Birzeit University

Faculty of Engineering and Technology Department of Electrical and Computer Engineering

Applied Cryptography- ENCS4320

Homework Assignment #2

First Semester 2025/2026

Students:

Baraa Said- 1220280

Saleh Akram- 1220641

Instructor:

Dr. Ahmed Shawahna

Date:

26-1-2026

AES:

Theoretical:

we use **AES-128**, a symmetric encryption algorithm that works on fixed **16-byte blocks** using a **128-bit (16-byte) key** for both encryption and decryption. We run AES in **CBC (Cipher Block Chaining) mode**, where each plaintext block is combined (XORed) with the previous ciphertext block before encryption; this chaining helps hide patterns and ensures the output changes if any block changes. Because CBC requires the data length to be a multiple of 16 bytes, we apply **PKCS#7 padding**, which adds extra bytes to the end of the message so it fits the block size and can be removed cleanly after decryption. For the first block, CBC uses a **16-byte Initialization Vector (IV)**, and we generate a fresh **random IV for every message**; the IV is not secret, but it must be unique per encryption so that encrypting the same message twice produces different ciphertexts and remains secure.

Code:

```
# ============================================================================
# PHASE I: AES ENCRYPTION FUNCTIONS
# ============================================================================

def generate_aes_key(student_id):
    """Generate AES-128 key from Student ID using MD5 (16 bytes)"""
    return hashlib.md5(student_id.encode()).digest()

def generate_reference_iv(student_id):
    """Generate reference IV from Student ID using SHA256 (first 16 bytes)"""
    return hashlib.sha256(student_id.encode()).digest()[:16]

def encrypt_message(plaintext, key):
    """Encrypt message using AES-128-CBC with PKCS#7 padding and random IV"""
    iv = secrets.token_bytes(16)  # Fresh random IV for each message
    ciphertext = custom_aes.aes_encrypt_cbc(plaintext.encode('utf-8'), key, iv)
    return iv + ciphertext  # IV is prepended to ciphertext

def decrypt_message(encrypted_data, key):
    """Decrypt AES-128-CBC encrypted message"""
    iv = encrypted_data[:16]  # Extract IV from first 16 bytes
    ciphertext = encrypted_data[16:]
    plaintext = custom_aes.aes_decrypt_cbc(ciphertext, key, iv)
    return plaintext.decode('utf-8')
```

Test:

```
============================================================
TESTING PHASE I: AES ENCRYPTION
============================================================
[CRYPTO] Generated AES Key (hex): 11b4ccc48f349a3ae7a6089ab8c88251

[TEST] Original message: Hello, this is a secure message!
[TEST] Encrypted (hex): 3fcbc314e91990b23f03f0b21523b37355f0dad2571c9eea12ff96598756bb48f6b82984e74c505a6948de64ffae9781
175fa2a195e9b357d55b39673663c1af
[TEST] Length: 64 bytes (16 IV + 48 ciphertext)
[TEST] Decrypted message: Hello, this is a secure message!
[TEST] ✓ SUCCESS: Encryption/Decryption working!
```

As we show the decrypted message is equal to the original message so the Encryotion/Decryption is success.

```
[TEST] encrypte with different IV
Ciphertext 1:  3fcbc314e91990b23f03f0b21523b37355f0dad2571c9eea12ff96598756bb48f6b82984e74c505a6948de64ffae9781175fa2a19
5e9b357d55b39673663c1af
Ciphertext 2:  fbf4f0b46e662505c2c02f0696ac00e5bef2915743dc0b5b52d4c4d9ad9f14986df540af781c9b22b23824133a8ef9caa937f036a
c3671b9bbcddc086778a601
[TEST] ✓ SUCCESS: Different IVs produce different ciphertexts!
```

When we use different iv for the same massage that we encrypt and same Key it tells different ciphertext.

```
[TEST] encrypte with same IV
[CRYPTO] Reference IV (hex): 8d3c1f443281cdd69a7e3f0f079cc58d
Ciphertext 1:  8d3c1f443281cdd69a7e3f0f079cc58d6050eb1f384dbd333dfb8e1c5d2bb19b4ee6976579afdd039d3191b3c18891f2fce169fe7
b6c9689b664eb252c5cf1ca
Ciphertext 2:  8d3c1f443281cdd69a7e3f0f079cc58d6050eb1f384dbd333dfb8e1c5d2bb19b4ee6976579afdd039d3191b3c18891f2fce169fe7
b6c9689b664eb252c5cf1ca
[TEST] Same IVs produce same ciphertext
```

When we encrypt the same massage with same IV and same Key its produce same ciphertext as we show.

RSA:

Theoretical:

In our authentication infrastructure, we use **RSA digital signatures (RSA-128)** to solve the key distribution and identity-verification problem: each participant generates an RSA key pair, keeps the **private key** secret, and shares the **public key**. To bind identity, the public key is linked to the participant's **Student ID**, so when a message claims to come from a specific ID, the receiver knows which public key should be used to verify it. When sending data, the sender signs the message (typically a hashed/encoded representation of it) with their private key, and the receiver verifies the signature using the sender's public key before trusting the data. This gives **authentication** (the sender really owns the private key for that Student ID) and **integrity** (any change in the message breaks verification), which prevents impersonation attacks because only the legitimate key owner can produce a valid signature.

Code:

```python
def generate_rsa_keypair():
    """Generate RSA-128 bit key pair for digital signatures"""
    # Generate two 64-bit primes for 128-bit modulus
    p = getPrime(64)
    q = getPrime(64)
    n = p * q
    e = 65537
    phi = (p - 1) * (q - 1)
    d = pow(e, -1, phi)

    pubkey = RSA128Key(n, e)
    privkey = RSA128Key(n, e, d)
    print(f"[RSA] Generated 128-bit key pair (n={n.bit_length()} bits)")
    return (pubkey, privkey)

def sign_data(data, private_key):
    """Sign data using RSA-128 private key with truncated MD5 hash"""
    # Use MD5 hash truncated to 10 bytes (80 bits) to fit in 128-bit RSA
    h = hashlib.md5(data).digest()[:10]
    m = bytes_to_long(h)
    # RSA signature: s = m^d mod n
    sig = pow(m, private_key.d, private_key.n)
    return long_to_bytes(sig, 16)  # 16 bytes = 128 bits

def verify_signature(data, signature, public_key):
    """Verify RSA-128 signature using public key"""
    try:
        # Compute expected hash
        h = hashlib.md5(data).digest()[:10]
        m = bytes_to_long(h)
        # RSA verify: m' = s^e mod n
        sig_int = bytes_to_long(signature)
        m_verify = pow(sig_int, public_key.e, public_key.n)
        return m == m_verify
    except Exception:
        return False
```

Test:

```
============================================================
TESTING PHASE II: RSA SIGNATURES
============================================================
[RSA] Generated 128-bit key pair (n=128 bits)
[RSA] Public key (n=218085904763562227282892839302098235619, e=65537)

[TEST] Data to sign: b'Test message for signing'
[TEST] Signature (hex): 03f23c3b03636191a216ac5da9a933ac
m:  565210997945935015083489
m_verify:  565210997945935015083489
[TEST] ✓ SUCCESS: Signature verified with correct key!
m:  880510933996487530114916
m_verify:  565210997945935015083489
[TEST] ✓ SUCCESS: Signature rejected for wrong data!
```

the output shows a 128-bit RSA key pair being generated and the **public key (n, e)** printed. The program then signs the test message ("Test message for signing") and prints the signature in hex. In the first verification, m and m_verify match and you get **"Signature verified with correct key!"**, meaning the signature is valid for the original message. In the second test, the data is changed (different m), while the verified value stays the same, so the program correctly prints **"Signature rejected for wrong data!"**—this proves your verification step detects tampering and only accepts signatures that correspond to the exact original content.

DH:

Theoretical:

The **Diffie–Hellman (DH)** protocol allows two parties to establish a **shared secret** without ever transmitting that secret over the network. First, both sides agree on public parameters: a large prime p and a generator g. Each party then chooses a random private value (e.g., a and b) and keeps it secret, computes a public value, and sends it to the other side: A=g^a mod p and B=g^b mod p . After exchanging A and B, both parties independently compute the same shared secret: one computes s=B^a mod p while the other computes s=A^b mod p, and these are equal due to the algebraic structure of the group. The key security idea is that an eavesdropper who only sees p, g, A, and B cannot feasibly recover a or b (and therefore cannot compute s)**.**

Code:

```python
class DHKeyExchange:
    """Diffie-Hellman key exchange for Perfect Forward Secrecy"""

    def __init__(self):
        # Generate random private value (ephemeral - new each session)
        self.private_value = secrets.randbelow(DH_PRIME - 2) + 1
        # Compute public value: g^private mod p
        self.public_value = pow(DH_GENERATOR, self.private_value, DH_PRIME)
        print(f"[DH] Generated DH values")

    def compute_shared_secret(self, other_public_value):
        """Compute shared secret: (other_public)^private mod p"""
        shared_secret = pow(other_public_value, self.private_value, DH_PRIME)
        return shared_secret

    def derive_session_key(self, shared_secret):
        """Derive AES-128 session key: SHA256(shared_secret)[0:16]"""
        secret_bytes = shared_secret.to_bytes((shared_secret.bit_length() + 7) // 8, 'big')
        session_key = hashlib.sha256(secret_bytes).digest()[:16]
        return session_key
```

Test:

```
============================================================
TESTING PHASE II: DIFFIE-HELLMAN KEY EXCHANGE
============================================================

[TEST] Party A generating DH values...
[DH] Generated private value
[DH] Computed public value: 121114829167954251748037626949869449213837797032980329368911104138593239287216384128700166255110199742854327482225592337425014429679766526755479436371667359211051219552148896934117923353174141594510069425831378834981968920419386756734591319850403616238791655219358384442911988070638657376410184382455791717160695593348390684612671416300975088245509867632164591057865784963034610714810252336723975518938034359122383068941307604059697734714848748999995735622074673

[TEST] Party B generating DH values...
[DH] Generated private value
[DH] Computed public value: 124443164136659138174522735020387059277757047920120200415757369818452999093247366509007186018155065037879539284726018725219068144037520073794915346904744039028358188312540102496644479539463207639598284902665262433793916778754721990150305873949882546892997181077822158457988764199761332028100563044453390584535485971796405997567848537813501044365272198681004137047409037867516149100968502696909826260686395172610241404887667838748834652092932409298683109008214480 1
```

As we show party A and B Generated there private value that is secret and compute public value that is not secret.

```
[TEST] Computing shared secrets...
[DH] Computed shared secret
[DH] Computed shared secret
[TEST] ✓ SUCCESS: Both parties computed same shared secret!
[DH] Derived session key: 2f2920f3d72525ff15bd4a8719b69c70
[DH] Derived session key: 2f2920f3d72525ff15bd4a8719b69c70
[TEST] ✓ SUCCESS: Both parties derived same session key!
```

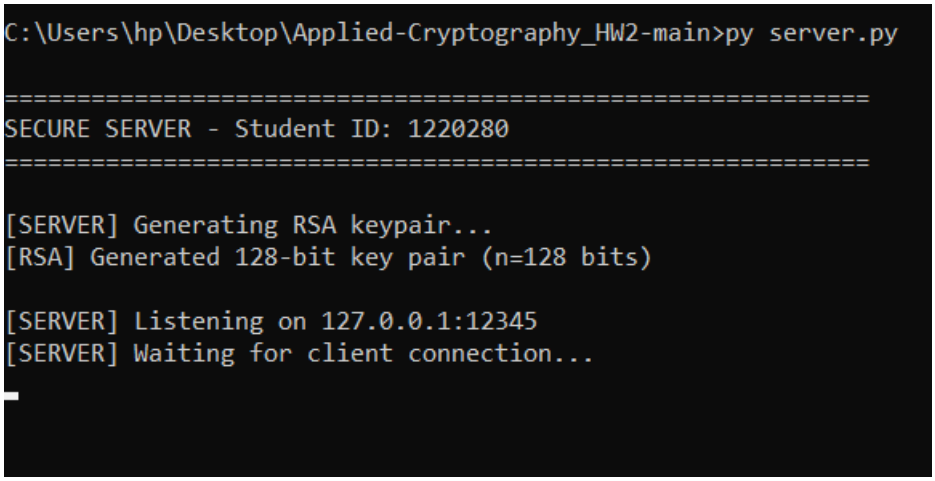
After that A and B share their public value and they computed their shared secret value that as we show is equal.

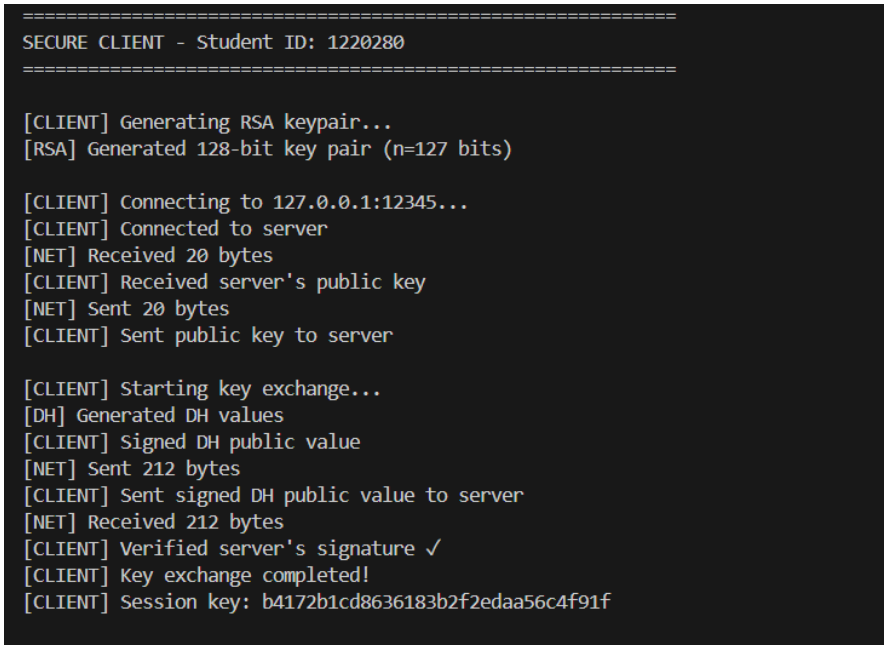
Networking:

Theoretical:

The **server** starts, generates an **RSA key pair**, binds to `127.0.0.1:12345`, and waits for a connection The **client** starts, generates its own **RSA key pair**, and connects to the server. Both sides exchange **RSA public keys** so they can verify each other's signatures. They begin the **Diffie–Hellman key exchange**: each side generates DH values and **signs its DH public value** using its RSA private key. The signed DH public values are exchanged, and each side **verifies the received signature**. After successful verification, both parties compute the same **shared secret** and derive the same **session key**. A **secure channel is established**, and the server is ready to receive **encrypted messages** by AES-CBC with a fresh IV per message. (as we show in the test figures).

Test:

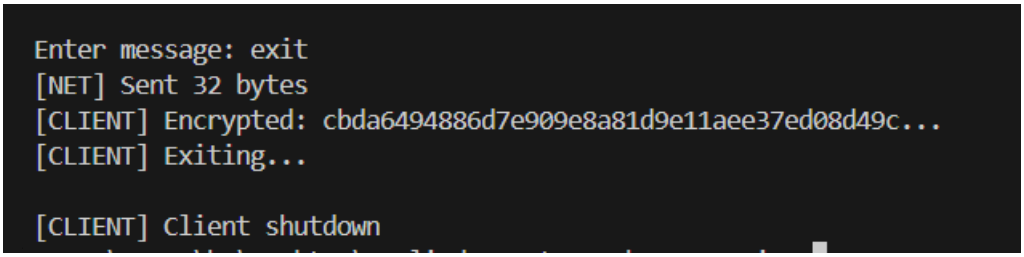


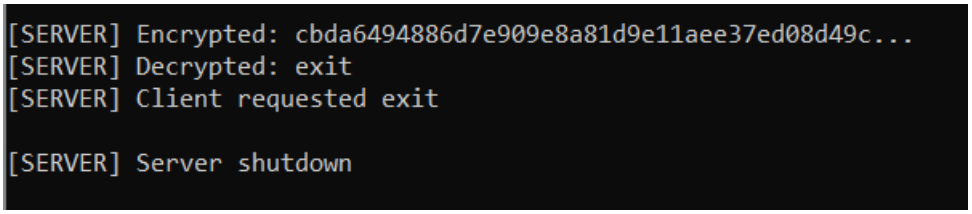


```
[SERVER] Client connected from ('127.0.0.1', 62100)
[NET] Sent 20 bytes
[SERVER] Sent public key to client
[NET] Received 20 bytes
[SERVER] Received client's public key

[SERVER] Starting key exchange...
[DH] Generated DH values
[NET] Received 212 bytes
[SERVER] Verified client's signature ✓
[SERVER] Signed DH public value
[NET] Sent 212 bytes
[SERVER] Sent signed DH public value to client
[SERVER] Key exchange completed!
[SERVER] Session key: b4172b1cd8636183b2f2edaa56c4f91f

============================================================
SECURE CHANNEL ESTABLISHED
============================================================

[SERVER] Ready to receive encrypted messages...
```

```
============================================================
SECURE CHANNEL ESTABLISHED
============================================================

[CLIENT] You can now send encrypted messages.
[CLIENT] Type 'exit' to quit.

Enter message: Applied Cryptography
[NET] Sent 48 bytes
[CLIENT] Encrypted: 79260a72154189cab6ff3820f66ddf5be58b7033...
[NET] Received 64 bytes
[CLIENT] Response: Server received: 'Applied Cryptography'

Enter message:
```

```
============================================================
SECURE CHANNEL ESTABLISHED
============================================================

[SERVER] Ready to receive encrypted messages...
[NET] Received 48 bytes

[SERVER] Encrypted: 79260a72154189cab6ff3820f66ddf5be58b7033...
[SERVER] Decrypted: Applied Cryptography
[NET] Sent 64 bytes
```


Enter message: exit
[NET] Sent 32 bytes
[CLIENT] Encrypted: cbda6494886d7e909e8a81d9e11aee37ed08d49c...
[CLIENT] Exiting...
[CLIENT] Client shutdown


[SERVER] Encrypted: cbda6494886d7e909e8a81d9e11aee37ed08d49c...
[SERVER] Decrypted: exit
[SERVER] Client requested exit
[SERVER] Server shutdown

Conclusion:

In conclusion, we successfully built a secure client–server communication channel by combining **Diffie–Hellman** to generate a shared session secret without transmitting it, **RSA signatures** to authenticate exchanged values and verify identities, and **AES-128-CBC** (with PKCS#7 padding and a fresh random 16-byte IV per message) to encrypt all data after the handshake. This design achieves **confidentiality**, since anyone monitoring network traffic can only observe ciphertext; **integrity**, because tampering with messages causes decryption/validation to fail; **authenticity**, because signature verification confirms which party sent each message; and **forward secrecy**, because each session derives a new ephemeral key, keeping past sessions secure even if long-term keys are compromised later.