIS

How to Inspect HTTPS Connection:

Chrome DevTools:

1. Open DevTools (F12)
2. Navigate to Security tab
3. Refresh the page
4. Observe:
 - Connection status
 - Certificate details
 - TLS version
 - Cipher suite

Example Output:

Connection: Secure (TLS 1.3)

Cipher Suite: TLS_AES_128_GCM_SHA256

Key Exchange: X25519

Certificate: Valid (DigiCert)

Certificate Transparency: Compliant

Connection: Secure (TLS 1.3)

Cipher Suite: TLS_AES_128_GCM_SHA256

Key Exchange: X25519

Certificate: Valid (DigiCert)

Certificate Transparency: Compliant

text

Wireshark Capture Analysis:

Filter: ssl or tls

Packet Breakdown:

Client Hello (TLS 1.2)

SNI: www.example.com

Cipher Suites: 15 suites

Server Hello

Version: TLS 1.2

Cipher: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

Certificate

Length: 4321 bytes

Certificate Count: 3

Server Key Exchange

Curve: secp256r1

Public Key: 65 bytes

text

7. COMMON SSL/TLS ATTACKS AND MITIGATIONS

Attack	Description	Mitigation
POODLE	SSL 3.0 padding oracle	Disable SSL 3.0
BEAST	CBC mode vulnerability	Use TLS 1.2+ with AEAD
Heartbleed	OpenSSL buffer over-read	Update OpenSSL
CRIME/BREACH compression	Compression oracle	Disable TLS
Downgrade Attack TLS_FALLBACK_SCSV	Force use of weak protocols	
MITM (ARP spoofing)	Certificate substitution	Certificate pinning
Weak Cipher Suites only	RC4, DES, Export ciphers	Modern cipher suites

8. BEST PRACTICES

Server Configuration:

✓ Use TLS 1.3 or TLS 1.2 minimum

- ✓ Disable SSL 3.0, TLS 1.0, TLS 1.1
- ✓ Use strong cipher suites (AEAD only)
- ✓ Enable Perfect Forward Secrecy (ECDHE/DHE)
- ✓ Implement HSTS (HTTP Strict Transport Security)
- ✓ Use OCSP Stapling for faster validation
- ✓ Enable Certificate Transparency
- ✓ Regular certificate rotation

Client Configuration:

- ✓ Verify certificate chain
- ✓ Check certificate revocation (CRL/OCSP)
- ✓ Validate hostname
- ✓ Reject self-signed certificates in production
- ✓ Implement certificate pinning for critical apps

9. PERFORMANCE CONSIDERATIONS

=====

=====

Optimization Techniques:

- Session Resumption (Session IDs or Session Tickets)
- OCSP Stapling (reduce client-side validation delay)
- TLS 1.3 0-RTT (zero round-trip time for resumed connections)
- HTTP/2 or HTTP/3 multiplexing over single TLS connection

Benchmark (typical):

- TLS 1.2 Full Handshake: ~2 RTT + cert validation (100-300ms)
- TLS 1.3 Full Handshake: ~1 RTT (50-150ms)
- TLS 1.3 Resumed (0-RTT): ~0 RTT (minimal overhead)

10. TESTING TOOLS

=====

=====

1. OpenSSL:

```
$ openssl s_client -connect example.com:443 -tls1_3
```

2. SSL Labs (Qualys):

```
https://www.ssllabs.com/ssltest/
```

3. testssl.sh:

```
$ ./testssl.sh https://example.com
```

4. Nmap with SSL scripts:

```
$ nmap --script ssl-enum-ciphers -p 443 example.com
```

=====

=====

CONCLUSION

=====

TLS provides the cryptographic foundation for secure internet communication.

Proper implementation requires:

- Up-to-date protocol versions
- Strong cryptographic algorithms
- Proper certificate management
- Regular security audits
- Performance optimization

The evolution to TLS 1.3 has significantly improved both security and performance, removing legacy vulnerabilities while reducing connection latency.

Q4: Blockchain Cryptography Simulation

Demonstrates blockchain fundamentals including hashing, digital signatures, Merkle trees, and consensus mechanisms

Code:

```
import hashlib
import json
import time
from datetime import datetime
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"""
```

```
    def __init__(self, transactions):
```

```
        self.transactions = transactions
```

```
        self.tree = self.build_tree()
```

```
        self.root = self.tree[-1][0] if self.tree else None
```

```
    def hash_data(self, data):
```

```
        """Hash a single piece of data"""
```

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

```
    def build_tree(self):
```

```
        """Build Merkle tree from transactions"""
```

```
        if not self.transactions:
```

```
            return []
```

```
        # Level 0: Hash all transactions
```

```
        current_level = [self.hash_data(tx) for tx in self.transactions]
```

```
        tree = [current_level.copy()]
```

```
        # Build tree bottom-up
```

```
        while len(current_level) > 1:
```

```

next_level = []

# Process pairs
for i in range(0, len(current_level), 2):
    if i + 1 < len(current_level):
        combined = current_level[i] + current_level[i + 1]
    else:
        # Odd number: duplicate last hash
        combined = current_level[i] + current_level[i]

    next_level.append(self.hash_data(combined))

tree.append(next_level)
current_level = next_level

return tree

```

```

def get_proof(self, transaction_index):
    """Get Merkle proof for a transaction"""
    if transaction_index >= len(self.transactions):
        return None

```

```

proof = []
index = transaction_index

```

```

for level in self.tree[:-1]: # Exclude root

```

```

if index % 2 == 0:
    # Left node: need right sibling
    if index + 1 < len(level):
        proof.append(('R', level[index + 1]))
    else:
        proof.append(('R', level[index])) # Duplicate
    else:
        # Right node: need left sibling
        proof.append(('L', level[index - 1]))

    index = index // 2

```

```

return proof

```

```

def verify_proof(self, transaction, proof, root):
    """Verify Merkle proof"""
    current_hash = self.hash_data(transaction)

    for direction, sibling_hash in proof:
        if direction == 'L':
            combined = sibling_hash + current_hash
        else:
            combined = current_hash + sibling_hash
        current_hash = self.hash_data(combined)

    return current_hash == root

```

```
class Block:
```

```
    """Represents a single block in the blockchain"""
```

```
    def __init__(self, index, transactions, previous_hash, difficulty=4):
```

```
        self.index = index
```

```
        self.timestamp = time.time()
```

```
        self.transactions = transactions
```

```
        self.previous_hash = previous_hash
```

```
        self.difficulty = difficulty
```

```
        self.nonce = 0
```

```
        self.merkle_tree = MerkleTree([json.dumps(tx, sort_keys=True) for tx in  
transactions])
```

```
        self.merkle_root = self.merkle_tree.root
```

```
        self.hash = self.calculate_hash()
```

```
    def calculate_hash(self):
```

```
        """Calculate block hash"""
```

```
        block_data = {
```

```
            'index': self.index,
```

```
            'timestamp': self.timestamp,
```

```
            'transactions': self.transactions,
```

```
            'previous_hash': self.previous_hash,
```

```
            'merkle_root': self.merkle_root,
```

```
            'nonce': self.nonce
```

```
        }
```

```

block_string = json.dumps(block_data, sort_keys=True)
return hashlib.sha256(block_string.encode()).hexdigest()

def mine_block(self):
    """Proof of Work mining"""
    target = '0' * self.difficulty

    print(f'Mining block {self.index} (difficulty: {self.difficulty})...', end='',
flush=True)

    start_time = time.time()

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

    elapsed = time.time() - start_time
    print(f' Mined! (nonce: {self.nonce}, time: {elapsed:.2f}s)')
    return self.hash

def to_dict(self):
    """Convert block to dictionary"""
    return {
        'index': self.index,
        'timestamp': self.timestamp,
        'datetime': datetime.fromtimestamp(self.timestamp).strftime('%Y-%m-%d %H:%M:%S'),
        'transactions': self.transactions,
        'previous_hash': self.previous_hash,

```

```
    'merkle_root': self.merkle_root,  
    'nonce': self.nonce,  
    'hash': self.hash  
}
```

```
class DigitalWallet:
```

```
    """Manages cryptographic keys for blockchain transactions"""
```

```
    def __init__(self, owner):
```

```
        self.owner = owner
```

```
        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_pem = self.public_key.public_bytes(  
            encoding=serialization.Encoding.PEM,
```

```
            format=serialization.PublicFormat.SubjectPublicKeyInfo  
        )
```

```
        address_hash = hashlib.sha256(public_pem).hexdigest()
```

```
        return address_hash[:40] # Bitcoin-style address
```

```

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

def verify_transaction(self, transaction, signature, public_key):
    """Verify transaction signature"""
    try:
        transaction_string = json.dumps(transaction, sort_keys=True)
        signature_bytes = base64.b64decode(signature)
        public_key.verify(
            signature_bytes,
            transaction_string.encode(),
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),

```

```
        hashes.SHA256()
    )
    return True
except:
    return False
```

```
class Blockchain:
```

```
    """Blockchain implementation with Proof of Work consensus"""
```

```
    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, [{
            'from': 'genesis',
            'to': 'network',
            'amount': 0,
            'timestamp': time.time()
        }])
```



```
    }, '0', self.difficulty)

    genesis_block.mine_block()

    self.chain.append(genesis_block)

    print("✓ Genesis block created")
```

```
def get_latest_block(self):

    """Get the most recent block"""

    return self.chain[-1]
```

```
def add_transaction(self, transaction, signature, sender_wallet):

    """Add a signed transaction to pending pool"""

    # Verify signature

    if not sender_wallet.verify_transaction(transaction, signature,
sender_wallet.public_key):

        print("X Invalid transaction signature")

        return False
```

```
    transaction['signature'] = signature

    self.pending_transactions.append(transaction)

    print(f"✓ Transaction added: {transaction['from'][:8]}... →
{transaction['to'][:8]}... ({transaction['amount']} coins)")

    return True
```

```
def mine_pending_transactions(self, miner_address):

    """Mine a new block with pending transactions"""

    if not self.pending_transactions:
```

```
print("No transactions to mine")  
return False
```

```
# Add mining reward transaction
```

```
reward_tx = {  
    'from': 'network',  
    'to': miner_address,  
    'amount': self.mining_reward,  
    'timestamp': time.time(),  
    'signature': 'mining_reward'  
}
```

```
transactions = self.pending_transactions + [reward_tx]
```

```
# Create and mine new block
```

```
new_block = Block(  
    len(self.chain),  
    transactions,  
    self.get_latest_block().hash,  
    self.difficulty  
)  
new_block.mine_block()
```

```
# Add to chain
```

```
self.chain.append(new_block)  
self.pending_transactions = []
```

```
print(f'✓ Block {new_block.index} added to chain')  
return new_block
```

```
def validate_chain(self):
```

```
    """Validate entire blockchain"""
```

```
    print("\nValidating blockchain...")
```

```
    for i in range(1, len(self.chain)):
```

```
        current_block = self.chain[i]
```

```
        previous_block = self.chain[i - 1]
```

```
        # Check hash
```

```
        if current_block.hash != current_block.calculate_hash():
```

```
            print(f'X Block {i} hash is invalid')
```

```
            return False
```

```
        # Check link to previous block
```

```
        if current_block.previous_hash != previous_block.hash:
```

```
            print(f'X Block {i} is not properly linked')
```

```
            return False
```

```
        # Check proof of work
```

```
        if current_block.hash[:self.difficulty] != '0' * self.difficulty:
```

```
            print(f'X Block {i} does not meet difficulty requirement')
```

```
            return False
```

```
print("✓ Blockchain is valid")
```

```
return True
```

```
def demonstrate_tampering(self):
```

```
    """Show that tampering is detectable"""
```

```
    if len(self.chain) < 2:
```

```
        print("Need at least 2 blocks to demonstrate tampering")
```

```
    return
```

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

```
print("TAMPER-PROOFING DEMONSTRATION")
```

```
print("="*70)
```

```
# Show original state
```

```
print("\nOriginal blockchain is valid:")
```

```
self.validate_chain()
```

```
# Tamper with a block
```

```
print(f"\n⚠ Attempting to tamper with block 1...")
```

```
original_amount = self.chain[1].transactions[0]['amount']
```

```
self.chain[1].transactions[0]['amount'] = 999999
```

```
print(f" Changed transaction amount: {original_amount} → 999999")
```

```
# Show detection
```

```
print("\nValidating tampered blockchain:")
```

```
self.validate_chain()
```

```
# Restore
```

```
self.chain[1].transactions[0]['amount'] = original_amount
```

```
print(f"\n✓ Restored original value: {original_amount}")
```

```
def get_balance(self, address):
```

```
    """Calculate balance for an address"""
```

```
    balance = 0
```

```
    for block in self.chain:
```

```
        for tx in block.transactions:
```

```
            if tx['to'] == address:
```

```
                balance += tx['amount']
```

```
            if tx['from'] == address:
```

```
                balance -= tx['amount']
```

```
    return balance
```

```
def display_chain(self):
```

```
    """Display the entire blockchain"""
```

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

```
    print("BLOCKCHAIN STATE")
```

```
    print("="*70)
```

```
    for block in self.chain:
```

```

    print(f"\nBlock #{block.index}")

    print(f'  Timestamp:
{datetime.fromtimestamp(block.timestamp).strftime('%Y-%m-%d
%H:%M:%S')}")

    print(f'  Hash: {block.hash}')
    print(f'  Previous: {block.previous_hash}')
    print(f'  Merkle Root: {block.merkle_root}')
    print(f'  Nonce: {block.nonce}')
    print(f'  Transactions: {len(block.transactions)}')
    for i, tx in enumerate(block.transactions, 1):
        print(f'    {i}. {tx['from'][:8]}... → {tx['to'][:8]}... : {tx['amount']}
coins")

```

```

def demonstrate_blockchain():

```

```

    """Full blockchain demonstration"""

```

```

    print("="*70)

```

```

    print("BLOCKCHAIN CRYPTOGRAPHY DEMONSTRATION")

```

```

    print("="*70)

```

```

    # Create blockchain

```

```

    print("\n[1] INITIALIZING BLOCKCHAIN")

```

```

    print("-"*70)

```

```

    blockchain = Blockchain(difficulty=4)

```

```

    # Create wallets

```

```

    print("\n[2] CREATING DIGITAL WALLETS")

```

```
print("-"*70)

alice_wallet = DigitalWallet("Alice")
bob_wallet = DigitalWallet("Bob")
charlie_wallet = DigitalWallet("Charlie")


print(f'Alice's address: {alice_wallet.address}')
print(f'Bob's address: {bob_wallet.address}')
print(f'Charlie's address: {charlie_wallet.address}')


# Create and sign transactions
print("\n[3] CREATING SIGNED TRANSACTIONS")
print("-"*70)


# Give Alice some initial coins (simulate)
blockchain.mine_pending_transactions(alice_wallet.address)


# Alice sends to Bob
tx1 = {
    'from': alice_wallet.address,
    'to': bob_wallet.address,
    'amount': 10,
    'timestamp': time.time()
}

sig1 = alice_wallet.sign_transaction(tx1)
blockchain.add_transaction(tx1, sig1, alice_wallet)
```

```
# Alice sends to Charlie
```

```
tx2 = {  
    'from': alice_wallet.address,  
    'to': charlie_wallet.address,  
    'amount': 15,  
    'timestamp': time.time()  
}
```

```
sig2 = alice_wallet.sign_transaction(tx2)
```

```
blockchain.add_transaction(tx2, sig2, alice_wallet)
```

```
# Mine block
```

```
print("\n[4] MINING BLOCK")
```

```
print("-"*70)
```

```
blockchain.mine_pending_transactions(bob_wallet.address)
```

```
# Bob sends to Charlie
```

```
tx3 = {  
    'from': bob_wallet.address,  
    'to': charlie_wallet.address,  
    'amount': 5,  
    'timestamp': time.time()  
}
```

```
sig3 = bob_wallet.sign_transaction(tx3)
```

```
blockchain.add_transaction(tx3, sig3, bob_wallet)
```

```
# Mine another block
```



```
blockchain.mine_pending_transactions(charlie_wallet.address)

# Display blockchain
blockchain.display_chain()

# Show balances
print("\n[5] ACCOUNT BALANCES")
print("-"*70)
print(f'Alice: {blockchain.get_balance(alice_wallet.address)} coins')
print(f'Bob: {blockchain.get_balance(bob_wallet.address)} coins')
print(f'Charlie: {blockchain.get_balance(charlie_wallet.address)} coins')

# Validate chain
print("\n[6] BLOCKCHAIN VALIDATION")
print("-"*70)
blockchain.validate_chain()

# Demonstrate tampering
blockchain.demonstrate_tampering()

# Merkle tree demonstration
print("\n[7] MERKLE TREE VERIFICATION")
print("-"*70)
block = blockchain.chain[1]
merkle_tree = block.merkle_tree
```

```

print(f'Merkle Root: {merkle_tree.root}')
print(f'Transactions in block: {len(block.transactions)}')

# Verify a transaction
tx_index = 0
tx = json.dumps(block.transactions[tx_index], sort_keys=True)
proof = merkle_tree.get_proof(tx_index)
is_valid = merkle_tree.verify_proof(tx, proof, merkle_tree.root)

print(f'\nVerifying transaction {tx_index}:')
print(f' Proof length: {len(proof)} hashes')
print(f' Valid: {is_valid}')

```

```

# Consensus discussion
print("\n[8] CONSENSUS MECHANISMS")
print("-"*70)

```

```

consensus_info = ""

```

PROOF OF WORK (PoW) - Used in Bitcoin

- Miners compete to solve cryptographic puzzle
- First to find valid nonce broadcasts block
- Energy intensive but highly secure
- Difficulty adjusts based on network hashrate

PROOF OF STAKE (PoS) - Used in Ethereum 2.0

- Validators stake cryptocurrency to validate blocks
- Selected based on stake amount and age

- Energy efficient compared to PoW
- Risk of "nothing at stake" problem

DELEGATED PROOF OF STAKE (DPoS)

- Stakeholders vote for delegates
- Delegates validate transactions
- Faster but more centralized

PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT)

- Used in permissioned blockchains
- Consensus through voting among known validators
- Fast but requires known participants

"""

print(consensus_info)

Public key cryptography in blockchain

print("\n[9] PUBLIC KEY CRYPTOGRAPHY ROLE")

print("-"*70)

crypto_role = """

In Bitcoin/Ethereum:

1. Wallet Address Generation:

Private Key → Public Key → Address (via hashing)

2. Transaction Signing:

- User signs transaction with private key (ECDSA)

- Network verifies with public key
- Proves ownership without revealing private key

3. Non-repudiation:

- Signed transactions cannot be denied
- Immutable record on blockchain

4. Elliptic Curve Cryptography (ECC):

- Bitcoin uses secp256k1 curve
- 256-bit private key
- Smaller keys than RSA for same security

5. Hash Functions:

- SHA-256 for Bitcoin
- Keccak-256 for Ethereum
- RIPEMD-160 for address generation

"""

print(crypto_role)

if __name__ == "__main__":

 demonstrate_blockchain()

Output:

```
(.venv) PS D:\VsCode\University_Assignments\Crypto_Assignments\Advanced_Crypto_Assignment_Mayank> & D:/VsCode/University_
env/Scripts/python.exe d:/VsCode/University_Assignments/Crypto_Assignments/Advanced_Crypto_Assignment_Mayank/blockchain_s
=====
BLOCKCHAIN CRYPTOGRAPHY DEMONSTRATION
=====

[1] INITIALIZING BLOCKCHAIN
-----
Mining block 0 (difficulty: 4)... Mined! (nonce: 23186, time: 0.11s)
✓ Genesis block created

[2] CREATING DIGITAL WALLETS
-----
Alice's address: 0443941a8f0c4764e56b7d54c01db1f577232184
Bob's address: a0eb480bc289ea49f113b0fcb4f27287c8c5e6fd
Charlie's address: e26603da4f44c1a84bc3b2c28014431cfed7c9c2

[3] CREATING SIGNED TRANSACTIONS
-----
No transactions to mine
✓ Transaction added: 0443941a... → a0eb480b... (10 coins)
✓ Transaction added: 0443941a... → e26603da... (15 coins)

[4] MINING BLOCK
-----
Mining block 1 (difficulty: 4)... Mined! (nonce: 103962, time: 0.94s)
✓ Block 1 added to chain
✓ Transaction added: a0eb480b... → e26603da... (5 coins)
Mining block 2 (difficulty: 4)... Mined! (nonce: 77376, time: 0.55s)
✓ Block 2 added to chain

=====
BLOCKCHAIN STATE
=====

Block #0
  Timestamp: 2025-11-18 21:36:11
  Hash: 000036cb730110e8c352a00aa758e8034f77358a8b6ea9c613f4043d74476997
  Previous: 0
  Merkle Root: 9cddfe666f51210bc7138b5df6651b06a943608bdeab05c287b811c9e500c63c
  Nonce: 23186
```

Block #1

Timestamp: 2025-11-18 21:36:12

Hash: 0000e0ba39f256c77a3a6a5ba43e516ac24b23efcdcc85c6b2b2c0f58946aad2

Previous: 000036cb730110e8c352a00aa758e8034f77358a8b6ea9c613f4043d74476997

Merkle Root: 4a037a04c885b6234108a036c19b2344bcffe6ddbf0982ab6f1626db417251dd

Nonce: 103962

Transactions: 3

1. 0443941a... → a0eb480b... : 10 coins
2. 0443941a... → e26603da... : 15 coins
3. network... → a0eb480b... : 50 coins

Block #2

Timestamp: 2025-11-18 21:36:13

Hash: 0000ffff96a17a7d4cb941e7043c76a5f4a9d38f447b58683fecf226e18f09e3

Previous: 0000e0ba39f256c77a3a6a5ba43e516ac24b23efcdcc85c6b2b2c0f58946aad2

Merkle Root: 48e43621806a7824ff05e30086f908a92e6ea186b4a0e7342a3915fd9f7ccdd3

Nonce: 77376

Transactions: 2

1. a0eb480b... → e26603da... : 5 coins
2. network... → e26603da... : 50 coins

[5] ACCOUNT BALANCES

Alice: -25 coins

Bob: 55 coins

Charlie: 70 coins

[6] BLOCKCHAIN VALIDATION

Validating blockchain...

✓ Blockchain is valid

=====

TAMPER-PROOFING DEMONSTRATION

=====

```
(.venv) PS D:\VsCode\University_Assignments\Crypto_Assignments\Advanced_Crypto_Assignment_Mayank> & D:/V  
env/Scripts/python.exe d:/VsCode/University_Assignments/Crypto_Assignments/Advanced_Crypto_Assignment_Ma
```

Validating blockchain...

✓ Blockchain is valid

⚠ Attempting to tamper with block 1...

Changed transaction amount: 10 → 999999

Validating tampered blockchain:

Validating blockchain...

✗ Block 1 hash is invalid

✓ Restored original value: 10

[7] MERKLE TREE VERIFICATION

Merkle Root: 4a037a04c885b6234108a036c19b2344bcffe6ddbf0982ab6f1626db417251dd
Transactions in block: 3

Verifying transaction 0:

Proof length: 2 hashes

Valid: True

[8] CONSENSUS MECHANISMS

PROOF OF WORK (PoW) - Used in Bitcoin

- Miners compete to solve cryptographic puzzle
- First to find valid nonce broadcasts block
- Energy intensive but highly secure
- Difficulty adjusts based on network hashrate

PROOF OF STAKE (PoS) - Used in Ethereum 2.0

- Validators stake cryptocurrency to validate blocks
- Selected based on stake amount and age
- Energy efficient compared to PoW
- Risk of "nothing at stake" problem

DELEGATED PROOF OF STAKE (DPoS)

env/scripts/python.exe d:/vscode/University_Assignments/Crypto_Assignments/Advanced_Crypto_

- Consensus through voting among known validators
- Fast but requires known participants

[9] PUBLIC KEY CRYPTOGRAPHY ROLE

In Bitcoin/Ethereum:

1. Wallet Address Generation:

Private Key → Public Key → Address (via hashing)

2. Transaction Signing:

- User signs transaction with private key (ECDSA)
- Network verifies with public key
- Proves ownership without revealing private key

3. Non-repudiation:

- Signed transactions cannot be denied
- Immutable record on blockchain

4. Elliptic Curve Cryptography (ECC):

- Bitcoin uses secp256k1 curve
- 256-bit private key
- Smaller keys than RSA for same security

5. Hash Functions:

- SHA-256 for Bitcoin
- Keccak-256 for Ethereum
- RIPEMD-160 for address generation