Christopher K. Leung (2965-7518-69)
CSCI-531 Applied Cryptography

Implementation of

Advanced Encryption Standard (AES)

128-bit Electronic Codebook Mode

Introduction

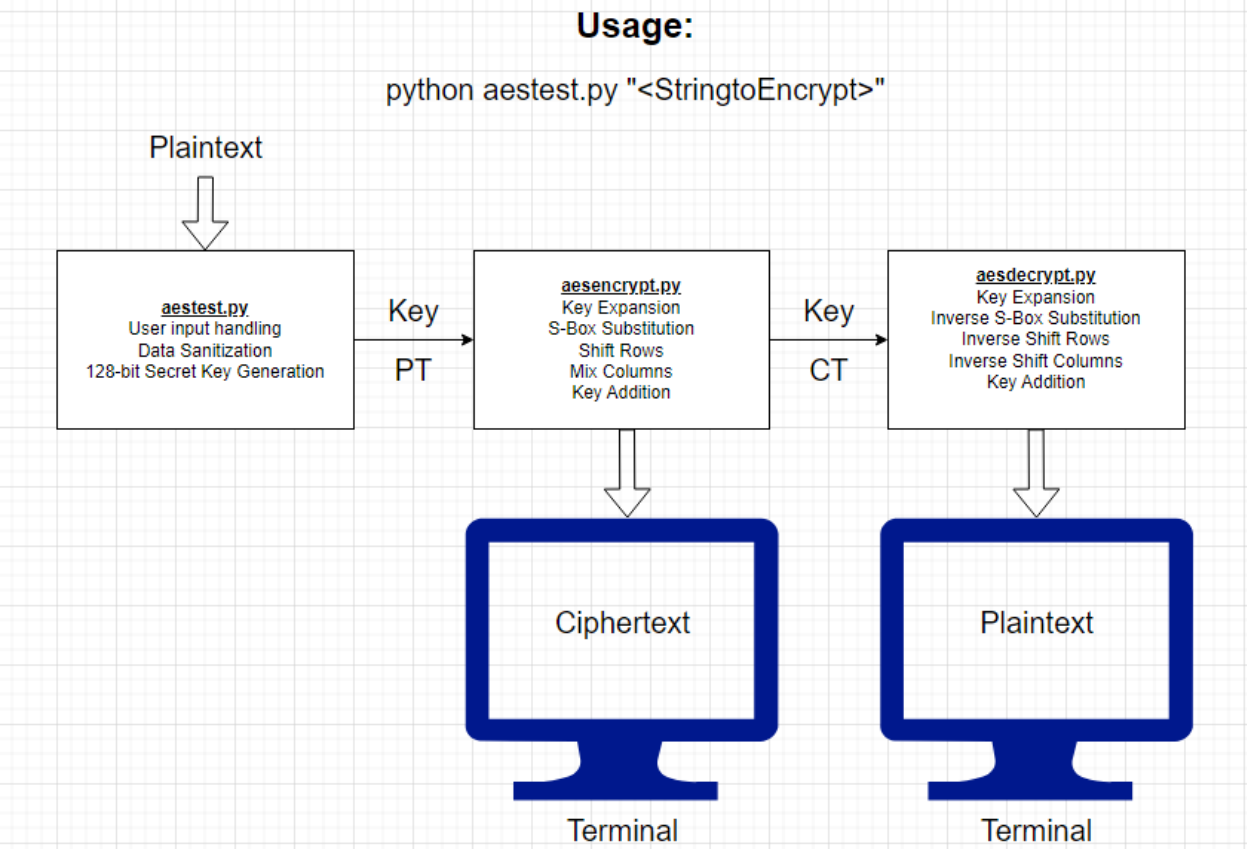
The Advanced Encryption Standard (AES) has been set forth and have been approved by the Federal Information Processing Standards (FIPS) for data encryption. Although the AES algorithm allows for key sizes up to 256 bits, our specification was to implement AES-128 using electronic codebook (ECB) as its only mode of operation.

Environment

For my environment, I am running PyCharm 2022.3.2 Professional Edition on Windows 11 Home version 22H2

Block Diagram

The block diagram below illustrates the interaction and the passage of data between the python scripts



Usage

I have generated 3 files for the grader:

1. aestest.py
2. aesencrypt.py
3. aesdecrypt.py

aestest.py Functional Description

Parameters:

python aestest.py "<StringtoEncrypt>"

Description:

This python script is the **only** entry point for encryption and decryption. This script will handle user input, ISO/IEC 7816-4 padding and generation of the primary 16-byte secret key used throughout the algorithm.

I have used Python's os.urandom() library to generate a 16-byte secret key every time this script is run. The user **shall not** provide a secret key as a command line argument.

Verification of ISO/IEC 7816-4 padding scheme has been outlined by our CSCI-531 grader:

https://piazza.com/class/lcgjdtutgvz7o9/post/77

Verification to use other libraries to generate the random cipher key:

https://piazza.com/class/lcgjdtutgvz7o9/post/94

Edge cases:

Some edge cases were covered by utilizing Python's sys.argv to ensure proper formatting is achieved and correct block sizes are passed into aesencrypt.py

1. aestest.py checks if length of args < 2. If so, script will notify user the proper formatting and quit

```
C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>python aestest.py
---------------------------------
CSCI-531 AES-128 ECB Implementation
---------------------------------
[ERROR]: Not enough command line arguments
[Usage]: python3 aestest.py "<StringtoEncrypt>"
Exiting Now..
```

2. aestest.py checks if length of args > 2. If so, script will notify user that too many arguments have been passed in.

```
C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>python aestest.py "ENCRYPTME" shouldnotbehere
---------------------------------
CSCI-531 AES-128 ECB Implementation
---------------------------------
[ERROR]: Too many command line arguments
[Usage]: python3 aestest.py "<StringtoEncrypt>"
[Tip 1]: Ensure plaintext is encased with double quotations "<StringtoEncrypt>"
[Tip 2]: If plaintext contains quotations ("") use escape characters (\) to include
[Example]: python3 aestest.py "\"<StringtoEncrypt>\""
Exiting Now..

C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>
```

3. aestest.py checks to ensure that there are no all-whitespace strings (strings with only spaces or ASCII byte code of 0x20) are passed in. Plaintext should always be passed in.

```
C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>python aestest.py "                    "
---------------------------------
CSCI-531 AES-128 ECB Implementation
---------------------------------
[ERROR]: Plaintext only contains spaces - No data to encrypt
[Usage]: python3 aestest.py "<StringtoEncrypt>"
Exiting Now..

C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>
```

4. aestest.py checks to ensure that no empty strings are passed in (strings with length of 0)

```
C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>python aestest.py ""
---------------------------------
CSCI-531 AES-128 ECB Implementation
---------------------------------
[ERROR]: Plaintext is a NULL string - 0 bytes captured from CLI
[Usage]: python3 aestest.py "<StringtoEncrypt>"
Exiting Now..
```

5. aestest.py supports inter-quotes encryption. Quotation marks can be encrypted if **escape characters** are used. In the screen capture below, "\"Hello, World!\"" becomes "Hello, World!" once decrypted.

```
C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>python aestest.py "\"Hello, World!\""
---------------------------------
CSCI-531 AES-128 ECB Implementation
---------------------------------
[aestest.py] AES-128 Random Key (HEX):
0x4f 0x03 0x4f 0x2d 0xde 0xd8 0x9a 0x6c 0xcd 0xbc 0xa6 0xaa 0x2e 0xd1 0x70 0xad

[aesencrypt.py] Ciphertext:
0x5e 0xe1 0x49 0x1c 0x45 0x41 0xd2 0x0c 0xcd 0xc7 0xc5 0x00 0x56 0x21 0x26 0xac

[aesdecrypt.py] Plaintext (ASCII):
"Hello, World!"
```

Regular operation

Since aestest.py is the only entry point, users should utilize the parameter section to become familiar with the script.

The encoding scheme chosen for plaintext parsing and data sanitization is UTF-8

The encoding scheme chosen for the secret key generated by os.urandom() is in hexadecimal format

Once all conditions have been met (i.e padding, plaintext length, etc.) aestest.py will pass (x2) byte arrays (plaintext and key) to aesencrypt.aes_enc_main().

```
C:\Users\Christopher Leung\Desktop\CSCI531\AES\CSCI531_AES\src>python aestest.py "Lorem ipsum dolor sit amet. Ea voluptatibus
 consequatur aut recusandae dolore est molestiae minima ut architecto consequatur. Vel asperiores dignissimos sed error enim
id consequatur similique ut facilis architecto."
-----------------------------------
CSCI-531 AES-128 ECB Implementation
-----------------------------------
[aestest.py] AES-128 Random Key (HEX):
0xca 0xc3 0xaf 0xd2 0x9b 0x99 0xdd 0xd7 0xcb 0xd6 0x73 0xf5 0xf5 0xdd 0x35 0x17

[aesencrypt.py] Ciphertext:
0xaa 0x34 0x7f 0x32 0x70 0x12 0x2b 0x22 0x73 0x8c 0x0f 0x89 0x15 0xee 0x31 0x24 0x51 0xa7 0xbc 0xf3 0x19 0xa7 0x9d 0xac 0xa1
0xc4 0xad 0x5b 0xe0 0xfb 0x36 0xea 0xac 0x68 0x29 0xa5 0xf6 0xea 0x07 0x37 0x69 0xca 0xce 0xec 0xf4 0x02 0x06 0x77 0xb2 0xe8
0x9b 0xeb 0x28 0x0c 0xb9 0xbb 0xa2 0x4f 0x9b 0x38 0x94 0xf4 0xe7 0xbf 0xcd 0x57 0xb3 0x40 0x64 0x1a 0x91 0x4b 0x9b 0x94 0x21
0x7f 0x77 0xdb 0x08 0xe1 0xbc 0x7a 0x8b 0xa1 0x1a 0xca 0xf5 0x10 0x91 0xa5 0x49 0xa4 0x79 0xea 0x9a 0xb8 0x6f 0xa2 0x52 0x36
0xb4 0x03 0xd4 0x19 0x26 0xdc 0xfb 0xd4 0x0d 0x4b 0xef 0xfe 0xfa 0x10 0xaf 0xa0 0x6f 0x0b 0xad 0x13 0xbd 0x2c 0x09 0x61 0x68
0xca 0x60 0x98 0xbf 0x69 0x1f 0x3b 0xb0 0x77 0x0a 0xf2 0x76 0x78 0xf2 0x1f 0x2b 0x4e 0xe6 0x46 0x8f 0x12 0xa9 0x55 0xe6 0xbf
0xeb 0x3d 0xc0 0xa9 0xaf 0xe8 0xd6 0xe0 0xb4 0xb7 0x19 0x3c 0x40 0x86 0x24 0x2f 0x66 0x7b 0xa4 0x84 0x6c 0x70 0xf3 0x17 0x69
0xd6 0x68 0xbc 0xc0 0xf6 0x44 0xbc 0xb0 0xf7 0x45 0x99 0x1d 0x99 0x7a 0xa3 0xc0 0xfd 0x00 0xd8 0x25 0xf9 0xaf 0x78 0x37 0x4c
0xc5 0xaf 0xcf 0xd8 0x73 0xf9 0xcd 0x46 0xa3 0x2e 0x5e 0x0d 0x6c 0x5c 0x48 0x0d 0x49 0xcb 0x7c 0xbb 0x03 0x81 0x93 0x1f

[aesdecrypt.py] Plaintext (ASCII):
Lorem ipsum dolor sit amet. Ea voluptatibus consequatur aut recusandae dolore est molestiae minima ut architecto consequatur.
 Vel asperiores dignissimos sed error enim id consequatur similique ut facilis architecto.
```

aesencrypt.py Functional Description

Input:

Plaintext and Cipher Key from aestest.py

Output:

Ciphertext and Cipher Key to aesdecrypt.py

Description:

aesencrypt.py is designed to accept only input from aestest.py. This is to ensure that the user interacts with aesencrypt.py properly and input has been sanitized through the middleware. Per FIPS 197, I have chosen to maintain the basic unit for processing (byte), however, on occasion will you see in the source code that I have chosen to convert 4 bytes into a 32-bit word to assist with the logic.

aesencrypt.py follows the encryption process by utilizing sub-routines to assist with certain tasks. I have broken these up into the following sub-tasks that do not create a collision with other functions such that my methods would be able to be dropped “as-is” into another python script:

1. key_expansion()
2. s_box_sub()
3. shift_rows()
4. mix_cols()
5. xor_2d()

key_expansion()

Take 128-bit key and turn it into (x10) additional 128-bit keys. These keys will be utilize across all rounds of aesencrypt.py during XOR’ing the round key with the current state matrix. The return from this function creates a 11 X 4 matrix. Each row stores (x4) 32-bit words used for key addition.

s_box_sub()

Takes in a state matrix, and captures every most significant nibble (MS_nibble) and least significant nibble (LS_Nibble) and utilizes the Rijndael S-Box constant matrix to perform a substitution. Each MS_nibble and LS_Nibble represent row and column respectively to perform a Rijndael S-Box lookup.

shift_rows()

This function creates a 32-bit word from each row in our state array shifts the value by an amount dependent on the row, then replaces each byte in the state matrix

mix_cols()

Perform a 1X4 * 4X1 matrix multiplication to obtain new values for the state array. In conjunction with the constant mix columns matrix, there are only 3 values of interest during this multiplication.

1. 0x01 – value does not change
2. 0x02 – value needs to be shifted to the left by 1. If the MS bit is set before the shift, XOR the shifted value by the Galois Field 2^8 irreducible polynomial 0x1B
3. 0x03 – similar to 0x02 ^ 0x01

xor_2d()

Take in (x2) 2D arrays and perform XOR on all bytes of the arrays. Store back into the first array passed in.

Christopher K. Leung (2965-7518-69)
CSCI-531 Applied Cryptography

aesdncrypt.py Functional Description

Input:

Ciphertext and Cipher Key from aesencrypt.py

Output:

None

Description:

aesdncrypt.py is designed to accept only input from aesencrypt.py. This is to ensure that the user has interacted with the command chain of aestest.py -> aesencrypt.py -> aesdecrypt.py to obtain the decryption of a given ciphertext.

aesdncrypt.py follows the decryption process by utilizing sub-routines to assist with certain tasks. I have broken these up into the following sub-tasks that do not create a collision with other functions such that my methods would be able to be dropped “as-is” into another python script:

1. key_expansion()
2. s_box_inv_sub()
3. shift_rows_inv()
4. inv_mix_cols()
5. xor_2d()

key_expansion()

Take 128-bit key and turn it into (x10) additional 128-bit keys. These keys will be utilize across all rounds of aesdncrypt.py during XOR’ing the round key with the current state matrix. The return from this function creates a 11 X 4 matrix. Each row stores (x4) 32-bit words used for key addition. For decryption, aesdecrypt.py iterates from key 10 to key 0.

s_box_inv_sub()

Takes in a state matrix and captures every most significant nibble (MS_nibble) and least significant nibble (LS_Nibble) and utilizes the Rijndael Inverse S-Box constant matrix to perform a substitution. Each MS_nibble and LS_Nibble represent row and column respectively to perform a Rijndael Inverse S-Box lookup.

shift_rows_inv()

This function creates a 32-bit word from each row in our state array and shifts the value right by an amount dependent on the row, then replaces each byte in the state matrix

Inv_mix_cols()

Perform a 1X4 * 4X1 matrix multiplication to obtain new values for the state array. In conjunction with the constant mix columns matrix, there are only 3 values of interest during this multiplication. I did not use a lookup table. I have utilized a stack overflow resource to properly implement this without the need for a pre-defined table

Inverse Mix Columns Resource : https://crypto.stackexchange.com/questions/2569/how-does-one-implement-the-inverse-of-aes-mixcolumns

1. 0x09 - ((((x * 2) * 2) * 2) + x)
2. 0x0B - (((((x * 2) * 2) + x) * 2) + x)
3. 0x0D - (((((x × 2) + x) × 2) × 2) + x)
4. 0x0E - (((((x × 2) + x) × 2) + x) * 2)

The idea behind this is to break up the values (0x09, 0x0B, 0x0D, 0x0E) into their respective decimal values (9, 11, 13, 14) and utilize a combination of bit shifting to multiply by two, XOR for addition, and the Galois Field 2^8 irreducible polynomial (0x1B) to maintain our significant 8 bits of data.

xor_2d()

Take in (x2) 2D arrays and perform XOR on all bytes of the arrays. Store back into the first array passed in.

After the sub-routines perform the decryption process, the output is a 1D array of N bytes. From this, a method iso_iec_7816_4_unpad() searches for a special byte value (0x80) and returns a subset of this 1D array without the padding. This completes the decryption process

Christopher K. Leung (2965-7518-69)
CSCI-531 Applied Cryptography

Resources

AES FIPS Publication 197:

https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

Project Requirements:
https://piazza.com/class_profile/get_resource/lcgjdtutgvz7o9/ldukaqkhpl85xx

Rijindael Mix Columns:

https://en.wikipedia.org/wiki/Rijndael_MixColumns

Solving Inverse Mix Columns:

https://crypto.stackexchange.com/questions/2569/how-does-one-implement-the-inverse-of-aes-mixcolumns

Solving Mix Columns:

https://crypto.stackexchange.com/questions/2402/how-to-solve-mixcolumns