Course Title: Cryptography

Course Code: CCCY312

Due Date: 8 November 2025

Name: Hattan Akbar Alsayed

ID: 2340098

Section: (04) (16826)

Lab 2

Overview for AES & Block Cipher Modes

1.1.1 Advanced Encryption Standard (AES):

is a symmetric key (shared secret key) algorithm that encrypts data in blocks of 128 bits (16 bytes). The standard key sizes are 128,192, or 256 bits. The core function of AES is to take 16-byte plaintext block and transform it into a 16-byte ciphertext block through a series of complex mathematical rounds. These rounds provide Byte Substitution Layer (Confusion), ShiftRows & MixColumn Layers (Diffusion), and Key addition Layer (Whitening), which are essential properties for a secure cipher.

1.1.2 The Role of Block Cipher Modes:

Because AES operates on blocks, a mode of operation is required to securely encrypt messages. When message does not perfectly match the size of the block, padding is added to the last to ensure it meets the 16-byte requirements.

A critical component for secure modes is the initialization Vector (IV) which is a 16-byte unpredictable value used to introduce randomness. Its primary role is to ensure that identical plaintexts don't produce identical ciphertexts. Also, Reusing an IV with the same key is critical security failure and must be avoided

- **Electronic Codebook (ECB):** is the Simplest mode, which encrypts each 16-byte block independently. Its main weakness is that it's deterministic (identical plaintexts produce identical ciphertexts), this structure and patterns of data is making it insecure for most applications.

- **Output Feedback (OFB):** it transforms block cipher into stream cipher, generates a keystream by encrypting the IV then encrypting the output of the previous operation then this keystream is XORed with the plaintext to create the ciphertext. It is non-deterministic and effectively hides patterns mitigating the main weakness of ECB.

1.1.3 Security Goals:

- Confidentiality: is the primary goal of encryption. Which ensures that only the authorized parties can access the information.
- Integrity & Authenticity: This goal ensures data is not altered or tampered and that it originates from a trusted source.

The modes of ECB and OFB only provide Confidentiality which have no protection against an attacker modifying the ciphertext. But the Authenticated Encryption (AE) modes like Galois/Counter mode (GCM/CTR) provide both confidentiality and integrity in a single, secure package.

Why ECB Mode is insecure

1.2.1 ECB mechanism:

The ECB encrypts each block independently using the same secret key. If a 16-byte block of plaintext appears 100 times in your original message its corresponding 16-byte ciphertext block will also appear 100 times which leads to pattern leakage. Basically, identical plaintexts result in identical ciphertexts which is called deterministic.

This vulnerability is dangerous with structed or repetitive data because:
ECB encryption preserves the repetitions in the ciphertext, allowing an attacker to analyze frequencies and infer the underlying structure without the key.

1.2.2 In Known-plaintext Attack:

an attacker has access to a piece of the plaintext and its corresponding ciphertext.

Chosen-plaintext Attack: is even more powerful against ECB. Because the attacker can trick the system into encryption arbitrary plaintext blocks of their choice

How OFB Mode Mitigates ECB Weaknesses

OFB mode Mitigate ECB pattern leakage by transforming the AES block cipher into a stream cipher. It achieves this by generating a random keystream that is independent of plaintext using IV first then encrypts the previous keystream block, creating chain where keystream block depends on the one before it. The resulting keystream then combined with the plaintext using XOR. This ensures they result in completely different ciphertext blocks masking any patterns or repetitions in the original data

Conclusion

The Primary risk of ECB mode is that it is deterministic, meaning identical blocks of plaintext produce identical blocks of ciphertext. This leads to a vulnerability called pattern leakage, allowing the attacker to analyze repetitions in the ciphertext. This weakness makes ECB vulnerable to known-plaintext and chosen-plaintext attacks.

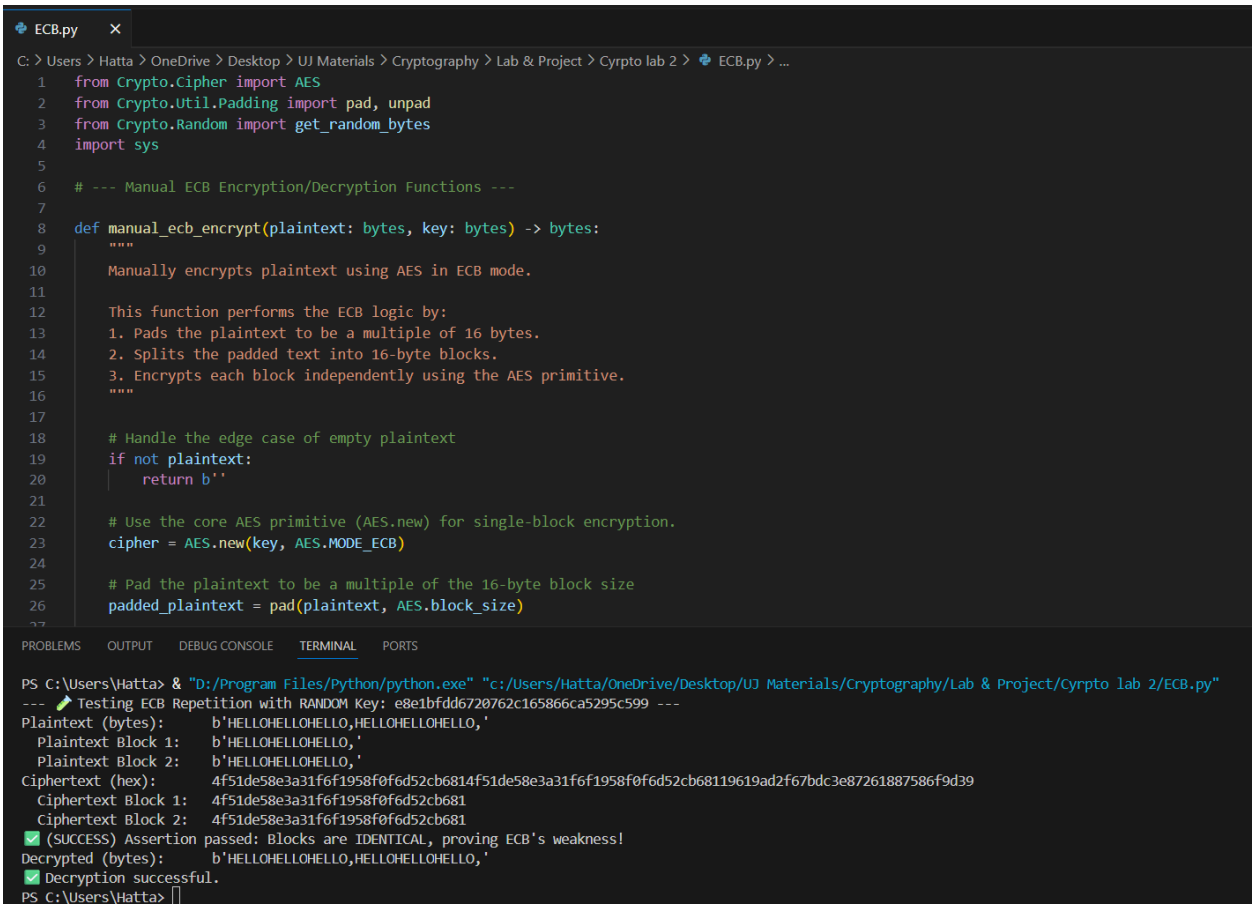
And for the Benefits of OFB and GCM The OFB mitigates ECB's risk by transforming the block cipher into a stream cipher. It generates a random, non-deterministic keystream that is independent of plaintext. While OFB improves ECB's confidentiality, neither mode provides integrity to protect against data tampering. A major benefit of a secure mode like GCM is that it is an Authenticated Encryption (AE) mode that provides both confidentiality and integrity in a single, secure package.

2. Code Implementation

This section details the manual implementation of AES-ECB and AES-OFB. The complete Python scripts (ECB.py, OFB.py) are available on BlackBoard. Per lab constraints, only the low-level AES primitive from pycryptodome was used; all mode logic was implemented manually.

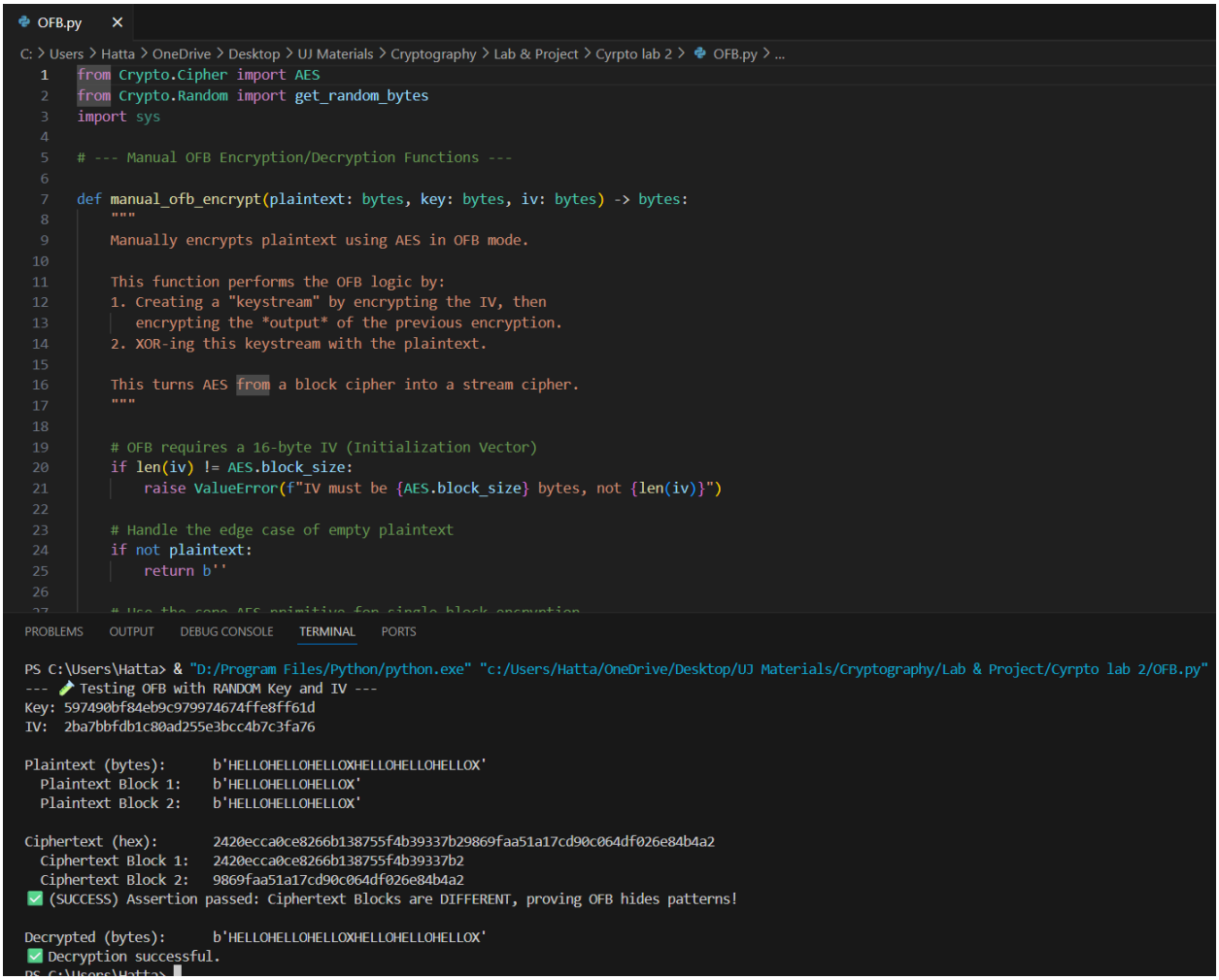
2.1 Implementation of ECB:

AES-ECB: The manual-ecb_encrypt function first pads the plaintext to a 16-byte boundary. It then iterates through the padded data,splitting it into 16-byte blocks and encrypting each block independently using the cipher.encrypt() primitive. The decrypt function reverses this.



2.2 Implementation of OFB

AES-OFB: The implementation functions as a stream cipher per the rubric's definition. A keystream is generated by first encrypting the 16-byte IV, then successively encrypting the previous keystream block. This keystream is XORed with the plaintext byte-by-byte, which inherently handles variable-length data without padding.



2.3 Code Design and Compliance

Functions are parameterized for key and iv. Safety is maintained by avoiding library mode helpers (using AES.new(key, AES.MODE_ECB) only for the primitive), validating the 16-byte IV size, and handling inputs. Test scripts use get_random_bytes for secure, non-hardcoded IV generation.

3. Cryptanalysis on AES-ECB

To fulfill the cryptanalysis requirement in the rubric, the find_ecb_repetitions function was implemented within ECB.py. This section details the method and results.

3.1 Detection Method

The find_ecb_repetition function analyzes raw ciphertext to find duplicate blocks. It operates by:

1. Iterating through the ciphertext and splitting it into 16-byte blocks.
2. Using a dictionary to map each unique block (as a hex string) to a list of its 0-based indices.
3. Filtering this dictionary to return only the entries that appeared more than once.

3.2 Analysis and Results

1. Test Design: A known-repeating 32-byte plaintext is encrypted

2. Analysis: The find_ecb_repetitions function is called on the resulting ciphertext.

3.Evidence: The tool successfully identifies the duplicated block and prints a report.

4. Comparison of AES-ECB and AES-OFB

4.1 AES-ECB Results:

The output from a sample run clearly demonstrates the mode's weakness:

Plaintext Block 1: HELLOHELLOHELLOX

Plaintext Block 2: HELLOHELLOHELLOX

Ciphertext Block 1: a74a51a8bd120085fca87d936b4cf521

Ciphertext Block 2: a74a51a8bd120085fca87d936b4cf521

The Identical plaintext blocks resulted in identical ciphertext blocks. This proves ECB's deterministic nature, where encrypting each block independently leads to significant pattern leakage.

This confirms that ECB fails to provide confidentiality for data structure.

4.2 AES-OFB Results:

The same repeating plaintext was used to test the OFB implementation with a randomly generated key and IV:

Plaintext Block 1: HELLOHELLOHELLOX

Plaintext Block 2: HELLOHELLOHELLOX

Ciphertext Block 1: eb3fb139c26b92d031e49a1d607e0488

Ciphertext Block 2: 787eb09c376dd59c0a2ea8107c637216

In this case, the resulting ciphertext blocks are completely different. OFB achieves this by XORing the plaintext with a non-repeating, keystream, which successfully randomizes the output and obscures the underlying pattern. This confirms that OFB successfully provides confidentiality for data structure.

However, OFB's security is still limited. It only provides confidentiality and offers no integrity or authenticity. This makes it vulnerable to malleability (bit-flipping attacks), where an attacker could modify the ciphertext in transit. Furthermore, OFB's security is critically dependent on the use of a unique, unpredictable IV for every encryption session.

5. Improvement of OFB

5.1 Weakness Identification:

1- Lack of integrity (Mellability): OFB is vulnerable to bit-flipping attacks. An attacker can modify the ciphertext in transit. Causing a predictable, malicious change in the decrypted plaintext that goes completely undetected.

2- IV Reuse Catastrophe: if an IV is ever reused with the same key, an attacker can XOR two ciphertexts to reveal the XOR of their plaintexts, which is a catastrophic loss of confidentiality.

5.2 Proposed Improvement:

1- CTR.GCM

2- OFB with nonce-derivation (KDF for IV)

3- Synthetic IV (SIV)-like construction

4- Encrypt-then-MAC

I Chose Encrypt-then-MAC:

This suggested improvement is to wrap the existing OFB implementation in an Encrypt-then-MAC scheme using HMAC-SHA256. This adds a layer of authenticity and integrity.

The scheme requires two independent keys: an encryption key (key_enc) and an authentication key (key_mac) for HMAC

*Encryption Process:

1- The plaintext is encrypted using the function we created with key_enc and a random IV, producing the ciphertext.

2- An authentication mac is generated by computing an HMAC over the IV and ciphertext together (HMAC(key_mac, ivi + ciphertext)).

3- The final transmitted package is (IV, ciphertext, mac).

*Decryption process:

1- The recipient first recalculated the expected MAC using their key_mac on the received IV and ciphertext.

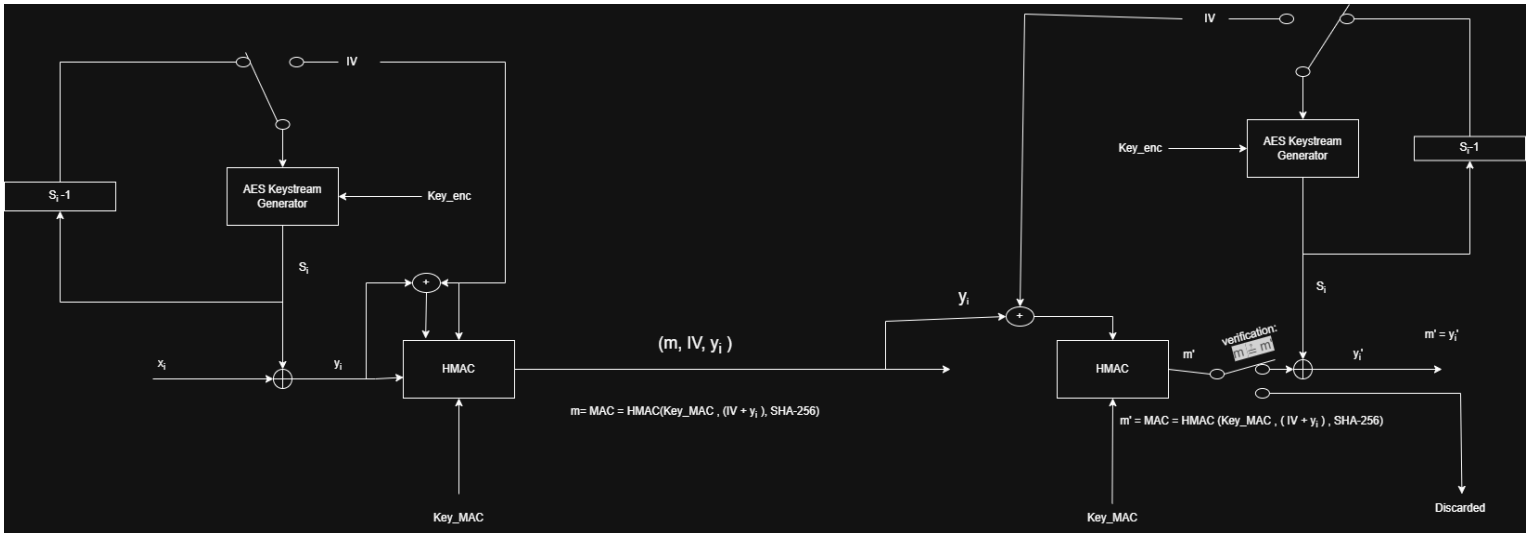
2- This expected MAC is securely compared to the received mac

3- If they do not match, the message is rejected as tampered, and decryption stops

4- If they do match, the message is authentic and is safely decrypted.

Why this Overcomes the weakness? The Encrypt-then-Mac design directly mitigates the malleability attack. Any modification to the ciphertext by an attacker will cause the HMAC verification to fail, as the tampered data will not match the original MAC. The Invalid message is then discarded, ensuring integrity.

Block diagram:



The Picture of the Block diagram will also be attached on BlackBoard.

5.3 Implementation of Encrypt-then-MAC:

This method mitigates the bit-flipping attack. Any modification to the ciphertext will cause the HMAC verification to fail and discard the message. Also, the Code is provided on BlackBoard with full Documentation. Which tested with the same pattern of the previous codes and the same plaintext.

improved_OFB.py

C: > Users > Hatta > OneDrive > Desktop > UJ Materials > Cryptography > Lab & Project > Cyrpto lab 2 > improved_OFB.py > ...

```python
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES
import hmac
import hashlib
import sys

# This assumes OFB.py is in the same directory
try:
    from OFB import manual_ofb_encrypt, manual_ofb_decrypt
except ImportError:
    print("Error: manual_ofb.py not found. Please ensure it's in the same directory.")
    sys.exit(1)

# ---(Encrypt-then-MAC) ---

def improved_ofb_encrypt(plaintext: bytes, key_enc: bytes, key_mac: bytes) -> tuple:
    """
    Encrypts and authenticates plaintext using OFB with an Encrypt-then-MAC scheme.

    Args:
        plaintext (bytes): The data to encrypt.
        key_enc (bytes): The 16-byte key for AES-OFB encryption.
        key_mac (bytes): The key for HMAC authentication (any size, 16+ is good).

    Returns:
        tuple: (iv, ciphertext, mac)
    """
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
✅ Decryption successful.
Decrypted Plaintext:    b'HELLOHELLOHELLOX'
✅ Test Passed: Plaintext matches.

--- Test 2: Simulating Bit-Flipping Attack ---
Simulating an attacker flipping the first byte of the ciphertext...

  Attacking 'manual_ofb_decrypt' (no integrity):
  Decrypted Tampered:   b'IELLOHELLOHELLOX'
   DANGER: Decryption succeeded on tampered data (produced garbage).
   WEAKNESS CONFIRMED: The old OFB cannot detect tampering.

  Attacking 'improved_ofb_decrypt' (with integrity):
   Authentication failed! Message has been tampered with or is invalid.
   SUCCESS: The improved function detected the tampering and refused to decrypt.
```

5.4 Comparison between improved-OFB and Original-OFB in case of Bit-Flipping attack

Function Tested	Input Data	Result	Security Implication
Manual_ofb_decrypt	**Tampered Ciphertext**	**SUCCESS** (Produces Garbage)	**FAIL**. The function cannot detect tampering, leading to a silent integrity breach
Improved_ofb_decrypt	**Tampered Ciphertext**	**FAILURE** (Raises Value Error)	**PASS**. The function correctly detects the tampering via the HMAC check and refuses to decrypt