AES Algorithm

Applied Cryptography

25 september 2024

Hussaini Mohmmad Qasim, Volonterio Luca

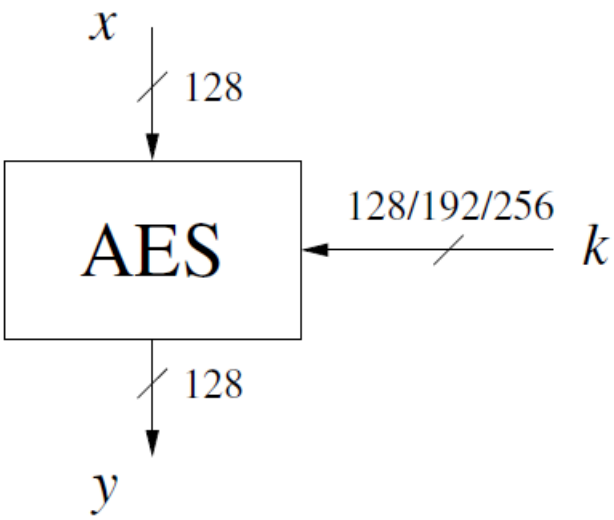
Contents

1 Objective

The goal of this lab is to explore the implementation of the Advanced Encryption Standard (AES) algorithm, which focuses on key generation, encryption, and decryption. This lab report explains how round keys are generated and how AES encryption and decryption processes work. Additionally, this lab introduces us to different AES modes of operation and their effects on encryption, followed by the implementation of file encryption and decryption using AES, ensuring that the original text is successfully recovered after decryption.

2 A short introduction on AES

The AES cipher is almost identical to the block cipher Rijndael. The Rijndael block and key size vary between 128, 192 and 256 bits. However, the AES standard only calls for a block size of 128 bits. Hence, only Rijndael with a block length of 128 bits is known as the AES algorithm.



As mentioned previously, three key lengths must be supported by Rijndael as this was a NIST design requirement. The number of internal rounds of the cipher depends on the key length.

Key lengths and number of rounds for AES

key lengths	# rounds = n_r
128 bit	10
192 bit	12
256 bit	14

Each AES operation consists of so-called layers. Each layer manipulates all 128 bits of data, which is arranged in a particular way in a 4x4 bytes matrix that is called state of the algorithm. There are only three different types of layers: a linear mixing layer (diffusion), a non-linear layer and a key addition layer. Each round, with the exception of the first, consists of all three layers: the plaintext is denoted as x, the ciphertext as y and the number of rounds as nr. In the key scheduling process, the same primitives are used.

3 AES Round-Keys Generation

A crucial aspect of AES is the generation of round keys from a given master key, also called Key Scheduling. These round keys are used in each round of the encryption and decryption process.

3.1 Key Expansion Algorithm

AES key expansion starts with a master key and generates multiple round keys. The provided code utilizes the following components:

S-box: A substitution box used to introduce non-linearity by replacing each byte with a corresponding value.

RCON: Round constants added to the key during the key expansion process.

text2matrix: A function that converts a 128-bit key into a 4x4 matrix of bytes, where each byte represents part of the key. The function *change_key*(*master_key*) generates the round keys by copying the first four words directly from the master key and then perfmrning the key espantion, decripted below.

3.2 Code Implementation

The function *change_key* uses the following code to generate the round keys from the master key:

```
def change_key(master_key):
    round_keys = text2matrix(master_key)
    for i in range(4, 4 * 11):
        round_keys.append([])
        if i % 4 == 0:
            byte = round_keys[i - 4][0]         \
                    ^ Sbox[round_keys[i - 1][1]]  \
                    ^ Rcon[int(i / 4)]
            round_keys[i].append(byte)
            for j in range(1, 4):
                byte = round_keys[i - 4][j]     \
                     ^ Sbox[round_keys[i - 1][(j + 1) % 4]]
                round_keys[i].append(byte)
        else:
            for j in range(4):
                byte = round_keys[i - 4][j]     \
```

```
                    ^ round_keys[i - 1][j]
                round_keys[i].append(byte)
    return round_keys
```

The *text2matrix* function converts the *128*-bit master key into a *4x4* matrix of bytes. The first four words (16 bytes) are taken directly from the master key. For each subsequent word: If the index is divisible by 4, the word undergoes the RotWord and SubWord operations (byte rotation and substitution via the S-box), followed by XOR with the RCON value and the corresponding word from the previous round. Otherwise, the word is generated by XORing the previous word with the word from the previous round.

3.3 Sample Output

A 128-bit master key

```
    0x2b7e151628aed2a6abf7158809cf4f3c
```

was used for testing. The (decimal) matrix form of this master key is:

```
[[43, 126, 21, 22],
[40, 174, 210, 166],
[171, 247, 21, 136],
[9, 207, 79, 60]]
```

The generated round keys are then printed in groups of four (hexadecimal) words. Each subkey corresponds to one of the 11 keys used in the AES encryption process.

```
Master Key:
0x2b 0x28 0xab 0x9
0x7e 0xae 0xf7 0xcf
0x15 0xd2 0x15 0x4f
0x16 0xa6 0x88 0x3c

Subkey 1:
0xa0 0x88 0x23 0x2a
0xfa 0x54 0xa3 0x6c
0xfe 0x2c 0x39 0x76
0x17 0xb1 0x39 0x5
...
Subkey 10:
0xd0 0xc9 0xe1 0xb6
0x14 0xee 0x3f 0x63
0xf9 0x25 0xc 0xc
0xa8 0x89 0xc8 0xa6
```

The program generates 11 subkeys (the original key plus 10 more for each encryption round), which is compliant with a 128 bit master key according to the AES standard.

4 Completing AES Functions for Encryption

In this part of the lab, we completed three missing functions critical to the AES encryption process: SubBytes, ShiftRows, and MixColumns. The goal was to implement these functions, understand their roles within the AES algorithm, and verify the correctness of the encryption process by comparing the output with known results. We completed the missing functions necessary for AES encryption.

4.1 AES Encryption Process

The encrypt function, which implements the AES encryption process, utilizes the completed transformations along with the *add_round_key*() function:

```
def encrypt(plaintext):
    plain_state = text2matrix(plaintext)
    add_round_key(plain_state, round_keys[:4])
    for i in range(1, 10):
        round_encrypt(plain_state, round_keys[4 * i : 4 * (i + 1)])
    sub_bytes(plain_state)
    shift_rows(plain_state)
    add_round_key(plain_state, round_keys[40:])
    return matrix2text(plain_state)
```

AddRoundKey: The AddRoundKey performs a bitwise XOR between the current state and the round current key.

```
def add_round_key(s, k):
        for i in range(4):
            for j in range(4):
                s[i][j] ^= k[i][j]
```

The function iterates through each byte in the state matrix s and replaces it with the corresponding bitwise XOR value.

SubBytes: The SubBytes function substitutes each byte of the state matrix with its corresponding value from a predefined substitution box (S-box).

```
def sub_bytes(s):
    "Build the SubBytes transformation"
    for i in range(4):
        for j in range(4):
            s[i][j] = Sbox[s[i][j]]
```

The function iterates through each byte in the state matrix s and replaces it with the corresponding byte from the S-box.

ShiftRows: The ShiftRows function performs a cyclic shift of the rows in the state matrix. Each row is shifted left by a number of bytes equal to its row index.

```
def shift_rows(s):
    "Build the ShiftRows transformation"
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]
```

This function shifts the second row left by 1 byte, the third row by 2 bytes, and the fourth row by 3 bytes. The first row remains unchanged.

MixColumns: The MixColumns function mixes the bytes within each column of the state matrix, providing diffusion by combining the values of the columns.

```
xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)
def mix_single_column(a):
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)
def mix_columns(s):
    for i in range(4):
     mix_single_column(s[i])
```

mix_single_column processes a single column of the state matrix by using polynomial arithmetic in GF(2^8) to mix the bytes. The *mix_columns* function iterates through each column of the state matrix, applying *mix_single_column* to each one.

Results: The encryption process was tested with the following plaintext:

```
plaintext = 0x3243f6a8885a308d313198a2e0370734
```

The ciphertext produced was:

```
ciphertext = encrypt(plaintext)
print('\n ciphertext is:', hex(ciphertext))
```

The output was verified against the known ciphertext:

```
ciphertext is: 0x3925841d02dc09fbdc118597196a0b32
```

The verification showed that the ciphering process was correct:

```
ciphering has been done correctly
```

5 AES Decryption Function Implementation

In this part of the lab, we focused on completing the decryption process for the Advanced Encryption Standard (AES) algorithm. we implemented three missing functions: invSubBytes, invShiftRows, and invMixColumns.

5.1 AES Decryption Process

The overall decryption process in the decrypt function utilizes the completed transformations along with the *add_round_key* function:

```
def decrypt(ciphertext):
    cipher_state = text2matrix(ciphertext)
    add_round_key(cipher_state, round_keys[40:])
    inv_shift_rows(cipher_state)
    inv_sub_bytes(cipher_state)
    for i in range(9, 0, -1):
        round_decrypt(cipher_state, round_keys[4 * i : 4 * (i + 1)])
    add_round_key(cipher_state, round_keys[:4])
    return matrix2text(cipher_state)
```

Inverse SubBytes: The *inv_sub_bytes* function substitutes each byte of the state matrix with its corresponding value from a predefined inverse substitution box (InvSbox).

```
def inv_sub_bytes(s):
    "Build the InvSubBytes transformation"
    for i in range(4):
        for j in range(4):
            s[i][j] = InvSbox[s[i][j]]
```

This function iterates through each byte in the state matrix s and replaces it with the corresponding byte from the InvSbox.

Inverse ShiftRows: The *inv_shift_rows* function performs a cyclic right shift of the rows in the state matrix, effectively reversing the shift operation applied during encryption.

```
def inv_shift_rows(s):
    "Build the InvShiftRows transformation"
    s[1][1], s[2][1], s[3][1], s[0][1] = s[0][1], s[1][1], s[2][1], s[3][1]
    s[2][2], s[3][2], s[0][2], s[1][2] = s[0][2], s[1][2], s[2][2], s[3][2]
    s[3][3], s[0][3], s[1][3], s[2][3] = s[0][3], s[1][3], s[2][3], s[3][3]
```

This function shifts the second row to the right by 1 byte, the third row by 2 bytes, and the fourth row by 3 bytes. The first row remains unchanged, reversing the effect of the encryption shift operation.

Inverse MixColumns: The *inv_mix_columns* function processes each column of the state matrix to reverse the mixing operation applied during encryption.

```
def inv_mix_columns(s):
    # Refer to Sec 4.1.3 in The Design of Rijndael
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
```

```
        s[i][2] ^= u
        s[i][3] ^= v
    mix_columns(s)
```

This function applies polynomial arithmetic in GF(2^8) to reverse the mixing of the columns. Each column is transformed to recover the original byte values before mixing occurred during encryption.

Results: The decryption process was tested with the following ciphertext:

```
ciphertext = 0x3925841d02dc09fbdc118597196a0b32
```

The corresponding plaintext was:

```
plaintext = 0x3243f6a8885a308d313198a2e0370734
```

The decryption was performed as follows:

```
decrypted = decrypt(ciphertext)
print('plaintext was:', hex(plaintext))
print('decryption returns:', hex(decrypted))
print('plaintext and ciphertext are the same:', hex(plaintext) == hex(decrypted))
```

The output confirmed:

```
plaintext was: 0x3243f6a8885a308d313198a2e0370734
decryption returns: 0x3243f6a8885a308d313198a2e0370734
plaintext and ciphertext are the same: True
```

6 AES Algorithm Using PyCryptodome Library

The PyCryptodome library is a Python package providing cryptographic services, including the Advanced Encryption Standard (AES). This part of the lab focuses on performing AES encryption and decryption using PyCryptodome, comparing the results with a custom AES implementation, and exploring different modes of operation such as ECB and CTR.

The goal here is to utilize the PyCryptodome library for AES encryption and decryption and explore the different AES modes of operation such as ECB and CTR to understand their differences. Also, we compare the reuslts of PyCryptodome AES encryption with our just showcased custom AES implementation.

To begin, we installed PyCryptodome using this command:

```
!pip install pycryptodome
```

The AES encryption process using PyCryptodome started by generating random 16-byte plaintext and a 16-byte AES master key. The plaintext was then encrypted using AES in ECB mode.

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

# Generate random 16-byte plaintext and key
```

```
plaintext_bytes = get_random_bytes(16)
plaintext = int.from_bytes(plaintext_bytes, byteorder='big')

master_key_bytes = get_random_bytes(16)
master_key = int.from_bytes(master_key_bytes, byteorder='big')

# Encrypt using AES in ECB mode
cipher = AES.new(master_key_bytes, AES.MODE_ECB)
ciphertext_bytes = cipher.encrypt(plaintext_bytes)
ciphertext = int.from_bytes(ciphertext_bytes, byteorder='big')
```

The state of the plaintext, master key, and ciphertext was printed for each byte block. This allowed us to see how the encryption transforms the plaintext into ciphertext using the provided key.

```
# Display plaintext, key, and ciphertext
print('Plaintext:', hex(plaintext))
print('Master key:', hex(master_key))
print('Ciphertext:', hex(ciphertext))

Plaintext (in hexadecimal): 0x5f0d6be0140ecf3453688606cf26e972
Master Key (in hexadecimal): 0x7da20ab142bc372c4de4110eac084330
Ciphertext (in hexadecimal): 0xc7dc88544beb483fbadb863b17d939b8
```

We compared the output of the PyCryptodome AES implementation with our previously developed implementation. We used the same plaintext and key for both algorithms, and we verified that the ciphertexts matched.

```
# Compare custom AES implementation with PyCryptodome results
plaintext_aes = 0x5f0d6be0140ecf3453688606cf26e972
masterkey_aes = 0x55fc4c6461aad1c2c0bc139eb4db8d87
ciphertext_aes = encrypt(plaintext_aes, masterkey_aes)
print('Both algorithms provide the same ciphertext:', ciphertext_aes == ciphertext)
```

Both the PyCryptodome AES and custom AES implementation produced the same ciphertexts:

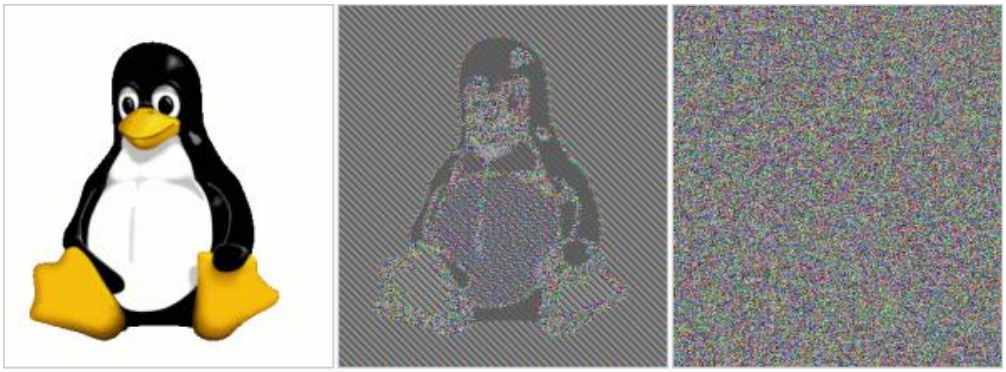
```
0xc7dc88544beb483fbadb863b17d939b8
```

After the decryption, we restored the original plaintext to ensure the correctness of both encryption and decryption processes. Please note that the original prototype of the *encrypt*() and *decrypt*() functions has been modified to accept the key as a parameter, because the original one was receiveing it as a global variable.

6.1 AES Various Modes of Operation

ECB Mode The default mode used in the previous steps was **ECB (Electronic Codebook)** mode. While this mode works efficiently by encrypting each block independently, it has security drawbacks due to the potential for repeating patterns in the ciphertext when the plaintext has repetitive blocks. This modality

requires the message to be divided into blocks, and each block is encrypted separately. The lack of diffusion between blocks fails to hide data patterns, this is why this is mostly never used in cryptographic protocols.



Original image | Encrypted using ECB mode | Modes other than ECB result in pseudo-randomness

CTR Mode Then, we explored the CTR (Counter) mode, which behaves like a stream cipher by generating a key stream using an incremental encrypted counter that is XORed with the plaintext. This mode provides improved security by using a unique nonce (number only used once) as an initial value for the counter, to make sure that even if the same plaintext is encrypted multiple times, the resulting ciphertext is different.

```
from base64 import b64encode

data = b"secret"
key = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_CTR)
ct_bytes = cipher.encrypt(data)
nonce = b64encode(cipher.nonce).decode('utf-8')
ct = b64encode(ct_bytes).decode('utf-8')

print(f'Nonce: {nonce}, Ciphertext: {ct}')
```

The output included the nonce and the corresponding ciphertext:

```
Nonce: "D+dlQ6o8HGo="
Ciphertext: "qswggsxu"
```

CTR mode confirmed that the same plaintext block would not produce the same ciphertext across multiple encryptions which enhance security. Here is the result we got running the encryption of the same plaintext with the same key 5 times:

```
{"nonce": "5TINh9AT58I=", "ciphertext": "eV2m9xlQ"}
{"nonce": "4NqPODzK7H0=", "ciphertext": "BWktSI4o"}
{"nonce": "UEiGtliLWD0=", "ciphertext": "b0m9v6lf"}
{"nonce": "sKJoFXsSlGQ=", "ciphertext": "i4tUEUiL"}
{"nonce": "aLIATEEOfpM=", "ciphertext": "93sFkRTy"}
```

7 File Encryption and Decryption

In this part of the lab, we will show how to encrypt and decrypt a text file using AES with ECB. ECB mode is a straightforward block cipher mode, where each block of plaintext is encrypted independently. We generated a random AES key, encrypt a text file, and ensure that decryption accurately restores the original content. After that, we read and encrypted text from a file using AES in ECB mode. Then, we decrypted the encrypted text and compared it with the original content to verify correctness. We read the content of a text file *(Test.txt)* into a string for encryption. The file contained a standard *"Lorem Ipsum"* placeholder text, which was used as the plaintext input for AES encryption.

7.1 AES Encryption and Decryption in ECB Mode

We implemented the AES encryption using PyCryptodome's AES module in ECB mode. The text was converted to bytes using UTF-8 encoding. We did padding to ensure the data length is a multiple of 16 bytes (the AES block size).

```
def encryptMessage(message, key):
  message = message.encode('utf-8')
  ciphertext = []
  for chunk in chunkstring(message, 16):
    plaintext = chunk.hex()
    plaintext = int(plaintext,16)
    ciphertext.append(encrypt(plaintext, key))
  return ciphertext
```

We decrypted the encrypted data using the same AES key to ensure that the ciphertext could be correctly restored to the original plaintext. We did unpadding the decrypted data to remove any padding bytes added during encryption. Finally, we converted the decrypted bytes back into a string.

```
def decryptMessage(enc_message, key):
  print("Alice receives encrypted message: ", enc_message)
  decrypted_aes_string = ""
  for block in enc_message:
    decrypted_aes = decrypt(block, key)
    decrypted_aes_string += bytes.fromhex(format(decrypted_aes,'x')).decode('utf-8')
  return decrypted_aes_string
```

File Output and Verification: To test the encryption and decryption process, the results were printed to the console and optionally saved to output files. This allowed us to confirm that the encryption and decryption steps functioned as expected. We then compared decrypted text to the original plaintext, and they matched exactly, confirming that the encryption and decryption processes were implemented correctly.

```
if original_text == decrypted_text:
    print("Success! The decrypted text matches the original text.")
```

```
else:
    print("Error! The decrypted text does not match the original text.")
```

The output confirmed:

```
Success! The decrypted text matches the original text.
```

8 Conclusions

In this lab, we implemented the AES encryption algorithm with a focus on round-key generation, encryption, and decryption. We generated 11 subkeys from a 128-bit master key and used them for both encryption and decryption processes. The core AES functions (SubBytes, ShiftRows, MixColumns) and their inverses were successfully implemented and verified through correct encryption and decryption of data. Additionally, we used the PyCryptodome library to compare our custom AES implementation, ensuring the accuracy of both methods. The implementation of file encryption and decryption in ECB mode further confirmed the versatility of the produced code.