CRYPTOGRAPHY

LAB ASSIGNMENT

Github: https://github.com/mayank24000/University_Assignments/tree/main/Crypto_Assignments/PublicKey_Crypto_Assignment_

Q-1 RSA Encryption/Decryption and Diffie-Hellman Key Exchange

CODE :

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes
import random
import time
from colorama import init, Fore, Style

init(autoreset=True)

class RSADemo:
    """RSA Encryption and Decryption Implementation"""

    def __init__(self, key_size=2048):
        self.key_size = key_size
        self.private_key = None
```

```
        self.public_key = None

    def generate_keys(self):
        """Generate RSA key pair"""
        print(f"{Fore.CYAN}Generating {self.key_size}-bit RSA key pair...")
        start = time.time()
        key = RSA.generate(self.key_size)
        self.private_key = key
        self.public_key = key.publickey()
        end = time.time()
        print(f"{Fore.GREEN}✓ Keys generated in {end-start:.4f} seconds")

        # Export keys
        private_pem = key.export_key()
        public_pem = self.public_key.export_key()

        # Save keys to files
        with open('private_key.pem', 'wb') as f:
            f.write(private_pem)
        with open('public_key.pem', 'wb') as f:
            f.write(public_pem)

        print(f"{Fore.YELLOW}Keys saved to private_key.pem and public_key.pem")

        return public_pem, private_pem

    def encrypt(self, message, public_key=None):
        """Encrypt message using RSA"""
        if public_key is None:
```

```
            public_key = self.public_key

        cipher = PKCS1_OAEP.new(public_key)
        ciphertext = cipher.encrypt(message)
        return ciphertext

    def decrypt(self, ciphertext):
        """Decrypt ciphertext using RSA"""
        cipher = PKCS1_OAEP.new(self.private_key)
        plaintext = cipher.decrypt(ciphertext)
        return plaintext

class DiffieHellman:
    """Diffie-Hellman Key Exchange Implementation"""

    def __init__(self):
        # Large prime and generator for production use
        self.p =
0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC740
20BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1
356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BF
B5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163BF0598DA48361C
55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F356208552BB9ED52907709696
6D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772C180E86039B
2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D22
61898FA051015728E5A8AACAA68FFFFFFFFFFFFFFFF
        self.g = 2

    def generate_private_key(self):
        """Generate private key for DH"""
        return random.randint(2, self.p - 2)
```

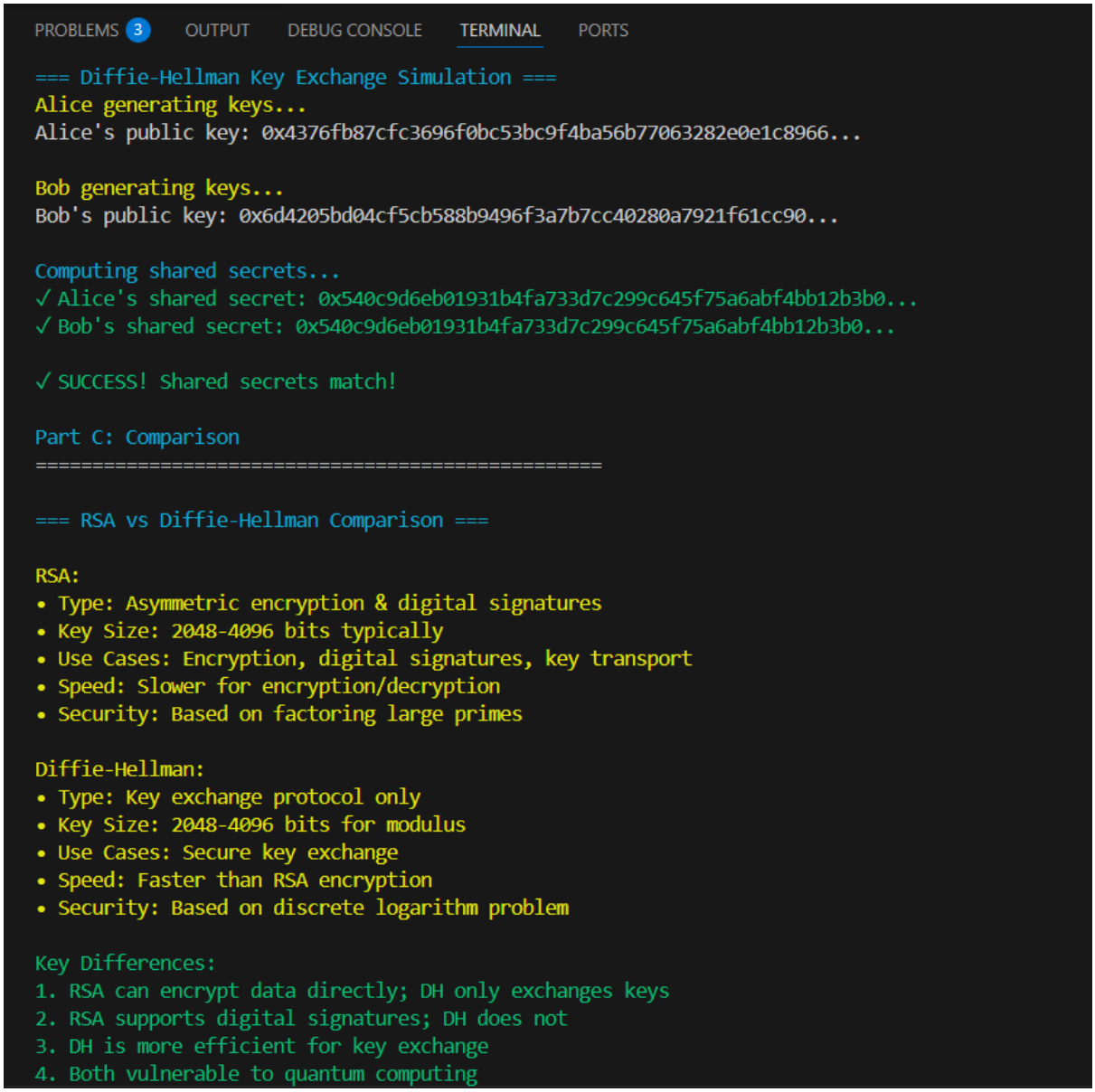
```
def generate_public_key(self, private_key):
    """Generate public key from private key"""
    return pow(self.g, private_key, self.p)

def compute_shared_secret(self, other_public, my_private):
    """Compute shared secret"""
    return pow(other_public, my_private, self.p)

def simulate_exchange(self):
    """Simulate complete DH key exchange between Alice and Bob"""
    print(f"\n{Fore.CYAN}=== Diffie-Hellman Key Exchange Simulation ===")

    # Alice generates keys
    print(f"{Fore.YELLOW}Alice generating keys...")
    alice_private = self.generate_private_key()
    alice_public = self.generate_public_key(alice_private)
    print(f"Alice's public key: {hex(alice_public)[:50]}...")

    # Bob generates keys
    print(f"\n{Fore.YELLOW}Bob generating keys...")
    bob_private = self.generate_private_key()
    bob_public = self.generate_public_key(bob_private)
    print(f"Bob's public key: {hex(bob_public)[:50]}...")

    # Exchange and compute shared secrets
    print(f"\n{Fore.CYAN}Computing shared secrets...")
    alice_shared = self.compute_shared_secret(bob_public, alice_private)
    bob_shared = self.compute_shared_secret(alice_public, bob_private)
```

```
        print(f"{Fore.GREEN}✓ Alice's shared secret: {hex(alice_shared)[:50]}...")
        print(f"{Fore.GREEN}✓ Bob's shared secret: {hex(bob_shared)[:50]}...")

        if alice_shared == bob_shared:
            print(f"\n{Fore.GREEN}✓ SUCCESS! Shared secrets match!")
        else:
            print(f"\n{Fore.RED}✗ ERROR! Shared secrets don't match!")

        return alice_shared

def compare_rsa_dh():
    """Compare RSA and Diffie-Hellman"""
    print(f"\n{Fore.CYAN}=== RSA vs Diffie-Hellman Comparison ===")
    print(f"""
{Fore.YELLOW}RSA:
• Type: Asymmetric encryption & digital signatures
• Key Size: 2048-4096 bits typically
• Use Cases: Encryption, digital signatures, key transport
• Speed: Slower for encryption/decryption
• Security: Based on factoring large primes

{Fore.YELLOW}Diffie-Hellman:
• Type: Key exchange protocol only
• Key Size: 2048-4096 bits for modulus
• Use Cases: Secure key exchange
• Speed: Faster than RSA encryption
• Security: Based on discrete logarithm problem

{Fore.GREEN}Key Differences:
```

```
1. RSA can encrypt data directly; DH only exchanges keys
2. RSA supports digital signatures; DH does not
3. DH is more efficient for key exchange
4. Both vulnerable to quantum computing
5. Often used together in protocols like TLS
    """)

def main():
    print(f"{Style.BRIGHT}{Fore.MAGENTA}=== Q1: Public Key Encryption & Key Exchange ===\n")

    # RSA Demo
    print(f"{Fore.CYAN}Part A: RSA Encryption/Decryption")
    print("="*50)
    rsa_demo = RSADemo(2048)
    rsa_demo.generate_keys()

    # Encrypt and decrypt a message
    message = b"This is a secure message using RSA encryption!"
    print(f"\n{Fore.YELLOW}Original message: {message.decode()}")

    ciphertext = rsa_demo.encrypt(message)
    print(f"{Fore.YELLOW}Ciphertext (hex): {ciphertext.hex()[:80]}...")

    decrypted = rsa_demo.decrypt(ciphertext)
    print(f"{Fore.GREEN}✓ Decrypted message: {decrypted.decode()}")

    # Diffie-Hellman Demo
    print(f"\n{Fore.CYAN}Part B: Diffie-Hellman Key Exchange")
```

```
    print("="*50)
    dh = DiffieHellman()
    shared_secret = dh.simulate_exchange()

    # Comparison
    print(f"\n{Fore.CYAN}Part C: Comparison")
    print("="*50)
    compare_rsa_dh()

    print(f"\n{Fore.GREEN}✓ Q1 Complete! Check generated key files.")

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

Solutoion :



Q2: Digital Signatures using RSA/DSA

Code:

```
from Crypto.PublicKey import RSA, DSA
from Crypto.Signature import pkcs1_15, DSS
from Crypto.Hash import SHA256
from Crypto.Random import get_random_bytes
import time
```

```
# Colorama is optional - use fallback if not available
try:
    from colorama import init, Fore, Style
    init(autoreset=True)
    HAS_COLOR = True
except ImportError:
    HAS_COLOR = False
    # Fallback class when colorama is not installed
    class Fore:
        CYAN = YELLOW = GREEN = RED = MAGENTA = ''
    class Style:
        BRIGHT = ''

class DigitalSignature:
    """Digital Signature Implementation using RSA and DSA"""

    def __init__(self):
        self.rsa_key = None
        self.dsa_key = None

    def generate_rsa_keypair(self, keysize=2048):
        """Generate RSA key pair for signatures"""
        print(f"{Fore.CYAN}Generating {keysize}-bit RSA key pair for signatures...")
        start = time.time()
        self.rsa_key = RSA.generate(keysize)
        end = time.time()
        print(f"{Fore.GREEN}✓ RSA keys generated in {end-start:.4f} seconds")

        # Save keys
```

```
        with open('rsa_signing_key.pem', 'wb') as f:
            f.write(self.rsa_key.export_key())
        with open('rsa_verify_key.pem', 'wb') as f:
            f.write(self.rsa_key.publickey().export_key())

        return self.rsa_key

    def generate_dsa_keypair(self, keysize=2048):
        """Generate DSA key pair for signatures"""
        print(f"{Fore.CYAN}Generating {keysize}-bit DSA key pair for signatures...")
        start = time.time()
        self.dsa_key = DSA.generate(keysize)
        end = time.time()
        print(f"{Fore.GREEN}✓ DSA keys generated in {end-start:.4f} seconds")

        # Save keys
        with open('dsa_signing_key.pem', 'wb') as f:
            f.write(self.dsa_key.export_key())
        with open('dsa_verify_key.pem', 'wb') as f:
            f.write(self.dsa_key.publickey().export_key())

        return self.dsa_key

    def sign_message_rsa(self, message):
        """Sign a message using RSA"""
        if not self.rsa_key:
            self.generate_rsa_keypair()

        # Hash the message
```

```
    h = SHA256.new(message)

    # Sign the hash
    signature = pkcs1_15.new(self.rsa_key).sign(h)

    print(f"{Fore.GREEN}✓ Message signed with RSA")
    print(f"{Fore.YELLOW}Signature (hex): {signature.hex()[:80]}...")

    return signature, h

def verify_signature_rsa(self, message, signature, public_key=None):
    """Verify RSA signature"""
    if public_key is None:
        public_key = self.rsa_key.publickey()

    h = SHA256.new(message)

    try:
        pkcs1_15.new(public_key).verify(h, signature)
        print(f"{Fore.GREEN}✓ RSA Signature verified successfully!")
        return True
    except (ValueError, TypeError) as e:
        print(f"{Fore.RED}✗ RSA Signature verification failed: {e}")
        return False

def sign_message_dsa(self, message):
    """Sign a message using DSA"""
    if not self.dsa_key:
        self.generate_dsa_keypair()
```

```
    # Hash the message
    h = SHA256.new(message)

    # Sign the hash
    signature = DSS.new(self.dsa_key, 'fips-186-3').sign(h)

    print(f"{Fore.GREEN}✓ Message signed with DSA")
    print(f"{Fore.YELLOW}Signature (hex): {signature.hex()[:80]}...")

    return signature, h

def verify_signature_dsa(self, message, signature, public_key=None):
    """Verify DSA signature"""
    if public_key is None:
        public_key = self.dsa_key.publickey()

    h = SHA256.new(message)

    try:
        DSS.new(public_key, 'fips-186-3').verify(h, signature)
        print(f"{Fore.GREEN}✓ DSA Signature verified successfully!")
        return True
    except (ValueError, TypeError) as e:
        print(f"{Fore.RED}✗ DSA Signature verification failed: {e}")
        return False

def demonstrate_tampering(self, message, signature, algorithm='RSA'):
    """Demonstrate what happens when message is tampered"""
```

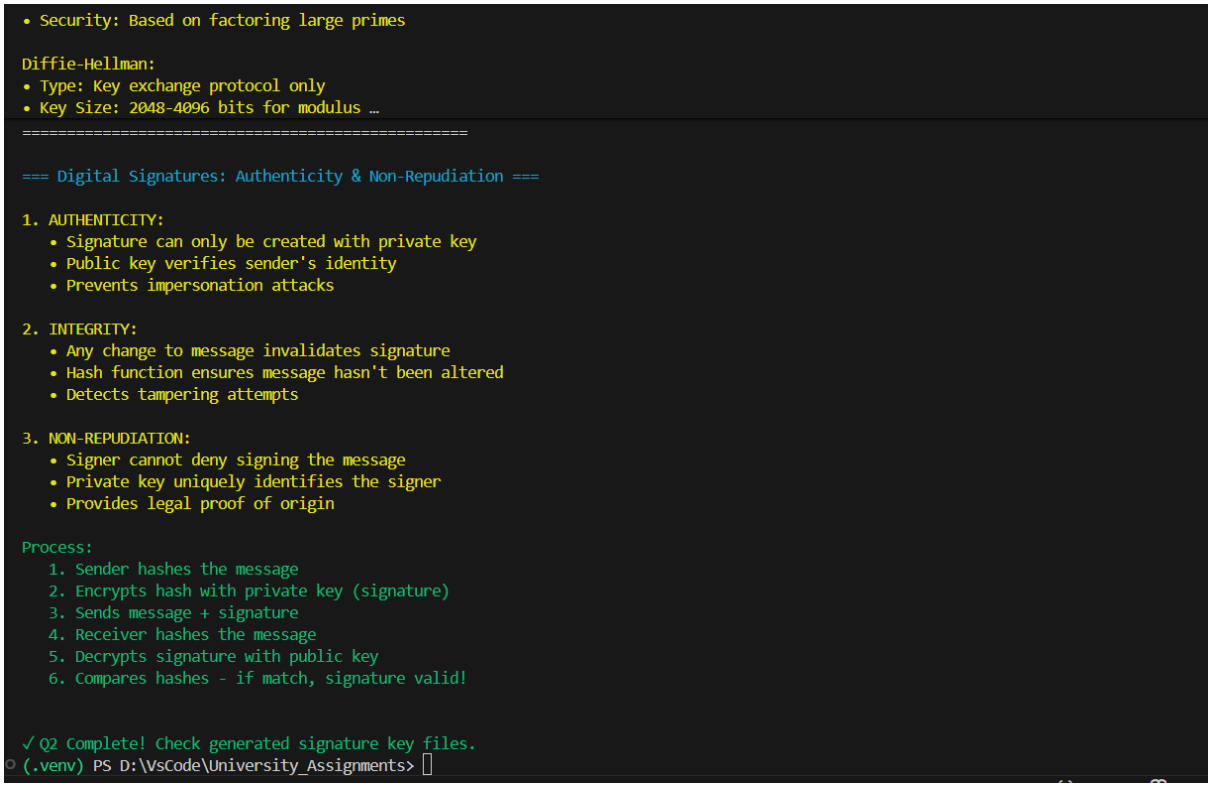
```
        print(f"\n{Fore.YELLOW}=== Demonstrating Message Tampering ===")

        # Tamper with the message
        tampered_message = message.replace(b"secure", b"hacked")
        print(f"Original: {message.decode()}")
        print(f"Tampered: {tampered_message.decode()}")

        if algorithm == 'RSA':
            result = self.verify_signature_rsa(tampered_message, signature)
        else:
            result = self.verify_signature_dsa(tampered_message, signature)

        if not result:
            print(f"{Fore.GREEN}✓ Tampering detected! Signature verification failed as expected.")

def explain_digital_signatures():
    """Explain how digital signatures ensure authenticity and non-repudiation"""
    print(f"\n{Fore.CYAN}=== Digital Signatures: Authenticity & Non-Repudiation ===")
    print(f"""
{Fore.YELLOW}1. AUTHENTICITY:
  • Signature can only be created with private key
  • Public key verifies sender's identity
  • Prevents impersonation attacks

{Fore.YELLOW}2. INTEGRITY:
  • Any change to message invalidates signature
  • Hash function ensures message hasn't been altered
  • Detects tampering attempts
```

```
{Fore.YELLOW}3. NON-REPUDIATION:
  • Signer cannot deny signing the message
  • Private key uniquely identifies the signer
  • Provides legal proof of origin

{Fore.GREEN}Process:
  1. Sender hashes the message
  2. Encrypts hash with private key (signature)
  3. Sends message + signature
  4. Receiver hashes the message
  5. Decrypts signature with public key
  6. Compares hashes - if match, signature valid!
  """)

def main():
    print(f"{Style.BRIGHT}{Fore.MAGENTA}=== Q2: Digital Signatures ===\n")

    if not HAS_COLOR:
        print("Note: Install 'colorama' for colored output: pip install colorama\n")

    ds = DigitalSignature()

    # Test message
    message = b"This is a secure and authenticated message"

    # RSA Signatures
    print(f"{Fore.CYAN}Part A: RSA Digital Signatures")
    print("="*50)
```

```
ds.generate_rsa_keypair()
print(f"\n{Fore.YELLOW}Message to sign: {message.decode()}")

# Sign and verify with RSA
rsa_signature, _ = ds.sign_message_rsa(message)
ds.verify_signature_rsa(message, rsa_signature)

# Demonstrate tampering detection
ds.demonstrate_tampering(message, rsa_signature, 'RSA')

# DSA Signatures
print(f"\n{Fore.CYAN}Part B: DSA Digital Signatures")
print("="*50)
ds.generate_dsa_keypair()

# Sign and verify with DSA
dsa_signature, _ = ds.sign_message_dsa(message)
ds.verify_signature_dsa(message, dsa_signature)

# Demonstrate tampering detection
ds.demonstrate_tampering(message, dsa_signature, 'DSA')

# Explain concepts
print(f"\n{Fore.CYAN}Part C: Discussion")
print("="*50)
explain_digital_signatures()

print(f"\n{Fore.GREEN}✓ Q2 Complete! Check generated signature key files.")
```

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

Solution :



Q3: Public Key Infrastructure (PKI) Simulation - Enhanced with debugging

Code:

```
from cryptography import x509
from cryptography.x509.oid import NameOID, ExtensionOID
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.backends import default_backend
import datetime
```

```
import os
import sys
import traceback

# Optional colorama with fallback
try:
    from colorama import init, Fore, Style
    init(autoreset=True)
    HAS_COLOR = True
except ImportError:
    HAS_COLOR = False
    class Fore:
        CYAN = YELLOW = GREEN = RED = MAGENTA = ''
    class Style:
        BRIGHT = ''

class PKISimulator:
    """Simulate basic PKI operations"""

    def __init__(self):
        self.ca_key = None
        self.ca_cert = None

        # Get absolute path for certificates directory
        self.cert_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), "certificates")

        # Debug: Show where we're trying to save certificates
        print(f"{Fore.YELLOW}Certificate directory: {self.cert_dir}")
        print(f"{Fore.YELLOW}Current working directory: {os.getcwd()}")
```

```
        # Create directory with error handling
        try:
            if not os.path.exists(self.cert_dir):
                os.makedirs(self.cert_dir)
                print(f"{Fore.GREEN}✓ Created certificates directory: {self.cert_dir}")
            else:
                print(f"{Fore.GREEN}✓ Certificates directory already exists: {self.cert_dir}")

            # Test write permissions
            test_file = os.path.join(self.cert_dir, "test_write.txt")
            with open(test_file, 'w') as f:
                f.write("test")
            os.remove(test_file)
            print(f"{Fore.GREEN}✓ Write permissions verified for certificates directory")

        except PermissionError as e:
            print(f"{Fore.RED}✗ Permission denied creating/accessing directory: {e}")
            print(f"{Fore.RED}  Try running with administrator privileges or choose a different directory")
            sys.exit(1)
        except Exception as e:
            print(f"{Fore.RED}✗ Error creating certificates directory: {e}")
            traceback.print_exc()
            sys.exit(1)

    def create_ca_certificate(self):
        """Create a Certificate Authority (CA) certificate"""
        print(f"\n{Fore.CYAN}Creating Certificate Authority (CA)...")
```

```
        try:
            # Generate CA private key
            print(f"{Fore.YELLOW}  Generating 4096-bit RSA key...")
            self.ca_key = rsa.generate_private_key(
                public_exponent=65537,
                key_size=4096,
                backend=default_backend()
            )
            print(f"{Fore.GREEN}  ✓ CA private key generated")

            # Create CA certificate
            subject = issuer = x509.Name([
                x509.NameAttribute(NameOID.COUNTRY_NAME, "IN"),
                x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "Delhi"),
                x509.NameAttribute(NameOID.LOCALITY_NAME, "New Delhi"),
                x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Demo CA
Authority"),
                x509.NameAttribute(NameOID.COMMON_NAME, "Demo Root CA"),
            ])

            # Use timezone-aware datetime
            now = datetime.datetime.now(datetime.timezone.utc)

            print(f"{Fore.YELLOW}  Building CA certificate...")
            self.ca_cert = (
                x509.CertificateBuilder()
                .subject_name(subject)
                .issuer_name(issuer)
```

```
    .public_key(self.ca_key.public_key())
    .serial_number(x509.random_serial_number())
    .not_valid_before(now)
    .not_valid_after(now + datetime.timedelta(days=3650))
    .add_extension(
        x509.BasicConstraints(ca=True, path_length=None),
        critical=True,
    )
    .add_extension(
        x509.KeyUsage(
            key_cert_sign=True,
            crl_sign=True,
            digital_signature=False,
            content_commitment=False,
            key_encipherment=False,
            data_encipherment=False,
            key_agreement=False,
            encipher_only=False,
            decipher_only=False,
        ),
        critical=True,
    )
    .sign(self.ca_key, hashes.SHA256(), backend=default_backend())
)
print(f"{Fore.GREEN}  ✓ CA certificate created")

# Save CA certificate and key
ca_cert_path = os.path.join(self.cert_dir, "ca_certificate.pem")
ca_key_path = os.path.join(self.cert_dir, "ca_private_key.pem")
```

```
        # Save certificate
        print(f"{Fore.YELLOW}  Saving CA certificate to {ca_cert_path}...")
        with open(ca_cert_path, "wb") as f:
            cert_bytes = self.ca_cert.public_bytes(serialization.Encoding.PEM)
            f.write(cert_bytes)
            print(f"{Fore.GREEN}  ✓ Written {len(cert_bytes)} bytes")

        # Verify certificate was saved
        if os.path.exists(ca_cert_path):
            file_size = os.path.getsize(ca_cert_path)
            print(f"{Fore.GREEN}✓ CA Certificate saved successfully: {ca_cert_path}
({file_size} bytes)")
        else:
            print(f"{Fore.RED}✗ Failed to save CA certificate!")

        # Save private key
        print(f"{Fore.YELLOW}  Saving CA private key to {ca_key_path}...")
        with open(ca_key_path, "wb") as f:
            key_bytes = self.ca_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.TraditionalOpenSSL,
                encryption_algorithm=serialization.NoEncryption()
            )
            f.write(key_bytes)
            print(f"{Fore.GREEN}  ✓ Written {len(key_bytes)} bytes")

        # Verify key was saved
        if os.path.exists(ca_key_path):
```

```
            file_size = os.path.getsize(ca_key_path)
            print(f"{Fore.GREEN}✓ CA Private Key saved successfully: {ca_key_path} ({file_size} bytes)")
        else:
            print(f"{Fore.RED}✗ Failed to save CA private key!")

        return self.ca_cert, self.ca_key

    except Exception as e:
        print(f"{Fore.RED}✗ Error creating CA certificate: {e}")
        traceback.print_exc()
        return None, None

def create_self_signed_certificate(self, common_name="example.com"):
    """Create a self-signed X.509 certificate"""
    print(f"\n{Fore.CYAN}Creating self-signed certificate for {common_name}...")

    try:
        # Generate private key
        print(f"{Fore.YELLOW}  Generating 2048-bit RSA key...")
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        print(f"{Fore.GREEN}  ✓ Private key generated")

        # Certificate details
        subject = issuer = x509.Name([
```

```
            x509.NameAttribute(NameOID.COUNTRY_NAME, "IN"),
            x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME,
"Maharashtra"),
            x509.NameAttribute(NameOID.LOCALITY_NAME, "Mumbai"),
            x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Demo
Organization"),
            x509.NameAttribute(NameOID.COMMON_NAME, common_name),
        ])

        # Use timezone-aware datetime
        now = datetime.datetime.now(datetime.timezone.utc)

        # Create certificate
        print(f"{Fore.YELLOW}  Building self-signed certificate...")
        cert = (
            x509.CertificateBuilder()
            .subject_name(subject)
            .issuer_name(issuer)
            .public_key(private_key.public_key())
            .serial_number(x509.random_serial_number())
            .not_valid_before(now)
            .not_valid_after(now + datetime.timedelta(days=365))
            .add_extension(
                x509.SubjectAlternativeName([
                    x509.DNSName(common_name),
                    x509.DNSName(f"www.{common_name}"),
                ]),
                critical=False,
            )
            .sign(private_key, hashes.SHA256(), backend=default_backend())
```

```
        )
        print(f"{Fore.GREEN}  ✓ Certificate created")

        # Clean filename (remove dots for safety)
        safe_name = common_name.replace(".", "_")

        # Save certificate and key
        cert_path = os.path.join(self.cert_dir, f"{safe_name}_selfsigned.pem")
        key_path = os.path.join(self.cert_dir, f"{safe_name}_private_key.pem")

        # Save certificate
        print(f"{Fore.YELLOW}  Saving certificate to {cert_path}...")
        with open(cert_path, "wb") as f:
            cert_bytes = cert.public_bytes(serialization.Encoding.PEM)
            f.write(cert_bytes)
            print(f"{Fore.GREEN}  ✓ Written {len(cert_bytes)} bytes")

        # Verify certificate was saved
        if os.path.exists(cert_path):
            file_size = os.path.getsize(cert_path)
            print(f"{Fore.GREEN}✓ Self-signed certificate saved: {cert_path} ({file_size}
bytes)")
        else:
            print(f"{Fore.RED}✗ Failed to save certificate!")

        # Save private key
        print(f"{Fore.YELLOW}  Saving private key to {key_path}...")
        with open(key_path, "wb") as f:
            key_bytes = private_key.private_bytes(
```

```
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.TraditionalOpenSSL,
                encryption_algorithm=serialization.NoEncryption()
            )
            f.write(key_bytes)
            print(f"{Fore.GREEN}  ✓ Written {len(key_bytes)} bytes")

        # Verify key was saved
        if os.path.exists(key_path):
            file_size = os.path.getsize(key_path)
            print(f"{Fore.GREEN}✓ Private key saved: {key_path} ({file_size} bytes)")
        else:
            print(f"{Fore.RED}✗ Failed to save private key!")

        self.display_certificate_info(cert)

        return cert, private_key

    except Exception as e:
        print(f"{Fore.RED}✗ Error creating self-signed certificate: {e}")
        traceback.print_exc()
        return None, None

def create_ca_signed_certificate(self, common_name="client.example.com"):
    """Create a certificate signed by CA"""
    if not self.ca_cert or not self.ca_key:
        print(f"{Fore.YELLOW}CA not found, creating one first...")
        self.create_ca_certificate()
```

```
    print(f"\n{Fore.CYAN}Creating CA-signed certificate for {common_name}...")

    try:
        # Generate private key for the certificate
        print(f"{Fore.YELLOW}  Generating 2048-bit RSA key...")
        cert_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        print(f"{Fore.GREEN}  ✓ Private key generated")

        # Certificate details
        subject = x509.Name([
            x509.NameAttribute(NameOID.COUNTRY_NAME, "IN"),
            x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "Karnataka"),
            x509.NameAttribute(NameOID.LOCALITY_NAME, "Bangalore"),
            x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Client
Organization"),
            x509.NameAttribute(NameOID.COMMON_NAME, common_name),
        ])

        # Use timezone-aware datetime
        now = datetime.datetime.now(datetime.timezone.utc)

        # Create certificate signed by CA
        print(f"{Fore.YELLOW}  Building CA-signed certificate...")
        cert = (
            x509.CertificateBuilder()
```

```
    .subject_name(subject)
    .issuer_name(self.ca_cert.issuer)
    .public_key(cert_key.public_key())
    .serial_number(x509.random_serial_number())
    .not_valid_before(now)
    .not_valid_after(now + datetime.timedelta(days=365))
    .add_extension(
        x509.SubjectAlternativeName([
            x509.DNSName(common_name),
        ]),
        critical=False,
    )
    .sign(self.ca_key, hashes.SHA256(), backend=default_backend())
)
print(f"{Fore.GREEN}  ✓ Certificate created and signed by CA")

# Clean filename
safe_name = common_name.replace(".", "_")

# Save certificate and key
cert_path = os.path.join(self.cert_dir, f"{safe_name}_ca_signed.pem")
key_path = os.path.join(self.cert_dir, f"{safe_name}_ca_signed_key.pem")

# Save certificate
print(f"{Fore.YELLOW}  Saving certificate to {cert_path}...")
with open(cert_path, "wb") as f:
    cert_bytes = cert.public_bytes(serialization.Encoding.PEM)
    f.write(cert_bytes)
    print(f"{Fore.GREEN}  ✓ Written {len(cert_bytes)} bytes")
```

```
        # Verify certificate was saved
        if os.path.exists(cert_path):
            file_size = os.path.getsize(cert_path)
            print(f"{Fore.GREEN}✓ CA-signed certificate saved: {cert_path} ({file_size}
bytes)")
        else:
            print(f"{Fore.RED}✗ Failed to save certificate!")

        # Save private key
        print(f"{Fore.YELLOW}  Saving private key to {key_path}...")
        with open(key_path, "wb") as f:
            key_bytes = cert_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.TraditionalOpenSSL,
                encryption_algorithm=serialization.NoEncryption()
            )
            f.write(key_bytes)
            print(f"{Fore.GREEN}  ✓ Written {len(key_bytes)} bytes")

        # Verify key was saved
        if os.path.exists(key_path):
            file_size = os.path.getsize(key_path)
            print(f"{Fore.GREEN}✓ Private key saved: {key_path} ({file_size} bytes)")
        else:
            print(f"{Fore.RED}✗ Failed to save private key!")

        self.display_certificate_info(cert)
```

```
        return cert

    except Exception as e:
        print(f"{Fore.RED}✗ Error creating CA-signed certificate: {e}")
        traceback.print_exc()
        return None

def display_certificate_info(self, cert):
    """Display certificate information"""
    print(f"\n{Fore.YELLOW}Certificate Information:")
    print(f"  Serial Number: {cert.serial_number}")
    print(f"  Issuer: {cert.issuer.rfc4514_string()}")
    print(f"  Subject: {cert.subject.rfc4514_string()}")
    print(f"  Valid From: {cert.not_valid_before_utc}")
    print(f"  Valid Until: {cert.not_valid_after_utc}")
    print(f"  Signature Algorithm: {cert.signature_algorithm_oid._name}")

def list_certificates(self):
    """List all certificates in the directory"""
    print(f"\n{Fore.CYAN}=== Certificates in {self.cert_dir} ===")

    if os.path.exists(self.cert_dir):
        files = os.listdir(self.cert_dir)
        if files:
            for file in files:
                file_path = os.path.join(self.cert_dir, file)
                file_size = os.path.getsize(file_path)
                print(f"  {Fore.GREEN}✓ {file} ({file_size} bytes)")
        else:
```

```
            print(f"  {Fore.YELLOW}No certificates found in directory")
        else:
            print(f"  {Fore.RED}Directory does not exist!")

    def explain_pki_components(self):
        """Explain PKI components and trust models"""
        print(f"\n{Fore.CYAN}=== PKI Components and Trust Models ===")
        print(f"""
{Fore.YELLOW}1. CERTIFICATE AUTHORITY (CA):
  • Issues and signs digital certificates
  • Trusted third party
  • Maintains certificate database
  • Root CA → Intermediate CA → End-entity certificates

{Fore.YELLOW}2. CERTIFICATE REVOCATION LIST (CRL):
  • List of revoked certificates
  • Updated periodically by CA
  • Checked during certificate validation
  • Alternative: OCSP (Online Certificate Status Protocol)

{Fore.YELLOW}3. CERTIFICATE CHAIN:
  • Root CA Certificate (self-signed, trusted)
    └─> Intermediate CA Certificate
      └─> End-entity Certificate (website/user)
  • Each level signs the next
  • Verification traces back to trusted root

{Fore.YELLOW}4. TRUST MODELS:
```

{Fore.GREEN}a) Hierarchical Trust Model:

- Single root CA at top
- Tree structure with subordinate CAs
- Used in most web browsers

{Fore.GREEN}b) Web of Trust:

- No central authority
- Users sign each other's keys
- Used in PGP/GPG

{Fore.GREEN}c) Bridge Trust Model:

- Multiple PKI domains connected
- Bridge CA enables cross-certification

{Fore.YELLOW}5. BROWSER HTTPS VALIDATION:

1. Browser connects to HTTPS site
2. Server sends certificate chain
3. Browser verifies:
   - Certificate signature
   - Certificate validity dates
   - Domain name match
   - Not in revocation list
   - Chain leads to trusted root CA
4. If all checks pass → secure connection established
5. Otherwise → security warning displayed

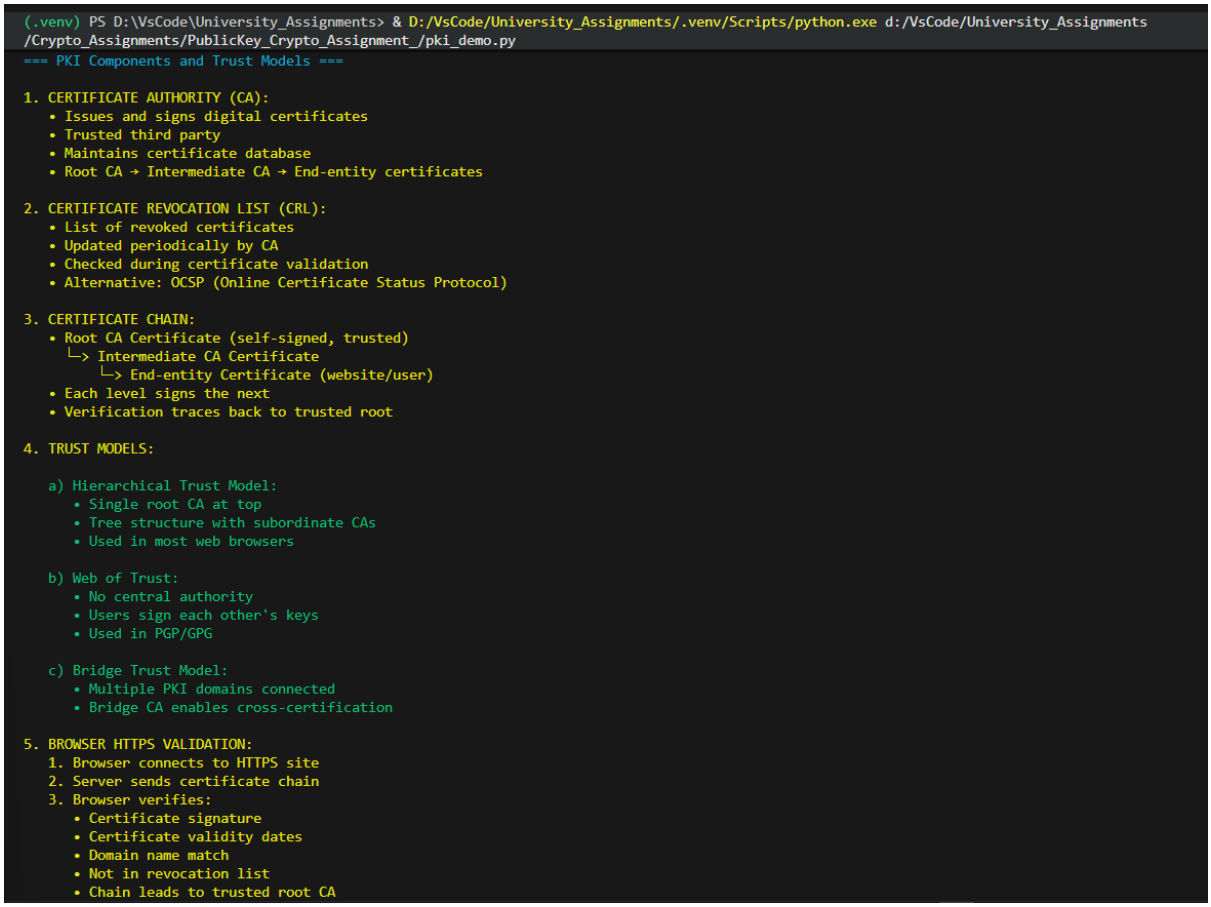
""")

def main():

```
    print(f"{Style.BRIGHT}{Fore.MAGENTA}=== Q3: Public Key Infrastructure (PKI) ===\n")

    if not HAS_COLOR:
        print("Note: Install 'colorama' for colored output: pip install colorama\n")

    try:
        pki = PKISimulator()

        # Part A: Create self-signed certificate
        print(f"\n{Fore.CYAN}Part A: Self-Signed X.509 Certificate")
        print("="*50)
        pki.create_self_signed_certificate("demo.example.com")

        # Create CA and CA-signed certificate
        print(f"\n{Fore.CYAN}Creating Certificate Hierarchy")
        print("="*50)
        pki.create_ca_certificate()
        pki.create_ca_signed_certificate("secure.example.com")

        # List all created certificates
        pki.list_certificates()

        # Part B & C: Explain PKI components
        print(f"\n{Fore.CYAN}Part B & C: PKI Components and Trust Models")
        print("="*50)
        pki.explain_pki_components()

        print(f"\n{Fore.GREEN}✓ Q3 Complete! Check 'certificates' directory for generated files.")
```

```
        print(f"{Fore.YELLOW}Full path: {pki.cert_dir}")

    except Exception as e:
        print(f"{Fore.RED}✗ Fatal error: {e}")
        traceback.print_exc()
        return 1

    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Solution :



Q4: Key Exchange Protocol Comparison - Classic DH vs ECDH

Code:

```
import random
import time
import hashlib

# Optional imports with fallbacks
try:
    from tinyec import registry
    HAS_TINYEC = True
except ImportError:
    HAS_TINYEC = False
    print("Warning: tinyec not installed. Install with: pip install tinyec")

try:
    from colorama import init, Fore, Style
    init(autoreset=True)
    HAS_COLOR = True
except ImportError:
    HAS_COLOR = False
    class Fore:
        CYAN = YELLOW = GREEN = RED = MAGENTA = ''
    class Style:
        BRIGHT = ''

try:
    import matplotlib.pyplot as plt
    import numpy as np
    HAS_MATPLOTLIB = True
```

```
except ImportError:
    HAS_MATPLOTLIB = False
    print("Warning: matplotlib not installed. Install with: pip install matplotlib numpy")
    # Minimal numpy fallback for basic operations
    class np:
        @staticmethod
        def mean(data):
            return sum(data) / len(data)

        @staticmethod
        def std(data):
            m = sum(data) / len(data)
            variance = sum((x - m) ** 2 for x in data) / len(data)
            return variance ** 0.5

class ClassicDiffieHellman:
    """Classic Diffie-Hellman Implementation"""

    def __init__(self, bits=2048):
        self.bits = bits
        if bits == 2048:
            # 2048-bit MODP group (RFC 3526)
            self.p = int("""FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
                29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
                EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
                E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
                EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
                C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
                83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
```

```
            670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
            E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
            DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
            15728E5A 8AACAA68 FFFFFFFF FFFFFFFF""".replace(" ", "").replace("\n", ""),
16)
            self.g = 2
        else:
            # Simple prime for demonstration
            self.p = self._generate_prime(bits)
            self.g = 2

    def _generate_prime(self, bits):
        """Generate a prime number (simplified)"""
        # For demonstration, using known safe primes
        if bits == 512:
            return
int("FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC7
4020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374F
E1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386
BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163BF0598DA4836
1C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F356208552BB9ED529077096
966D", 16)
        return int("FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD1", 16)

    def generate_private_key(self):
        """Generate private key"""
        return random.randint(2, self.p - 2)

    def generate_public_key(self, private_key):
        """Generate public key"""
        return pow(self.g, private_key, self.p)
```

```
    def compute_shared_secret(self, other_public, my_private):
        """Compute shared secret"""
        return pow(other_public, my_private, self.p)

    def perform_exchange(self):
        """Perform complete key exchange and measure time"""
        start_time = time.perf_counter()

        # Alice
        alice_private = self.generate_private_key()
        alice_public = self.generate_public_key(alice_private)

        # Bob
        bob_private = self.generate_private_key()
        bob_public = self.generate_public_key(bob_private)

        # Shared secrets
        alice_shared = self.compute_shared_secret(bob_public, alice_private)
        bob_shared = self.compute_shared_secret(alice_public, bob_private)

        end_time = time.perf_counter()

        return alice_shared == bob_shared, end_time - start_time

class EllipticCurveDiffieHellman:
    """Elliptic Curve Diffie-Hellman Implementation"""

    def __init__(self, curve_name="secp256r1"):
```

```
    if not HAS_TINYEC:
        raise ImportError("tinyec library required for ECDH. Install with: pip install tinyec")
    self.curve_name = curve_name
    self.curve = registry.get_curve(curve_name)

def generate_private_key(self):
    """Generate private key"""
    return random.randint(1, self.curve.field.n - 1)

def generate_public_key(self, private_key):
    """Generate public key (point on curve)"""
    return private_key * self.curve.g

def compute_shared_secret(self, other_public, my_private):
    """Compute shared secret"""
    shared_point = my_private * other_public
    # Use x-coordinate as shared secret
    return shared_point.x

def perform_exchange(self):
    """Perform complete key exchange and measure time"""
    start_time = time.perf_counter()

    # Alice
    alice_private = self.generate_private_key()
    alice_public = self.generate_public_key(alice_private)

    # Bob
    bob_private = self.generate_private_key()
```

```
        bob_public = self.generate_public_key(bob_private)

        # Shared secrets
        alice_shared = self.compute_shared_secret(bob_public, alice_private)
        bob_shared = self.compute_shared_secret(alice_public, bob_private)

        end_time = time.perf_counter()

        return alice_shared == bob_shared, end_time - start_time

def compare_protocols():
    """Compare Classic DH and ECDH protocols"""
    print(f"{Fore.CYAN}=== Protocol Comparison: Classic DH vs ECDH ===\n")

    # Test configurations
    test_configs = [
        ("Classic DH (512-bit)", ClassicDiffieHellman(512)),
        ("Classic DH (2048-bit)", ClassicDiffieHellman(2048)),
    ]

    # Add ECDH tests only if tinyec is available
    if HAS_TINYEC:
        test_configs.extend([
            ("ECDH (secp192r1)", EllipticCurveDiffieHellman("secp192r1")),
            ("ECDH (secp256r1)", EllipticCurveDiffieHellman("secp256r1")),
        ])
    else:
        print(f"{Fore.YELLOW}Note: ECDH tests skipped (tinyec not installed)\n")
```

```
results = []

for name, protocol in test_configs:
    print(f"{Fore.YELLOW}Testing {name}...")

    # Run multiple iterations for accurate timing
    times = []
    for _ in range(10):
        success, elapsed = protocol.perform_exchange()
        if success:
            times.append(elapsed * 1000)  # Convert to milliseconds

    avg_time = np.mean(times)
    std_time = np.std(times)

    results.append({
        'name': name,
        'avg_time': avg_time,
        'std_time': std_time
    })

    print(f"{Fore.GREEN}✓ Average time: {avg_time:.4f}ms (±{std_time:.4f}ms)")

# Create comparison chart if matplotlib is available
if HAS_MATPLOTLIB and len(results) > 0:
    create_comparison_chart(results)
else:
    print(f"\n{Fore.YELLOW}Note: Chart generation skipped (matplotlib not installed)")
```

```
    return results


def create_comparison_chart(results):
    """Create bar chart comparing protocols"""
    names = [r['name'] for r in results]
    times = [r['avg_time'] for r in results]
    errors = [r['std_time'] for r in results]

    plt.figure(figsize=(10, 6))
    colors = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#96CEB4']
    bars = plt.bar(names, times, yerr=errors, capsize=5, color=colors[:len(names)],
                   edgecolor='black', linewidth=1.5)

    plt.xlabel('Key Exchange Protocol', fontsize=12)
    plt.ylabel('Average Time (milliseconds)', fontsize=12)
    plt.title('Key Exchange Protocol Performance Comparison', fontsize=14,
fontweight='bold')
    plt.xticks(rotation=45, ha='right')

    # Add value labels on bars
    for bar, time_val in zip(bars, times):
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width()/2, height,
                 f'{time_val:.3f}ms', ha='center', va='bottom', fontsize=10)

    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.savefig('key_exchange_comparison.png', dpi=150)
    print(f"\n{Fore.GREEN}✓ Comparison chart saved as 'key_exchange_comparison.png'")
```

```
def explain_advantages():
    """Explain advantages of ECC over traditional methods"""
    print(f"\n{Fore.CYAN}=== Advantages of ECC in Resource-Constrained Environments ===")
    print(f"""
{Fore.YELLOW}1. SMALLER KEY SIZES:
  • ECC 256-bit ≈ RSA 3072-bit security level
  • ECC 384-bit ≈ RSA 7680-bit security level
  • Result: Less storage and bandwidth required

{Fore.YELLOW}2. FASTER COMPUTATION:
  • Smaller numbers = faster arithmetic operations
  • Lower power consumption
  • Ideal for IoT devices and mobile platforms

{Fore.YELLOW}3. MEMORY EFFICIENCY:
  • Reduced RAM requirements
  • Smaller certificate sizes
  • Better for embedded systems

{Fore.YELLOW}4. BATTERY LIFE:
  • Less computational overhead
  • Extended battery life in mobile devices
  • Critical for sensor networks

{Fore.YELLOW}5. SCALABILITY:
  • Better performance as security requirements increase
  • Future-proof against advancing threats
```

• Quantum-resistant variants being developed

{Fore.GREEN}REAL-WORLD APPLICATIONS:

• Mobile devices (iOS, Android)

• IoT sensors and actuators

• Blockchain and cryptocurrencies

• Smart cards and RFID

• Embedded automotive systems

• Satellite communications

""")

def security_comparison():

"""Compare security levels"""

print(f"\n{Fore.CYAN}=== Security Level Comparison ===")

security_table = """

Security Bit	RSA Key Size	DH Key Size	ECC Key Size
80	1024	1024	160
112	2048	2048	224
128	3072	3072	256
192	7680	7680	384
256	15360	15360	512

"""

print(f"{Fore.GREEN}{security_table}")

```
def main():
    print(f"{Style.BRIGHT}{Fore.MAGENTA}=== Q4: Key Exchange Protocol Comparison ===\n")

    if not HAS_COLOR:
        print("Note: Install 'colorama' for colored output: pip install colorama\n")

    # Part A: Implement both protocols
    print(f"{Fore.CYAN}Part A: Protocol Implementations")
    print("="*50)

    # Classic DH demonstration
    print(f"\n{Fore.YELLOW}Classic Diffie-Hellman Demo:")
    dh = ClassicDiffieHellman(512)  # Using smaller size for demo
    success, time_taken = dh.perform_exchange()
    print(f"{Fore.GREEN}✓ Key exchange successful: {success}")
    print(f"{Fore.GREEN}✓ Time taken: {time_taken*1000:.4f}ms")

    # ECDH demonstration
    if HAS_TINYEC:
        print(f"\n{Fore.YELLOW}Elliptic Curve Diffie-Hellman Demo:")
        ecdh = EllipticCurveDiffieHellman("secp256r1")
        success, time_taken = ecdh.perform_exchange()
        print(f"{Fore.GREEN}✓ Key exchange successful: {success}")
        print(f"{Fore.GREEN}✓ Time taken: {time_taken*1000:.4f}ms")
    else:
        print(f"\n{Fore.YELLOW}ECDH Demo skipped: Install tinyec for ECDH support")
```

```
    # Part B: Performance comparison
    print(f"\n{Fore.CYAN}Part B: Performance Measurement")
    print("="*50)
    results = compare_protocols()

    # Part C: Advantages discussion
    print(f"\n{Fore.CYAN}Part C: ECC Advantages")
    print("="*50)
    explain_advantages()
    security_comparison()

    print(f"\n{Fore.GREEN}✓ Q4 Complete!")
    if HAS_MATPLOTLIB:
        print(f"{Fore.GREEN}  Check 'key_exchange_comparison.png' for visualization.")

    # Show installation instructions if needed
    if not HAS_TINYEC or not HAS_MATPLOTLIB:
        print(f"\n{Fore.YELLOW}To enable all features, install missing dependencies:")
        if not HAS_TINYEC:
            print(f"  pip install tinyec")
        if not HAS_MATPLOTLIB:
            print(f"  pip install matplotlib numpy")

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

Solution :

```
ECDH Demo skipped: Install tinyec for ECDH support

Part B: Performance Measurement
================================================
=== Protocol Comparison: Classic DH vs ECDH ===

Note: ECDH tests skipped (tinyec not installed)

Testing Classic DH (512-bit)...
✓ Average time: 41.7156ms (±3.0578ms)
Testing Classic DH (2048-bit)...
✓ Average time: 122.7104ms (±6.3183ms)

Note: Chart generation skipped (matplotlib not installed)

Part C: ECC Advantages
================================================

=== Advantages of ECC in Resource-Constrained Environments ===

1. SMALLER KEY SIZES:
   • ECC 256-bit ≈ RSA 3072-bit security level
   • ECC 384-bit ≈ RSA 7680-bit security level
   • Result: Less storage and bandwidth required

2. FASTER COMPUTATION:
   • Smaller numbers = faster arithmetic operations
   • Lower power consumption
   • Ideal for IoT devices and mobile platforms

3. MEMORY EFFICIENCY:
   • Reduced RAM requirements
   • Smaller certificate sizes
   • Better for embedded systems

4. BATTERY LIFE:
   • Less computational overhead
   • Extended battery life in mobile devices
   • Critical for sensor networks

5. SCALABILITY:
   • Better performance as security requirements increase
   • Future-proof against advancing threats
   • Quantum-resistant variants being developed

REAL-WORLD APPLICATIONS:
   • Mobile devices (iOS, Android)
   • IoT sensors and actuators
   • Blockchain and cryptocurrencies
   • Smart cards and RFID
   • Embedded automotive systems
   • Satellite communications
```

=== Security Level Comparison ===

Security Bit	RSA Key Size	DH Key Size	ECC Key Size
80	1024	1024	160
112	2048	2048	224
128	3072	3072	256
192	7680	7680	384
256	15360	15360	512

Q5: SECURE COMMUNICATION PROTOCOLS ANALYSIS

==========

OPTION B: PGP ENCRYPTION ANALYSIS

==========

1. PGP HYBRID ENCRYPTION MODEL

PGP uses a hybrid cryptosystem combining symmetric and asymmetric encryption:

ENCRYPTION PROCESS:

1. Generate random session key (AES-256 typically)

2. Encrypt message with session key (symmetric encryption)

3. Encrypt session key with recipient's public key (RSA/ECC)

4. Send: Encrypted message + Encrypted session key

DECRYPTION PROCESS:

1. Decrypt session key using recipient's private key

2. Decrypt message using recovered session key

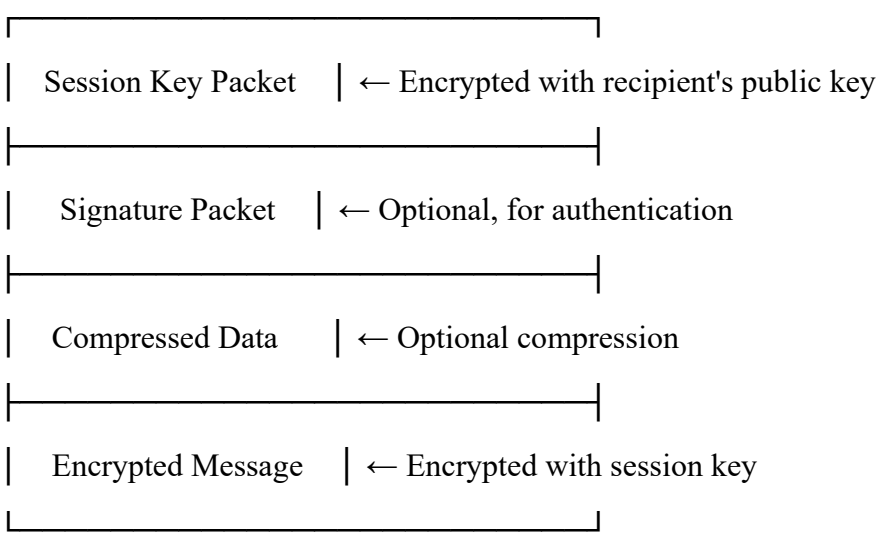
ADVANTAGES:

• Speed of symmetric encryption for large data

• Security of public key cryptography for key exchange

• No need for pre-shared keys

2. PGP MESSAGE FORMAT



3. KEY COMPONENTS

• Public Key: Used for encryption and signature verification

• Private Key: Used for decryption and signing

• Session Key: Symmetric key for actual message encryption

• Passphrase: Protects private key storage

4. WEB OF TRUST MODEL

Unlike PKI's hierarchical model, PGP uses Web of Trust:

• Users sign each other's keys

• Trust levels: Unknown, Untrusted, Marginal, Full, Ultimate

• No central authority required

• Based on personal relationships and reputation

5. BEST PRACTICES FOR SECURE KEY EXCHANGE

---

a) KEY VERIFICATION:

- Verify key fingerprints out-of-band (phone, in-person)
- Use multiple channels for confirmation
- Check key signatures from trusted parties

b) KEY SERVERS:

- Use reputable key servers (keys.openpgp.org)
- Verify keys from multiple sources
- Regularly update key revocation status

c) KEY MANAGEMENT:

- Use strong passphrases for private keys
- Generate keys on secure, offline systems
- Maintain secure backups of private keys
- Set expiration dates for keys
- Create revocation certificates in advance

d) OPERATIONAL SECURITY:

- Use air-gapped computers for sensitive operations
- Verify sender's signature on received messages
- Be cautious of key substitution attacks
- Regularly rotate encryption keys

6. COMMON ATTACKS AND MITIGATIONS

ATTACK: Man-in-the-Middle key substitution

MITIGATION: Always verify key fingerprints

ATTACK: Key compromise

MITIGATION: Use key revocation, maintain revocation certificate

ATTACK: Metadata leakage

MITIGATION: Use anonymous remailers, Tor network

ATTACK: Weak random number generation

MITIGATION: Use hardware RNG, entropy gathering daemons

7. PGP vs SSL/TLS COMPARISON

Aspect	PGP	SSL/TLS
Trust Model	Web of Trust	PKI (CA-based)
Use Case	Email, Files	Web, Real-time
Key Exchange	Manual/Keyserver	Automatic
Perfect FS	No (standard)	Yes (with ECDHE)
Transparency	User-controlled	CA-controlled

8. IMPLEMENTATION EXAMPLE (Python with python-gnupg)

```
import gnupg

# Initialize GPG
gpg = gnupg.GPG()

# Generate key pair
input_data = gpg.gen_key_input(
    name_real="Alice Smith",
    name_email="alice@example.com",
    key_type="RSA",
    key_length=4096
)
key = gpg.gen_key(input_data)

# Encrypt message
encrypted = gpg.encrypt(
    "Secret message",
    recipients=['bob@example.com'],
    sign='alice@example.com'
)

# Decrypt message
decrypted = gpg.decrypt(encrypted.data)
```

9. MODERN DEVELOPMENTS

• PGP/MIME for email integration

• OpenPGP smart cards for key storage

• ProtonMail/Tutanota for user-friendly encrypted email

• Signal Protocol combining best of PGP and modern crypto

• Post-quantum cryptography preparations

10. CONCLUSION

PGP remains a robust solution for asynchronous encrypted communication, particularly for email and file encryption. Its decentralized trust model provides user autonomy but requires more user education compared to SSL/TLS's automated approach. The hybrid encryption model efficiently combines security with performance, making it suitable for encrypting large amounts of data.

==========

ALTERNATIVE ANALYSIS: SSL/TLS HANDSHAKE

==========

For SSL/TLS analysis, the following would be examined:

1. HANDSHAKE PROCESS:
   • ClientHello → ServerHello
   • Certificate exchange
   • Key exchange (ECDHE/RSA)
   • ChangeCipherSpec
   • Encrypted application data

2. CERTIFICATE VALIDATION:
   • Chain verification
   • OCSP checking

- Certificate pinning

3. CIPHER SUITES:

- Modern: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- Forward secrecy with ECDHE
- AEAD ciphers (AES-GCM)

4. WIRESHARK ANALYSIS:

- Capture HTTPS session
- Identify handshake packets
- Examine certificate details
- Note cipher suite negotiation