Cryptography

Lab Assignment 3

Github Link :
https://github.com/AviatrixK/Uni-Assignments/tree/main/Crypto-assignments/PublicKey_Crypto_Assignment_Kiruthika

digitalSignature.py

```
"""
Q2: Digital Signatures using RSA/DSA
Author: Your Name
Date: 2024
"""

import os
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

class DigitalSignatureDemo:
    """Digital Signature Generation and
Verification"""

    def __init__(self, algorithm='RSA'):
        self.algorithm = algorithm
        self.private_key = None
        self.public_key = None

    def generate_keys(self, key_size=2048):
        """Generate key pair for signing"""
```

```
        print(f"\nGenerating {key_size}-bit {self.algorithm} key pair...")

        if self.algorithm == 'RSA':
            self.private_key = RSA.generate(key_size)
            self.public_key = self.private_key.publickey()
        elif self.algorithm == 'DSA':
            self.private_key = DSA.generate(key_size)
            self.public_key = self.private_key.publickey()

        # Save keys
        os.makedirs('keys', exist_ok=True)

        with open(f'keys/{self.algorithm.lower()}_signing_private.pem', 'wb') as f:
            f.write(self.private_key.export_key())

        with open(f'keys/{self.algorithm.lower()}_signing_public.pem', 'wb') as f:
            f.write(self.public_key.export_key())
```

```
        print(f"Keys saved to 'keys/' directory")

    def sign_message(self, message):
        """Sign a message with private key"""
        # Hash the message
        h = SHA256.new(message)

        # Sign with appropriate algorithm
        if self.algorithm == 'RSA':
            signature =
pkcs1_15.new(self.private_key).sign(h)
        elif self.algorithm == 'DSA':
            signature = DSS.new(self.private_key,
'fips-186-3').sign(h)

        return signature, h

    def verify_signature(self, message, signature,
public_key=None):
        """Verify signature with public key"""
        if public_key is None:
            public_key = self.public_key

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

```
        try:
            if self.algorithm == 'RSA':
                pkcs1_15.new(public_key).verify(h, 
signature)
            elif self.algorithm == 'DSA':
                DSS.new(public_key, 'fips-186-
3').verify(h, signature)
            return True
        except (ValueError, TypeError):
            return False

    def demo(self):
        """Run digital signature demo"""
        print("\n" + "="*60)
        print(f"{self.algorithm} DIGITAL SIGNATURE 
DEMO")
        print("="*60)

        # Generate keys
        self.generate_keys()

        # Test messages
        messages = [
            b"This is an authentic message from 
Alice",
```

```
            b"Contract Agreement: Terms and
Conditions Apply",
            b"Transaction ID: 12345, Amount:
$1000.00"
        ]

        for i, message in enumerate(messages, 1):
            print(f"\n--- Message {i} ---")
            print(f"Original: {message.decode()}")

            # Sign
            start_time = time.time()
            signature, msg_hash =
self.sign_message(message)
            sign_time = time.time() - start_time

            print(f"Message hash:
{msg_hash.hexdigest()[:32]}...")
            print(f"Signature:
{signature.hex()[:64]}...")
            print(f"Signing time: {sign_time:.6f}
seconds")

            # Verify
            start_time = time.time()
            is_valid =
self.verify_signature(message, signature)
```

```
            verify_time = time.time() - start_time

            print(f"Verification result: {'✓
VALID' if is_valid else '✗ INVALID'}")
            print(f"Verification time:
{verify_time:.6f} seconds")

            # Test tampering
            tampered_message = message + b"
TAMPERED"
            is_valid_tampered =
self.verify_signature(tampered_message, signature)
            print(f"Tampered message verification:
{'✓ VALID' if is_valid_tampered else '✗
INVALID'}")

class SignatureProperties:
    """Demonstrate properties of digital
signatures"""

    @staticmethod
    def demonstrate_properties():
        print("\n" + "="*60)
        print("DIGITAL SIGNATURE PROPERTIES")
        print("="*60)
```

```
        print("""
1. AUTHENTICITY
   - Verifies the identity of the message sender
   - Only the holder of the private key can create valid signature
   - Public key verification proves sender's identity

2. INTEGRITY
   - Any modification to the message invalidates the signature
   - Ensures message hasn't been altered in transit
   - Based on cryptographic hash functions

3. NON-REPUDIATION
   - Sender cannot deny having signed the message
   - Signature is mathematically bound to both message and signer
   - Provides legal evidence of agreement/transaction

4. USE CASES
   - Email authentication (PGP/S-MIME)
   - Software distribution (code signing)
   - Financial transactions
   - Legal documents and contracts
```

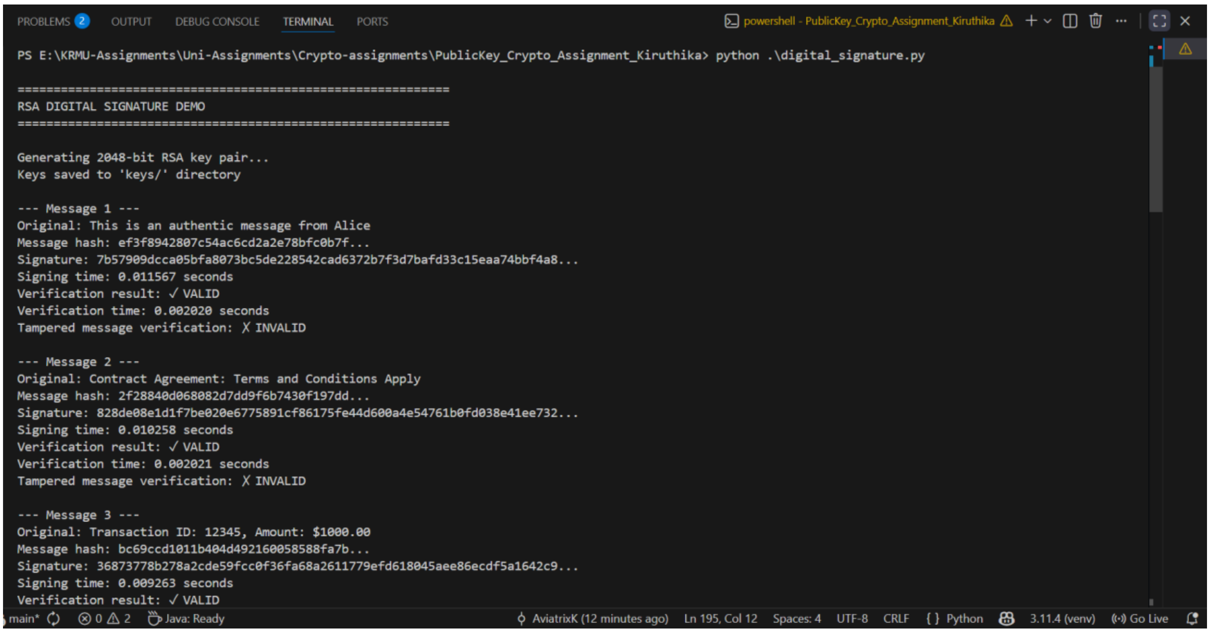
```
   - Blockchain transactions
        """)


def compare_signature_algorithms():
    """Compare RSA and DSA signatures"""
    print("\n" + "="*60)
    print("RSA vs DSA SIGNATURE COMPARISON")
    print("="*60)

    # Test both algorithms
    test_message = b"Performance comparison test
message"

    # RSA Signature
    print("\n--- RSA Signature ---")
    rsa_demo = DigitalSignatureDemo('RSA')
    rsa_demo.generate_keys(2048)

    start = time.time()
    rsa_sig, _ =
rsa_demo.sign_message(test_message)
    rsa_sign_time = time.time() - start

    start = time.time()
```

```
    rsa_demo.verify_signature(test_message, 
rsa_sig)
    rsa_verify_time = time.time() - start

    # DSA Signature
    print("\n--- DSA Signature ---")
    dsa_demo = DigitalSignatureDemo('DSA')
    dsa_demo.generate_keys(2048)

    start = time.time()
    dsa_sig, _ = 
dsa_demo.sign_message(test_message)
    dsa_sign_time = time.time() - start

    start = time.time()
    dsa_demo.verify_signature(test_message, 
dsa_sig)
    dsa_verify_time = time.time() - start

    # Results
    print("\n" + "="*60)
    print("PERFORMANCE RESULTS")
    print("="*60)
    print(f"RSA Signing Time: {rsa_sign_time:.6f} 
seconds")
```

```
    print(f"RSA Verification Time:
{rsa_verify_time:.6f} seconds")
    print(f"RSA Signature Size: {len(rsa_sig)}
bytes")
    print(f"\nDSA Signing Time: {dsa_sign_time:.6f}
seconds")
    print(f"DSA Verification Time:
{dsa_verify_time:.6f} seconds")
    print(f"DSA Signature Size: {len(dsa_sig)}
bytes")


def main():
    """Main execution function"""
    # RSA Digital Signature Demo
    rsa_demo = DigitalSignatureDemo('RSA')
    rsa_demo.demo()

    # DSA Digital Signature Demo
    dsa_demo = DigitalSignatureDemo('DSA')
    dsa_demo.demo()

    # Properties demonstration
    SignatureProperties.demonstrate_properties()

    # Algorithm comparison
    compare_signature_algorithms()
```

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

Output :



```
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika> python .\digital_signature.py

============================================================
RSA DIGITAL SIGNATURE DEMO
============================================================

Generating 2048-bit RSA key pair...
Keys saved to 'keys/' directory

--- Message 1 ---
Original: This is an authentic message from Alice
Message hash: ef3f8942807c54ac6cd2a2e78bfc0b7f...
Signature: 7b57909dcca05bfa8073bc5de228542cad6372b7f3d7bafd33c15eaa74bbf4a8...
Signing time: 0.011567 seconds
Verification result: ✓ VALID
Verification time: 0.002020 seconds
Tampered message verification: ✗ INVALID

--- Message 2 ---
Original: Contract Agreement: Terms and Conditions Apply
Message hash: 2f28840d068082d7dd9f6b7430f197dd...
Signature: 828de08e1d1f7be020e6775891cf86175fe44d600a4e54761b0fd038e41ee732...
Signing time: 0.010258 seconds
Verification result: ✓ VALID
Verification time: 0.002021 seconds
Tampered message verification: ✗ INVALID

--- Message 3 ---
Original: Transaction ID: 12345, Amount: $1000.00
Message hash: bc69ccd1011b404d492160058588fa7b...
Signature: 36873778b278a2cde59fcc0f36fa68a2611779efd618045aee86ecdf5a1642c9...
Signing time: 0.009263 seconds
Verification result: ✓ VALID
```

ecc_vs_dh.py

```
"""
Q4: Key Exchange Protocol Comparison - DH vs ECDH
Author: Your Name
Date: 2024
"""


import time
import random
import hashlib
import os
from tinyec import registry
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric
import dh, ec
from cryptography.hazmat.backends import
default_backend


# Try to import matplotlib, but don't fail if it's
not available
try:
    import matplotlib.pyplot as plt
    import numpy as np
    MATPLOTLIB_AVAILABLE = True
except ImportError:
```

```
    MATPLOTLIB_AVAILABLE = False
    print("Note: matplotlib not found. Graphs will
not be generated.")
    print("Install with: pip install matplotlib
numpy\n")


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


    def __init__(self, key_size=2048):
        self.key_size = key_size
        self.parameters = None
        self.generate_parameters()


    def generate_parameters(self):
        """Generate DH parameters"""
        print(f"Generating {self.key_size}-bit DH
parameters...")
        self.parameters = dh.generate_parameters(
            generator=2,
            key_size=self.key_size,
            backend=default_backend()
        )


    def generate_private_key(self):
```

```
        """Generate private key"""
        # Fixed: removed backend parameter
        return
self.parameters.generate_private_key()

    def perform_key_exchange(self):
        """Perform complete key exchange"""
        # Alice generates private key
        alice_private = self.generate_private_key()
        alice_public = alice_private.public_key()

        # Bob generates private key
        bob_private = self.generate_private_key()
        bob_public = bob_private.public_key()

        # Exchange and compute shared secrets
        alice_shared =
alice_private.exchange(bob_public)
        bob_shared =
bob_private.exchange(alice_public)

        # Derive keys from shared secrets
        alice_key =
hashlib.sha256(alice_shared).hexdigest()
        bob_key =
hashlib.sha256(bob_shared).hexdigest()
```

```
        return alice_key, bob_key,
len(alice_shared) * 8


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


    def __init__(self, curve_name='secp256r1'):
        self.curve_name = curve_name


        # Map curve names for different libraries
        if curve_name == 'secp256r1':
            self.crypto_curve = ec.SECP256R1()
            self.tinyec_curve =
registry.get_curve('secp256r1')
        elif curve_name == 'secp384r1':
            self.crypto_curve = ec.SECP384R1()
            self.tinyec_curve =
registry.get_curve('secp384r1')
        else:
            self.crypto_curve = ec.SECP521R1()
            self.tinyec_curve =
registry.get_curve('secp521r1')


    def generate_private_key(self):
```

```
        """Generate private key"""
        # Check cryptography version and use appropriate method
        try:
            # Try newer version first (without backend)
            return ec.generate_private_key(self.crypto_curve)
        except TypeError:
            # Fall back to older version (with backend)
            return ec.generate_private_key(self.crypto_curve, backend=default_backend())

    def perform_key_exchange(self):
        """Perform complete ECDH key exchange"""
        # Alice generates private key
        alice_private = self.generate_private_key()
        alice_public = alice_private.public_key()

        # Bob generates private key
        bob_private = self.generate_private_key()
        bob_public = bob_private.public_key()

        # Exchange and compute shared secrets
```

```
        alice_shared =
alice_private.exchange(ec.ECDH(), bob_public)
        bob_shared =
bob_private.exchange(ec.ECDH(), alice_public)

        # Derive keys from shared secrets
        alice_key =
hashlib.sha256(alice_shared).hexdigest()
        bob_key =
hashlib.sha256(bob_shared).hexdigest()

        return alice_key, bob_key,
len(alice_shared) * 8

    def perform_tinyec_exchange(self):
        """Perform ECDH using tinyec library"""
        # Alice's keys
        alice_private = random.randint(1,
self.tinyec_curve.field.n - 1)
        alice_public = alice_private *
self.tinyec_curve.g

        # Bob's keys
        bob_private = random.randint(1,
self.tinyec_curve.field.n - 1)
        bob_public = bob_private *
self.tinyec_curve.g
```

```
        # Compute shared secrets
        alice_shared = alice_private * bob_public
        bob_shared = bob_private * alice_public

        # Derive keys
        alice_key =
hashlib.sha256(str(alice_shared.x).encode()).hexdig
est()
        bob_key =
hashlib.sha256(str(bob_shared.x).encode()).hexdiges
t()

        return alice_key, bob_key

class SimpleDiffieHellman:
    """Simple DH implementation for demonstration
(faster)"""

    def __init__(self):
        # Using smaller pre-generated safe prime
for demonstration
        self.p =
0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD12
9024E088A67CC74020BBEA63B139B22514A08798E3404DDEF95
19B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B57
6625E7EC6F44C42E9A63637ED6B0BFF5CB6F406B7EDEE386BFB
5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A16
```

```
3BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD
961C62F356208552BB9ED529077096966D670C354E4ABC9804F
1746C08CA18217C32905E462E36CE3BE39E772C180E86039B27
83A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497
CEA956AE515D2261898FA051015728E5A8AACAA68FFFFFFFFF
FFFFFF
        self.g = 2

    def perform_key_exchange(self):
        """Perform simple DH key exchange"""
        # Alice's keys
        alice_private = random.randint(2, self.p -
2)
        alice_public = pow(self.g, alice_private,
self.p)

        # Bob's keys
        bob_private = random.randint(2, self.p - 2)
        bob_public = pow(self.g, bob_private,
self.p)

        # Compute shared secrets
        alice_shared = pow(bob_public,
alice_private, self.p)
        bob_shared = pow(alice_public, bob_private,
self.p)
```

```
        # Convert to hex keys
        alice_key =
hashlib.sha256(str(alice_shared).encode()).hexdiges
t()
        bob_key =
hashlib.sha256(str(bob_shared).encode()).hexdigest(
)

        return alice_key, bob_key

class ProtocolComparison:
    """Compare DH and ECDH protocols"""

    def __init__(self):
        self.results = {
            'dh': {'times': [], 'key_sizes': []},
            'ecdh': {'times': [], 'curves': []}
        }

    def benchmark_classic_dh_simple(self,
iterations=10):
        """Benchmark with simple DH implementation
(faster)"""
        print("\n" + "="*60)
        print("BENCHMARKING CLASSIC DIFFIE-HELLMAN
(Simplified)")
        print("="*60)
```

```
        print("\nUsing pre-generated 2048-bit safe
prime for faster execution")

        simple_dh = SimpleDiffieHellman()

        times = []
        for i in range(iterations):
            start = time.time()
            alice_key, bob_key =
simple_dh.perform_key_exchange()
            end = time.time()
            times.append(end - start)

            if i == 0:  # Verify only once
                assert alice_key == bob_key, "Key
mismatch!"

        avg_time = sum(times) / len(times)
        self.results['dh']['times'].append(avg_time
)
        self.results['dh']['key_sizes'].append(2048
)

        print(f"  Average time: {avg_time:.4f}
seconds")
```

```
        print(f"  Keys match: ✓")

    def benchmark_classic_dh(self, iterations=10):
        """Benchmark classic Diffie-Hellman"""
        print("\n" + "="*60)
        print("BENCHMARKING CLASSIC DIFFIE-
HELLMAN")
        print("="*60)

        # Use only one key size for faster
execution
        key_sizes = [2048]  # Reduced for much
faster execution

        for key_size in key_sizes:
            print(f"\nTesting {key_size}-bit
DH...")

            try:
                dh_instance =
ClassicDiffieHellman(key_size)

                times = []
                for i in range(min(iterations,
2)):  # Limit to 2 iterations for DH
                    start = time.time()
```

```
                    alice_key, bob_key, shared_size
= dh_instance.perform_key_exchange()
                    end = time.time()
                    times.append(end - start)

                    if i == 0:  # Verify only once
                        assert alice_key ==
bob_key, "Key mismatch!"

                avg_time = sum(times) / len(times)
                self.results['dh']['times'].append(
avg_time)
                self.results['dh']['key_sizes'].app
end(key_size)

                print(f"  Average time:
{avg_time:.4f} seconds")
                print(f"  Shared secret size:
{shared_size} bits")
                print(f"  Keys match: ✓")

            except Exception as e:
                print(f"  Error with {key_size}-bit
DH: {e}")
                print("  Using simplified DH
instead...")
```

```
                self.benchmark_classic_dh_simple(it
erations)
                break

    def benchmark_ecdh(self, iterations=10):
        """Benchmark Elliptic Curve Diffie-
Hellman"""
        print("\n" + "="*60)
        print("BENCHMARKING ELLIPTIC CURVE DIFFIE-
HELLMAN")
        print("="*60)

        curves = [
            ('secp256r1', 256),
            ('secp384r1', 384),
            ('secp521r1', 521)
        ]

        for curve_name, curve_bits in curves:
            print(f"\nTesting {curve_name}
({curve_bits} bits)...")
            ecdh_instance =
EllipticCurveDiffieHellman(curve_name)

            times = []
            for i in range(iterations):
```

```
                start = time.time()
                alice_key, bob_key, shared_size = 
ecdh_instance.perform_key_exchange()
                end = time.time()
                times.append(end - start)

                if i == 0:  # Verify only once
                    assert alice_key == bob_key, 
"Key mismatch!"

            avg_time = sum(times) / len(times)
            self.results['ecdh']['times'].append(av
g_time)
            self.results['ecdh']['curves'].append(c
urve_name)

            print(f"  Average time: {avg_time:.4f} 
seconds")
            print(f"  Shared secret size: 
{shared_size} bits")
            print(f"  Keys match: ✓")

    def compare_security_levels(self):
        """Compare security levels of DH and 
ECDH"""
        print("\n" + "="*60)
```

```
        print("SECURITY LEVEL COMPARISON")
        print("="*60)

        comparison = """
        ┌─────────────────┬─────────────────┬────────
────────────────┐
        │ Security Level  │ DH Key Size     │ ECC
Key Size        │
        ├─────────────────┼─────────────────┼────────
────────────────┤
        │ 80 bits         │ 1024 bits       │ 160-
223 bits        │
        │ 112 bits        │ 2048 bits       │ 224-
255 bits        │
        │ 128 bits        │ 3072 bits       │ 256-
383 bits        │
        │ 192 bits        │ 7680 bits       │ 384-
511 bits        │
        │ 256 bits        │ 15360 bits      │ 512+
bits            │
        └─────────────────┴─────────────────┴────────
────────────────┘

        Key Observations:
        • ECC provides equivalent security with
much smaller key sizes
        • 256-bit ECC ≈ 3072-bit DH in terms of
security
```

```
        • Smaller keys = faster computation, less bandwidth, less storage
        """
        print(comparison)

    def display_results_table(self):
        """Display results in a table format instead of plot"""
        print("\n" + "="*60)
        print("PERFORMANCE RESULTS TABLE")
        print("="*60)

        if self.results['dh']['times']:
            # DH Results
            print("\nClassic Diffie-Hellman Performance:")
            print("┌───────────────┬───────────────────┐")
            print("│ Key Size      │ Avg Time (sec)    │")
            print("├───────────────┼───────────────────┤")
            for size, time_val in zip(self.results['dh']['key_sizes'], self.results['dh']['times']):
                print(f"│ {size:4d} bits   │ {time_val:8.4f}         │")
```

```
            print("└──────────────┴──────────────────┘")

        if self.results['ecdh']['times']:
            # ECDH Results
            print("\nElliptic Curve Diffie-Hellman Performance:")
            print("┌──────────────┬──────────────────┐")

if __name__ == "__main__":
    pc = ProtocolComparison()
    pc.benchmark_classic_dh()
    pc.benchmark_ecdh()
    pc.compare_security_levels()
    pc.display_results_table()
```

Output :

```
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika> python .\ecc_vs_dh.py

============================================================
BENCHMARKING CLASSIC DIFFIE-HELLMAN
============================================================

Testing 2048-bit DH...
Generating 2048-bit DH parameters...
  Average time: 0.0209 seconds
  Shared secret size: 2048 bits
  Keys match: ✓

============================================================
BENCHMARKING ELLIPTIC CURVE DIFFIE-HELLMAN
============================================================

Testing secp256r1 (256 bits)...
  Average time: 0.0006 seconds
  Shared secret size: 256 bits
  Keys match: ✓

Testing secp384r1 (384 bits)...
  Average time: 0.0097 seconds
  Shared secret size: 384 bits
  Keys match: ✓

Testing secp521r1 (521 bits)...
  Average time: 0.0243 seconds
  Shared secret size: 528 bits
  Keys match: ✓

============================================================
SECURITY LEVEL COMPARISON
```

```
SECURITY LEVEL COMPARISON
============================================================

        | Security Level | DH Key Size | ECC Key Size |
        |----------------|-------------|--------------|
        | 80 bits        | 1024 bits   | 160-223 bits |
        | 112 bits       | 2048 bits   | 224-255 bits |
        | 128 bits       | 3072 bits   | 256-383 bits |
        | 192 bits       | 7680 bits   | 384-511 bits |
        | 256 bits       | 15360 bits  | 512+ bits    |

        Key Observations:
        • ECC provides equivalent security with much smaller key sizes
        • 256-bit ECC ≈ 3072-bit DH in terms of security
        • Smaller keys = faster computation, less bandwidth, less storage


============================================================
PERFORMANCE RESULTS TABLE
============================================================

Classic Diffie-Hellman Performance:
| Key Size  | Avg Time (sec) |
|-----------|----------------|
| 2048 bits | 0.0209         |

Elliptic Curve Diffie-Hellman Performance:

PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika>
```

pki_demo.py

```python
"""
Q3: Public Key Infrastructure (PKI) Demonstration
Author: Your Name
Date: 2024
"""

import os
import datetime
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

class PKIDemo:
    """PKI Certificate Generation and Validation"""

    def __init__(self):
        self.ca_key = None
        self.ca_cert = None
        os.makedirs('certificates', exist_ok=True)
```

```
    def generate_ca_certificate(self):
        """Generate a Certificate Authority (CA)
certificate"""
        print("\n" + "="*60)
        print("GENERATING CERTIFICATE AUTHORITY
(CA)")
        print("="*60)

        # Generate CA private key
        self.ca_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=4096,
            backend=default_backend()
        )

        # CA certificate details
        subject = issuer = x509.Name([
            x509.NameAttribute(NameOID.COUNTRY_NAME
, "IN"),
            x509.NameAttribute(NameOID.STATE_OR_PRO
VINCE_NAME, "Delhi"),
            x509.NameAttribute(NameOID.LOCALITY_NAM
E, "New Delhi"),
            x509.NameAttribute(NameOID.ORGANIZATION
_NAME, "Demo Root CA"),
```

```
            x509.NameAttribute(NameOID.ORGANIZATION
AL_UNIT_NAME, "Certificate Authority"),
            x509.NameAttribute(NameOID.COMMON_NAME,
"Demo Root CA"),
        ])

        # Create 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_numbe
r())
            .not_valid_before(datetime.datetime.utc
now())
            .not_valid_after(datetime.datetime.utcn
ow() + datetime.timedelta(days=3650))
            .add_extension(
                x509.BasicConstraints(ca=True,
path_length=None),
                critical=True,
            )
            .add_extension(
                x509.KeyUsage(
                    digital_signature=True,
```

```
                    content_commitment=False,
                    key_encipherment=False,
                    data_encipherment=False,
                    key_agreement=False,
                    key_cert_sign=True,
                    crl_sign=True,
                    encipher_only=False,
                    decipher_only=False,
                ),
                critical=True,
            )
            .add_extension(
                x509.SubjectKeyIdentifier.from_publ
ic_key(self.ca_key.public_key()),
                critical=False,
            )
            .sign(self.ca_key, hashes.SHA256(),
backend=default_backend())
        )

        # Save CA certificate and key
        with open('certificates/ca_cert.pem', 'wb')
as f:
            f.write(self.ca_cert.public_bytes(seria
lization.Encoding.PEM))
```

```
        with open('certificates/ca_key.pem', 'wb')
as f:
            f.write(self.ca_key.private_bytes(
                encoding=serialization.Encoding.PEM
,
                format=serialization.PrivateFormat.
TraditionalOpenSSL,
                encryption_algorithm=serialization.
NoEncryption()
            ))

        print("✓ CA Certificate generated and
saved")
        print(f"  Serial Number:
{self.ca_cert.serial_number}")
        print(f"  Valid From:
{self.ca_cert.not_valid_before}")
        print(f"  Valid Until:
{self.ca_cert.not_valid_after}")

    def generate_self_signed_certificate(self,
domain="example.com"):
        """Generate a self-signed X.509
certificate"""
        print("\n" + "="*60)
        print("GENERATING SELF-SIGNED CERTIFICATE")
        print("="*60)
```

```
        # Generate private key
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )

        # 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.ORGANIZATIONAL_UNIT_NAME, "IT Department"),
            x509.NameAttribute(NameOID.COMMON_NAME, domain),
        ])

        # Create certificate
        cert = (
```

```
            x509.CertificateBuilder()
            .subject_name(subject)
            .issuer_name(issuer)
            .public_key(private_key.public_key())
            .serial_number(x509.random_serial_numbe
r())
            .not_valid_before(datetime.datetime.utc
now())
            .not_valid_after(datetime.datetime.utcn
ow() + datetime.timedelta(days=365))
            .add_extension(
                x509.SubjectAlternativeName([
                    x509.DNSName(domain),
                    x509.DNSName(f"www.{domain}"),
                    x509.DNSName("localhost"),
                ]),
                critical=False,
            )
            .add_extension(
                x509.BasicConstraints(ca=False,
path_length=None),
                critical=True,
            )
            .sign(private_key, hashes.SHA256(),
backend=default_backend())
        )
```

```
        # Save certificate and key
        with
open(f'certificates/self_signed_{domain}.pem',
'wb') as f:
            f.write(cert.public_bytes(serialization
.Encoding.PEM))

        with
open(f'certificates/self_signed_{domain}_key.pem',
'wb') as f:
            f.write(private_key.private_bytes(
                encoding=serialization.Encoding.PEM
,
                format=serialization.PrivateFormat.
TraditionalOpenSSL,
                encryption_algorithm=serialization.
NoEncryption()
            ))

        print(f"✓ Self-signed certificate for
{domain} generated")
        self.display_certificate_info(cert)

        return cert, private_key
```

```
    def generate_ca_signed_certificate(self, 
domain="secure.example.com"):
        """Generate a certificate signed by CA"""
        if not self.ca_key or not self.ca_cert:
            self.generate_ca_certificate()

        print("\n" + "="*60)
        print("GENERATING CA-SIGNED CERTIFICATE")
        print("="*60)

        # Generate private key for the certificate
        cert_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )

        # Certificate subject
        subject = x509.Name([
            x509.NameAttribute(NameOID.COUNTRY_NAME
, "IN"),
            x509.NameAttribute(NameOID.STATE_OR_PRO
VINCE_NAME, "Karnataka"),
            x509.NameAttribute(NameOID.LOCALITY_NAM
E, "Bangalore"),
```

```
            x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Demo Company"),
            x509.NameAttribute(NameOID.ORGANIZATIONAL_UNIT_NAME, "Web Services"),
            x509.NameAttribute(NameOID.COMMON_NAME, domain),
        ])

        # Create certificate signed by CA
        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(datetime.datetime.utcnow())
            .not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=365))
            .add_extension(
                x509.SubjectAlternativeName([
                    x509.DNSName(domain),
                    x509.DNSName(f"*.{domain}"),
                ]),
                critical=False,
```

```
            )
            .add_extension(
                x509.BasicConstraints(ca=False,
path_length=None),
                critical=True,
            )
            .add_extension(
                x509.AuthorityKeyIdentifier.from_is
suer_public_key(self.ca_key.public_key()),
                critical=False,
            )
            .sign(self.ca_key, hashes.SHA256(),
backend=default_backend())
        )

        # Save certificate
        with
open(f'certificates/ca_signed_{domain}.pem', 'wb')
as f:
            f.write(cert.public_bytes(serialization
.Encoding.PEM))

        print(f"✓ CA-signed certificate for
{domain} generated")
        self.display_certificate_info(cert)
```

```
        return cert

    def display_certificate_info(self, cert):
        """Display certificate information"""
        print("\nCertificate Details:")
        print(f"  Subject:
{cert.subject.rfc4514_string()}")
        print(f"  Issuer:
{cert.issuer.rfc4514_string()}")
        print(f"  Serial Number:
{cert.serial_number}")
        print(f"  Not Valid Before:
{cert.not_valid_before}")
        print(f"  Not Valid After:
{cert.not_valid_after}")
        print(f"  Signature Algorithm:
{cert.signature_algorithm_oid}")

    def validate_certificate_chain(self, cert):
        """Validate certificate chain"""
        print("\n" + "="*60)
        print("CERTIFICATE CHAIN VALIDATION")
        print("="*60)

        # Check if self-signed
        if cert.issuer == cert.subject:
```

```
            print("⚠ Certificate is self-signed")
            print("  - No chain validation possible")
            print("  - Trust must be explicitly established")
        else:
            print("✓ Certificate is CA-signed")
            print("  - Issuer verified against CA certificate")
            print("  - Chain of trust established")

        # Check validity period
        now = datetime.datetime.utcnow()
        if cert.not_valid_before <= now <= cert.not_valid_after:
            print("✓ Certificate is within validity period")
        else:
            print("✗ Certificate is expired or not yet valid")

    def demonstrate_crl(self):
        """Demonstrate Certificate Revocation List concept"""
        print("\n" + "="*60)
        print("CERTIFICATE REVOCATION LIST (CRL)")
```

```
        print("="*60)

        print("""
CRL Components and Process:
1. CRL Generation:
   - CA maintains list of revoked certificates
   - Each entry contains serial number and
revocation date
   - CRL is signed by CA for authenticity

2. Revocation Reasons:
   - Key compromise
   - CA compromise
   - Affiliation changed
   - Certificate superseded
   - Cessation of operation

3. CRL Distribution:
   - Published at regular intervals
   - Available via HTTP/LDAP
   - Cached by clients

4. OCSP (Online Certificate Status Protocol):
   - Real-time certificate validation
   - More efficient than downloading full CRL
```

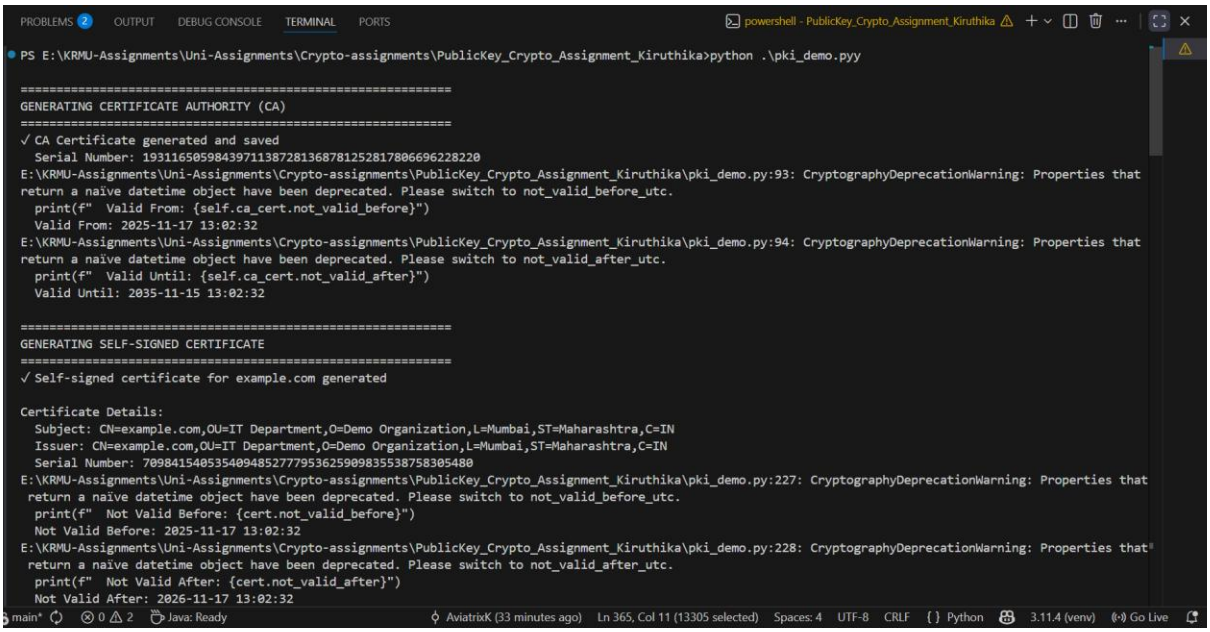
```
    - Returns: Good, Revoked, or Unknown
        """)

class PKITrustModels:
    """Demonstrate PKI Trust Models"""

    @staticmethod
    def explain_trust_models():
        print("\n" + "="*60)
        print("PKI TRUST MODELS")
        print("="*60)

        print("""
1. HIERARCHICAL TRUST MODEL:
   Root CA
      ├── Intermediate CA 1
      │     ├── End Entity Certificate 1
      │     └── End Entity Certificate 2
      └── Intermediate CA 2
            └── End Entity Certificate 3

   - Used by: Web PKI, Enterprise PKI
   - Single root of trust
   - Clear chain of authority
```

```
2. WEB OF TRUST MODEL:
   User A ↔ User B
      ↑        ↓
   User D ↔ User C

   - Used by: PGP/GPG
   - Decentralized trust
   - Users sign each other's keys

3. BRIDGE CA MODEL:
   CA1 ↔ Bridge CA ↔ CA2
            ↓
           CA3

   - Cross-certification between CAs
   - Enables inter-organizational trust

4. BROWSER CERTIFICATE VALIDATION:
   a) Check certificate chain to trusted root
   b) Verify certificate not expired
   c) Check certificate not revoked (CRL/OCSP)
   d) Verify domain name matches
   e) Check certificate constraints
   f) Validate signature algorithms
        """)
```

```
def main():
    """Main execution function"""
    # Initialize PKI Demo
    pki = PKIDemo()

    # Generate CA certificate
    pki.generate_ca_certificate()

    # Generate self-signed certificate
    self_signed_cert, _ =
pki.generate_self_signed_certificate("example.com")

    # Generate CA-signed certificate
    ca_signed_cert =
pki.generate_ca_signed_certificate("secure.example.
com")

    # Validate certificates
    pki.validate_certificate_chain(self_signed_cert
)
    pki.validate_certificate_chain(ca_signed_cert)

    # Demonstrate CRL
    pki.demonstrate_crl()
```

```
    # Explain trust models
    PKITrustModels.explain_trust_models()

    print("\n" + "="*60)
    print("PKI DEMONSTRATION COMPLETE")
    print("="*60)
    print("\nGenerated files in 'certificates/' directory:")
    for file in os.listdir('certificates'):
        print(f"  - {file}")

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

Output:



```
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika>python .\pki_demo.pyy

============================================================
GENERATING CERTIFICATE AUTHORITY (CA)
============================================================
✓ CA Certificate generated and saved
  Serial Number: 19311650598439711387281368781252817806696228220
E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika\pki_demo.py:93: CryptographyDeprecationWarning: Properties that return a naïve datetime object have been deprecated. Please switch to not_valid_before_utc.
  print(f"  Valid From: {self.ca_cert.not_valid_before}")
  Valid From: 2025-11-17 13:02:32
E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika\pki_demo.py:94: CryptographyDeprecationWarning: Properties that return a naïve datetime object have been deprecated. Please switch to not_valid_after_utc.
  print(f"  Valid Until: {self.ca_cert.not_valid_after}")
  Valid Until: 2035-11-15 13:02:32

============================================================
GENERATING SELF-SIGNED CERTIFICATE
============================================================
✓ Self-signed certificate for example.com generated

Certificate Details:
  Subject: CN=example.com,OU=IT Department,O=Demo Organization,L=Mumbai,ST=Maharashtra,C=IN
  Issuer: CN=example.com,OU=IT Department,O=Demo Organization,L=Mumbai,ST=Maharashtra,C=IN
  Serial Number: 7098415405354094852777953625909835538758305480
E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika\pki_demo.py:227: CryptographyDeprecationWarning: Properties that return a naïve datetime object have been deprecated. Please switch to not_valid_before_utc.
  print(f"  Not Valid Before: {cert.not_valid_before}")
  Not Valid Before: 2025-11-17 13:02:32
E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika\pki_demo.py:228: CryptographyDeprecationWarning: Properties that return a naïve datetime object have been deprecated. Please switch to not_valid_after_utc.
  print(f"  Not Valid After: {cert.not_valid_after}")
  Not Valid After: 2026-11-17 13:02:32
```

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS        powershell - PublicKey_Crypto_Assignment_Kiruthika ⚠
  Signature Algorithm: <ObjectIdentifier(oid=1.2.840.113549.1.1.11, name=sha256WithRSAEncryption)>

============================================================
GENERATING CA-SIGNED CERTIFICATE
============================================================
✓ CA-signed certificate for secure.example.com generated

Certificate Details:
  Subject: CN=secure.example.com,OU=Web Services,O=Demo Company,L=Bangalore,ST=Karnataka,C=IN
  Issuer: CN=Demo Root CA,OU=Certificate Authority,O=Demo Root CA,L=New Delhi,ST=Delhi,C=IN
  Serial Number: 226295709056426918265411538828455874487700202415
  Not Valid Before: 2025-11-17 13:02:32
  Not Valid After: 2026-11-17 13:02:32
  Signature Algorithm: <ObjectIdentifier(oid=1.2.840.113549.1.1.11, name=sha256WithRSAEncryption)>

============================================================
CERTIFICATE CHAIN VALIDATION
============================================================
⚠ Certificate is self-signed
  - No chain validation possible
  - Trust must be explicitly established
E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika\pki_demo.py:249: CryptographyDeprecationWarning: Properties that
 return a naïve datetime object have been deprecated. Please switch to not_valid_before_utc.
  if cert.not_valid_before <= now <= cert.not_valid_after:
E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika\pki_demo.py:249: CryptographyDeprecationWarning: Properties that
 return a naïve datetime object have been deprecated. Please switch to not_valid_after_utc.
  if cert.not_valid_before <= now <= cert.not_valid_after:
✓ Certificate is within validity period

============================================================
CERTIFICATE CHAIN VALIDATION
============================================================
✓ Certificate is CA-signed
main*   ⊗ 0 ⚠ 2   Java: Ready   AviatrixK (33 minutes ago)   Ln 365, Col 11 (13305 selected)   Spaces: 4   UTF-8   CRLF   { } Python   3.11.4 (venv)   Go Live
```

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS        powershell - PublicKey_Crypto_Assignment_Kiruthika ⚠
  - Issuer verified against CA certificate
  - Chain of trust established
✓ Certificate is within validity period

============================================================
CERTIFICATE REVOCATION LIST (CRL)
============================================================

CRL Components and Process:
1. CRL Generation:
   - CA maintains list of revoked certificates
   - Each entry contains serial number and revocation date
   - CRL is signed by CA for authenticity

2. Revocation Reasons:
   - Key compromise
   - CA compromise
   - Affiliation changed
   - Certificate superseded
   - Cessation of operation

3. CRL Distribution:
   - Published at regular intervals
   - Available via HTTP/LDAP
   - Cached by clients

4. OCSP (Online Certificate Status Protocol):
   - Real-time certificate validation
   - More efficient than downloading full CRL
   - Returns: Good, Revoked, or Unknown

============================================================
main*   ⊗ 0 ⚠ 2   Java: Ready   AviatrixK (33 minutes ago)   Ln 365, Col 11 (13305 selected)   Spaces: 4   UTF-8   CRLF   { } Python   3.11.4 (venv)   Go Live
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell - PublicKey_Crypto_Assignment_Kiruthika

```
PKI TRUST MODELS
============================================================

1. HIERARCHICAL TRUST MODEL:
   Root CA
      ├── Intermediate CA 1
      │     ├── End Entity Certificate 1
      │     └── End Entity Certificate 2
      └── Intermediate CA 2
            └── End Entity Certificate 3

   - Used by: Web PKI, Enterprise PKI
   - Single root of trust
   - Clear chain of authority

2. WEB OF TRUST MODEL:
   User A ↔ User B
      ↑        ↓
   User D ↔ User C

   - Used by: PGP/GPG
   - Decentralized trust
   - Users sign each other's keys

3. BRIDGE CA MODEL:
   CA1 ↔ Bridge CA ↔ CA2
               ↓
              CA3

   - Cross-certification between CAs
   - Enables inter-organizational trust

4. BROWSER CERTIFICATE VALIDATION:
```

main* ⊗ 0 ⚠ 2 Java: Ready AviatrixK (33 minutes ago) Ln 365, Col 11 (13305 selected) Spaces: 4 UTF-8 CRLF { } Python 3.11.4 (venv) Go Live

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell - PublicKey_Crypto_Assignment_Kiruthika

```
4. BROWSER CERTIFICATE VALIDATION:
   a) Check certificate chain to trusted root
   b) Verify certificate not expired
   c) Check certificate not revoked (CRL/OCSP)
   d) Verify domain name matches
   e) Check certificate constraints
   f) Validate signature algorithms


============================================================
PKI DEMONSTRATION COMPLETE
============================================================

Generated files in 'certificates/' directory:
  - ca_cert.pem
  - ca_key.pem
  - ca_signed_secure.example.com.pem
  - self_signed_example.com.pem
  - self_signed_example.com_key.pem
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika>
```

rsa_dh_simulation.py

```
"""
Q1: RSA Encryption/Decryption and Diffie-Hellman
Key Exchange
Author: Your Name
Date: 2024
"""


import os
import random
import hashlib
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes
import time


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


    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"Generating {self.key_size}-bit RSA
key pair...")
        self.private_key =
RSA.generate(self.key_size)
        self.public_key =
self.private_key.publickey()

        # Save keys to files
        os.makedirs('keys', exist_ok=True)

        with open('keys/rsa_private.pem', 'wb') as
f:
            f.write(self.private_key.export_key())

        with open('keys/rsa_public.pem', 'wb') as
f:
            f.write(self.public_key.export_key())

        print("Keys generated and saved to 'keys/'
directory")

    def encrypt(self, message):
        """Encrypt message using public key"""
        cipher_rsa =
PKCS1_OAEP.new(self.public_key)
```

```
        ciphertext = cipher_rsa.encrypt(message)
        return ciphertext

    def decrypt(self, ciphertext):
        """Decrypt message using private key"""
        cipher_rsa =
PKCS1_OAEP.new(self.private_key)
        plaintext = cipher_rsa.decrypt(ciphertext)
        return plaintext

    def demo(self):
        """Run RSA encryption/decryption demo"""
        print("\n" + "="*60)
        print("RSA ENCRYPTION/DECRYPTION DEMO")
        print("="*60)

        # Generate keys
        self.generate_keys()

        # Test message
        message = b"This is a secure message
encrypted with RSA!"
        print(f"\nOriginal message:
{message.decode()}")
```

```
        # Encrypt
        start_time = time.time()
        ciphertext = self.encrypt(message)
        encrypt_time = time.time() - start_time
        print(f"Encrypted (hex): 
{ciphertext.hex()[:80]}...")
        print(f"Encryption time: {encrypt_time:.6f} 
seconds")

        # Decrypt
        start_time = time.time()
        decrypted = self.decrypt(ciphertext)
        decrypt_time = time.time() - start_time
        print(f"Decrypted: {decrypted.decode()}")
        print(f"Decryption time: {decrypt_time:.6f} 
seconds")

        return encrypt_time, decrypt_time


class DiffieHellmanDemo:
    """Diffie-Hellman Key Exchange Demonstration"""

    def __init__(self, bits=1024):
        self.bits = bits
        self.p = self._generate_prime()
```

```
        self.g = self._find_generator()

    def _generate_prime(self):
        """Generate a large prime number"""
        # For demonstration, using a pre-selected
safe prime
        # In production, use proper prime
generation
        return
0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD12
9024E088A67CC74020BBEA63B139B22514A08798E3404DDEF95
19B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B57
6625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A
899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163B
F0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD96
1C62F356208552BB9ED529077096966D670C354E4ABC9804F17
46C08CA237327FFFFFFFFFFFFFFFF

    def _find_generator(self):
        """Find a generator for the group"""
        # For demonstration, using a common
generator
        return 2

    def generate_private_key(self):
        """Generate a private key"""
        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, private_key, 
other_public_key):
        """Compute shared secret"""
        return pow(other_public_key, private_key, 
self.p)

    def demo(self):
        """Run Diffie-Hellman key exchange demo"""
        print("\n" + "="*60)
        print("DIFFIE-HELLMAN KEY EXCHANGE DEMO")
        print("="*60)

        print(f"Public parameters:")
        print(f"Prime (p): {hex(self.p)[:50]}...")
        print(f"Generator (g): {self.g}")

        # Alice's keys
        print("\n--- Alice's Side ---")
        start_time = time.time()
        alice_private = self.generate_private_key()
```

```
        alice_public = 
self.generate_public_key(alice_private)
        alice_time = time.time() - start_time
        print(f"Alice's public key: 
{hex(alice_public)[:50]}...")

        # Bob's keys
        print("\n--- Bob's Side ---")
        start_time = time.time()
        bob_private = self.generate_private_key()
        bob_public = 
self.generate_public_key(bob_private)
        bob_time = time.time() - start_time
        print(f"Bob's public key: 
{hex(bob_public)[:50]}...")

        # Compute shared secrets
        print("\n--- Shared Secret Computation ---
")
        alice_shared = 
self.compute_shared_secret(alice_private, 
bob_public)
        bob_shared = 
self.compute_shared_secret(bob_private, 
alice_public)

        # Convert to usable key
```

```
        alice_key =
hashlib.sha256(str(alice_shared).encode()).hexdiges
t()
        bob_key =
hashlib.sha256(str(bob_shared).encode()).hexdigest(
)

        print(f"Alice's shared secret (SHA-256):
{alice_key[:32]}...")
        print(f"Bob's shared secret (SHA-256):
{bob_key[:32]}...")
        print(f"Secrets match: {alice_key ==
bob_key}")
        print(f"\nTotal key exchange time:
{alice_time + bob_time:.6f} seconds")

        return alice_time + bob_time


def compare_algorithms():
    """Compare RSA and Diffie-Hellman"""
    print("\n" + "="*60)
    print("COMPARISON: RSA vs DIFFIE-HELLMAN")
    print("="*60)

    comparison = """
     ┌──────────────────┬────────────────────────────┬
────┬────────────────────┐
```

```
    │ Aspect            │ RSA                            │
Diffie-Hellman             │
    ├───────────────────┼────────────────────────────────┼
───────────────────────────┤
    │ Purpose           │ Encryption & Signatures        │
Key Exchange Only          │
    │ Key Size          │ 2048-4096 bits                 │
2048-4096 bits             │
    │ Security Basis    │ Factoring problem              │
Discrete log problem       │
    │ Speed             │ Slower                         │
Faster                     │
    │ Use Cases         │ Digital certificates,          │
Perfect forward secrecy,   │
    │                   │ Email encryption               │
TLS key exchange           │
    │ Authentication    │ Built-in                       │
Requires additional        │
    └───────────────────┴────────────────────────────────┴
───────────────────────────┘
    """
    print(comparison)


def main():
    """Main execution function"""
    # RSA Demo
    rsa_demo = RSADemo()
```

```
    rsa_encrypt_time, rsa_decrypt_time =
rsa_demo.demo()

    # Diffie-Hellman Demo
    dh_demo = DiffieHellmanDemo()
    dh_time = dh_demo.demo()

    # Comparison
    compare_algorithms()

    # Performance Summary
    print("\n" + "="*60)
    print("PERFORMANCE SUMMARY")
    print("="*60)
    print(f"RSA Encryption Time:
{rsa_encrypt_time:.6f} seconds")
    print(f"RSA Decryption Time:
{rsa_decrypt_time:.6f} seconds")
    print(f"DH Key Exchange Time: {dh_time:.6f}
seconds")

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

Output:

```
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika> python .\rsa_dh_simulation.py

============================================================
RSA ENCRYPTION/DECRYPTION DEMO
============================================================
Generating 2048-bit RSA key pair...
Keys generated and saved to 'keys/' directory

Original message: This is a secure message encrypted with RSA!
Encrypted (hex): 0c94f6837ac1946180874a6898da1c579e3a91f1082ce8b7beb5b4ac46f4d50b7f6a8d23abd25922...
Encryption time: 0.004936 seconds
Decrypted: This is a secure message encrypted with RSA!
Decryption time: 0.010273 seconds

============================================================
DIFFIE-HELLMAN KEY EXCHANGE DEMO
============================================================
Public parameters:
Prime (p): 0xffffffffffffffffc90fdaa22168c234c4c6628b80dc1cd1...
Generator (g): 2

--- Alice's Side ---
Alice's public key: 0xa7fa0009b76df31cd1a2bff4abeaf48a634e5cf9cf6a0d2e...

--- Bob's Side ---
Bob's public key: 0x7a3e5d98b876833b8cd8a945fa15312b8ebc7331b9da7347...

--- Shared Secret Computation ---
Alice's shared secret (SHA-256): 796c748f4b2155dd4f71552205c8b7bd...
Bob's shared secret (SHA-256): 796c748f4b2155dd4f71552205c8b7bd...
Secrets match: True

Total key exchange time: 0.076037 seconds
```

```
============================================================
COMPARISON: RSA vs DIFFIE-HELLMAN
============================================================
```

Aspect	RSA	Diffie-Hellman
Purpose	Encryption & Signatures	Key Exchange Only
Key Size	2048-4096 bits	2048-4096 bits
Security Basis	Factoring problem	Discrete log problem
Speed	Slower	Faster
Use Cases	Digital certificates, Email encryption	Perfect forward secrecy, TLS key exchange
Authentication	Built-in	Requires additional

```
============================================================
PERFORMANCE SUMMARY
============================================================
RSA Encryption Time: 0.004936 seconds
RSA Decryption Time: 0.010273 seconds
DH Key Exchange Time: 0.076037 seconds
PS E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\PublicKey_Crypto_Assignment_Kiruthika>
```