Encryption and Decryption with AES and RSA

By: Ethan Vito

General way to create a cipher

- Each cipher in my program is named in the same way. You encrypt with the encrypt_<mode> and pass the plaintext, key, iv, and whatever else is necessary for that given mode.
- For example encrypt with ofb mode is simply encrypt_ofb
- **ENCRYPTION**
    - **Plaintext:** string (can be a hex value or ASCII printable)
    - **Key:** The key is passed as a string. For the purposes of this program, all keys are 128 bits long.
    - **IV:** The iv is passed as a string. IVs are also 128 bits long.
    - **Return value (for all encrypt functions):** All encrypt functions will return a bytes object containing the ciphertext
- There is a corresponding decrypt_<mode> function for each encrypt_<mode> function. It gets passed the ciphertext (which is a bytes object), key, iv, and whatever else is necessary for the given mode.
- **DECRYPTION**
    - **Ciphertext**: Ciphertext is passed as a bytes object, this is the only difference.
    - **Key, IV** are the same as encryption.
    - **Return value (for all decrypt functions):** All decrypt functions will return the original plaintext in bytes. Needs to be converted from bytes to string to get original plaintext.

Example of encrypting & decrypting with CFB mode:

```
key = '00000000000000000000000000000000'
iv = '00000000000000000000000000000000'
msg = 'Normal text'
segment_size = 64
ct = encrypt_cfb(msg, key, iv, segment_size)
print(ct.hex()) # print the cipher text

pt = decrypt_cfb(ct, key, iv, segment_size)
print(decode_byte_object(pt)) # print plaintext
```

This code here shows how to encrypt using cfb as an example. The function decode_byte_object is a helper function I wrote. Since the decrypt functions for each mode return a bytes object, we need to convert this back into a string so we can print it. decode_byte_object does this for us.

How I got my test data:

- In each of my tests, the key, message, iv, and all other values come from the AES standard under the section Appendix F: Example Vectors. This way I know if my program is producing the correct output
    - https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

Testing each mode

I've written a test_<mode> function for each mode that tests encrypting and decrypting for the given mode and prints additional information. If you run main.py it will run this test function for each mode. The examples through this document are the same as those found when running main.py.

Preparing a message

Preparing a message is easy. Simply create a string of the plaintext that you want to encrypt. The plaintext message can either be ASCII printable or a hex value. Hex values must be at least one byte. For example, the hex value **f** would be interpreted as a normal character and not a hex value. The hex value **ff** would be treated as hex, however. You would then pass this message to the encrypt function. Two examples shown below. The encrypt function handles the type conversion of the plaintext.

```
msg = 'The quick brown fox jumps over the lazy dog'
msg2 = '6a84867cd77e12ad07ea1be895c53fa3'
encrypt_cbc(msg, key, iv)
encrypt_cbc(msg2, key, iv)
```

Generating a key

Although I did not generate random keys for these tests (because I used the example vector tests from the AES standard) a user could generate a random key if they wanted to. If a user wants to generate a key, they could call the gen_key() function that I have written inside util.py. It will generate a random 128 bit key using python's secrets module and return it as a string.

Example key generation:

```
key = gen_key()
print(key)
Program output:
1bb865cff9f804a215724be943182b72
```

Encryption and Decryption

For each mode, I am encrypting the same 512 bit (64 byte) message, using the same key, and IV where applicable.

Concept and Purpose of Each Mode:

ECB

- ECB mode is the simplest of all the AES modes. It takes the plaintext block and runs it through AES encryption with a key. It does not chain or combine any outputs together.
- ECB is typically used when you want to encrypt single values like a session key. It is good for parallelism but is too simple to provide strong encryption for more than one block.
- ECB is not useful for generating different ciphertext given the same input (there is no randomness).

CBC

- CBC mode is the start of our chaining modes. It requires an IV that is XORed with the plaintext and chains the output of one block to the next block during encryption.
- It can be used in parallel but only during decryption.
- CBC solves the problem of duplicate data by utilizing an IV and feedback which generates different ciphertext for the same plaintext.
- CBC can be used for authentication and general block-oriented transmission.

CFB

- CFB mode works similarly to CBC mode. The difference lies in the segment size and bitwise shifting. The output of our ciphertext depends on the chosen segment size. Ciphertext is generated using s bits of input and previous ciphertext is used to generate the new input for subsequent blocks.
- CFB can be used in parallel but only during decryption.
- CFB can be used for authentication and general strean-oriented transmission.
- This mode also uses AES Encrypt function for both encryption and decryption which means you don’t need to have code space for AES Decrypt.

OFB

- OFB Mode is similar to CFB. Instead of using the ciphertext as part of the new input for the next block it uses the output of the encryption block.
- OFB can’t be used in parallel at all
- OFB mode also uses AES Encrypt function for both encryption and decryption.
- OFB mode is useful for stream oriented channels like satellite communication due to it being tolerant to data loss where an error in one block does not propagate.

CTR

- CTR mode uses a counter that starts at an initial value and increments by one for each block that is encrypted. The counter is used as an input for each encryption block.
- CTR mode is good for parallelization. It is very useful for high-speed requirements because of this. CTR mode uses Encrypt function for both encryption and decryption.
- CTR can be used as a stream cipher.

ECB:

If you were to encrypt and decrypt with ECB mode you would do the following:
The encrypt function takes the message, and key, while the decrypt function takes the ciphertext, and key.

```
key = '2b7e151628aed2a6abf7158809cf4f3c'
msg = '6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710'
ct = encrypt_ecb(msg, key) #encrypt msg
pt = decrypt_ecb(ct, key) #decrypt ct to get pt
```

```
ECB Multi-block message test from AES Standard.
Input to Program:
KEY = 2b7e151628aed2a6abf7158809cf4f3c | IV = None for ECB
Plaintext =       6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  3ad77bb40d7a3660a89ecaf32466ef97 f5d3d58503b9699de785895a96fdbaaf 43b1cd7f598ece23881b00e3ed030688 7b0c785e27e8ad3f8223207104725dd4
Cipher:           3ad77bb40d7a3660a89ecaf32466ef97 f5d3d58503b9699de785895a96fdbaaf 43b1cd7f598ece23881b00e3ed030688 7b0c785e27e8ad3f8223207104725dd4
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710

ECB test with ASCII printable characters (not hex)
Input to Program:
Key = 00000000000000000000000000000000 | IV = None for ECB
Plaintext = "Normal text"
Cipher: c6bb1fbca843ff5c8920e3f4309b384e
Decrypted Cipher: Normal text
```

Cipher output here as well if it's hard to read:

```
3ad77bb40d7a3660a89ecaf32466ef97
f5d3d58503b9699de785895a96fdbaaf
43b1cd7f598ece23881b00e3ed030688
7b0c785e27e8ad3f8223207104725dd4
```

CBC:

If you were to encrypt and decrypt with CBC mode you would do the following. The encrypt function takes the message, key, and iv, while the decrypt function takes the ciphertext, key, and iv.

```
key = '2b7e151628aed2a6abf7158809cf4f3c'
iv = '000102030405060708090a0b0c0d0e0f'
msg = '6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710'
ct = encrypt_cbc(msg, key, iv) #encrypt msg
pt = decrypt_cbc(ct, key, iv) #decrypt ct to get pt
```

The image below shows the output of the program using this input as well as an example with some different input

```
CBC Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f
Plaintext =      6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher: 7649abac8119b246cee98e9b12e9197d 5086cb9b507219ee95db113a917678b2 73bed6b8e3c1743b7116e69e22229516 3ff1caa1681fac09120eca307586e1a7
Cipher:          7649abac8119b246cee98e9b12e9197d 5086cb9b507219ee95db113a917678b2 73bed6b8e3c1743b7116e69e22229516 3ff1caa1681fac09120eca307586e1a7
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710

CBC test with ASCII printable characters (not hex)
Input to Program:
Key = 00000000000000000000000000000000 | IV = 00000000000000000000000000000000 | Plaintext = "Normal text"
Cipher: c6bb1fbca843ff5c8920e3f4309b384e
Decrypted Cipher: Normal text
```

Cipher output here as well if it’s hard to read:

```
7649abac8119b246cee98e9b12e9197d
5086cb9b507219ee95db113a917678b2
73bed6b8e3c1743b7116e69e22229516
3ff1caa1681fac09120eca307586e1a7
```

CFB:

If you were to encrypt and decrypt with CBC mode you would do the following. The encrypt function takes the message, key, iv, and segment_size while the decrypt function takes the ciphertext, key, iv, and segment_size.

- **CFB Mode requires a segment size (s bits). The size needs to be a multiple of 8 bits (1 byte). An error will be thrown otherwise**

```
key = '2b7e151628aed2a6abf7158809cf4f3c'
iv = '000102030405060708090a0b0c0d0e0f'
segment_size = 128
msg = '6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710'
ct = encrypt_cfb(msg, key, iv, segment_size) #encrypt msg
pt = decrypt_cfb(ct, key, iv,segment_size) #decrypt ct to
get pt
```

The image below shows the output of the program using this input as well as an example with some different input

```
Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f
Plaintext =       6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  3b3fd92eb72dad20333449f8e83cfb4a c8a64537a0b3a93fcde3cdad9f1ce58b 26751f67a3cbb140b1808cf187a4f4df c04b05357c5d1c0eeac4c66f9ff7f2e6
Cipher:           3b3fd92eb72dad20333449f8e83cfb4a c8a64537a0b3a93fcde3cdad9f1ce58b 26751f67a3cbb140b1808cf187a4f4df c04b05357c5d1c0eeac4c66f9ff7f2e6
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710

CFB test with ASCII printable characters (not hex)
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f | Plaintext = "Normal text"
Cipher: 288639b98ee60c4f14a3b05a5b5a852e
Decrypted Cipher: Normal text
```

Cipher output here as well if it’s hard to read:

```
3b3fd92eb72dad20333449f8e83cfb4a
c8a64537a0b3a93fcde3cdad9f1ce58b
26751f67a3cbb140b1808cf187a4f4df
c04b05357c5d1c0eeac4c66f9ff7f2e6
```

OFB

If you were to encrypt and decrypt with OFB mode you would do the following. The encrypt function takes the message, key, and iv, while the decrypt function takes the ciphertext, key, and iv.

```
key = '2b7e151628aed2a6abf7158809cf4f3c'
iv = '000102030405060708090a0b0c0d0e0f'
msg = '6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710'
ct = encrypt_ofb(msg, key, iv) #encrypt msg
pt = decrypt_ofb(ct, key, iv) #decrypt ct to get pt
```

The image below shows the output of the program using this input as well as an example with some different input

```
Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f
Plaintext =       6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  3b3fd92eb72dad20333449f8e83cfb4a 7789508d16918f03f53c52dac54ed825 9740051e9c5fecf64344f7a82260edcc 304c6528f659c77866a510d9c1d6ae5e
Cipher:           3b3fd92eb72dad20333449f8e83cfb4a 7789508d16918f03f53c52dac54ed825 9740051e9c5fecf64344f7a82260edcc 304c6528f659c77866a510d9c1d6ae5e
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710

OFB test with ASCII printable characters (not hex)
Input to Program:
Key = 00000000000000000000000000000000 | IV = 00000000000000000000000000000000 | Plaintext = "Normal text"
Cipher: 288639b98ee60c4fed348e5ccf312e2b
Decrypted Cipher: Normal text
```

Cipher output here as well if it’s hard to read:

```
3b3fd92eb72dad20333449f8e83cfb4a
7789508d16918f03f53c52dac54ed825
9740051e9c5fecf64344f7a82260edcc
304c6528f659c77866a510d9c1d6ae5e
```

CTR:

If you were to encrypt and decrypt with OFB mode you would do the following. The encrypt function takes the message, key, nonce, and counter while the decrypt function takes the ciphertext, key, nonce, and counter. The **counter value** you provide is the value the counter will start incrementing from.The **nonce** is a fixed 64 bit value that is put together with the counter to make the entire 128 bit counter block. I implemented the counter block the same way as pycryptodome. I have pasted their documentation below for clarification.

```
key = '2b7e151628aed2a6abf7158809cf4f3c'
nonce = 'f78a0908c9083734'
counter = '0000000000000000'
msg = '6bc1bee22e409f96e93d7e117393172a
ae2d8a571e03ac9c9eb76fac45af8e51
30c81c46a35ce411e5fbc1191a0a52ef
f69f2445df4f9b17ad2b417be66c3710'
ct = encrypt_ctr(msg, key, nonce, counter) #encrypt msg
pt = decrypt_ctr(ct, key, nonce, counter) #decrypt ct to
get pt
```

The image below shows the output of the program using this input as well as an example with some different input.

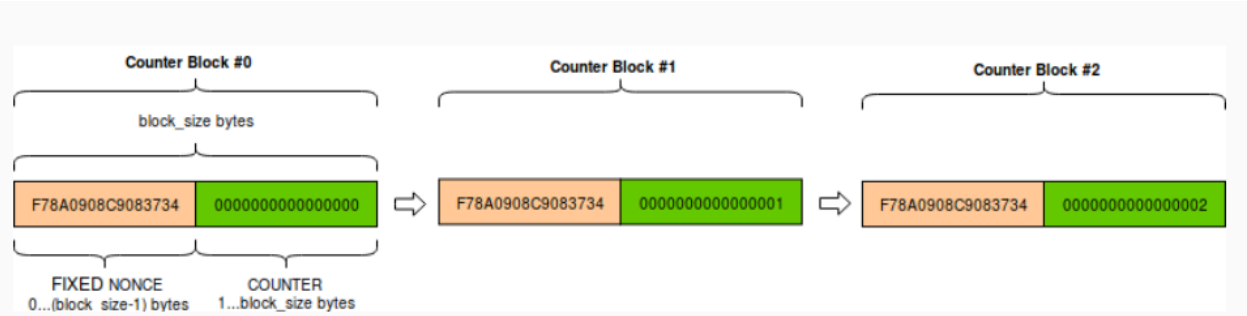
Note: the AES standard example for CTR mode does not use a nonce to create their counter block; their initial counter value is set to **f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff**. I used the same key and plaintext as the standard like I have in all the other examples though. I just didn't use the same initial counter value.

```
Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | Nonce = f78a0908c9083734 | Counter = 0000000000000000
Plaintext =       6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  4a22cfef8843bb96e1ddfe8b1960472a 7930b7193726abf52480fdb058653367 d2ff5e93611574ac6997de2c8f5e6d1d 4b251767ac762e72a8552da6068c5894
Cipher:           4a22cfef8843bb96e1ddfe8b1960472a 7930b7193726abf52480fdb058653367 d2ff5e93611574ac6997de2c8f5e6d1d 4b251767ac762e72a8552da6068c5894
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710

CTR test with ASCII printable characters (not hex)
Input to Program:
Key = 00000000000000000000000000000000 | Nonce = f78a0908c9083734 | Counter = 0000000000000000 | Plaintext = "Normal text"
Cipher: 19b99e6e1f765caff05e7b5089382241
Decrypted Cipher: Normal text
```

Cipher output here as well if it's hard to read:

```
4a22cfef8843bb96e1ddfe8b1960472a
7930b7193726abf52480fdb058653367
d2ff5e93611574ac6997de2c8f5e6d1d
4b251767ac762e72a8552da6068c5894
```



A *counter block* is exactly as long as the cipher block size (e.g. 16 bytes for AES). It consists of the concatenation of two pieces:

1. a fixed **nonce**, set at initialization.
2. a variable **counter**, which gets increased by 1 for any subsequent counter block. The counter is big endian encoded.

Here is how the nonce and counter work together.

Introducing Errors

In order to introduce errors I've written a function called **flip_bit** inside util.py. It will take a ciphertext block and flip the least significant bit (LSB).

Example:

```
flipped_ciphertext = flip_bit(ciphertext)
```

```
Original key value in binary
 0b10101101111110000101010001011000101000101011101101001010100110101010111111011100010101100010000000100111001111010011110011110
Key value with the LSB flipped
 0b10101101111110000101010001011000101000101011101101001010100110101010111111011100010101100010000000100111001111010011110011111
Original Key 2b7e151628aed2a6abf7158809cf4f3c
Bit flip key 2b7e151628aed2a6abf7158809cf4f3d
```

- If you look at the binary representation of the key here, the only difference is that at the LSB position, the value has become a **1** instead of a **0**.
- I've also printed the hex representation of these keys to further illustrate.
- Taking a look at the last 4 bytes of the key in hex.
    - Original key last 4 bytes: **4f3c**
    - Bit Flipped key last 4 bytes: **4f3d**
- Notice the only difference is that we have a **d** now instead of a **c** in the last hex value

How I will implement this function with the ciphertext

For each of the modes, I implemented error propagation by causing a bit flip in the second ciphertext block. I took the resulting ciphertext after encryption and caused a bit flip in the second ciphertext block.

Code Example of how to introduce a bit flip

```
key = '2b7e151628aed2a6abf7158809cf4f3c'
iv = '000102030405060708090a0b0c0d0e0f'
msg = ('6bc1bee22e409f96e93d7e117393172a'
       'ae2d8a571e03ac9c9eb76fac45af8e51'
       '30c81c46a35ce411e5fbc1191a0a52ef'
       'f69f2445df4f9b17ad2b417be66c3710')
ct = (encrypt_cbc(msg, key, iv))

# split ciphertext into an array of bytes
split = split_into_chunks(ct)
# flip a bit in the second ciphertext block
split[1] = flip_bit((split[1]))
# take the array of bytes and turn it back into a single byte object
ct = concatenate_byte_objects(split)

plaintext = decrypt_cbc(ct, key, iv)

print('Cipher:          ', group_string(ct.hex()))
print('Decrypted Cipher:', group_string(decode_byte_object(plaintext)))
```

NOTE on code examples for encrypting and decrypting with errors:

The methods for encrypting and decrypting are exactly the same as before; they are not used any differently so I do not provide code samples here. I caused the bit flip by manually updating the ciphertext outside of the encrypt and decrypt functions. Due to this the code example is exactly the same for each mode.

ECB Error Analysis:

```
ECB Multi-block message test from AES Standard.
Input to Program:
KEY = 2b7e151628aed2a6abf7158809cf4f3c | IV = None for ECB
Plaintext =      6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  3ad77bb40d7a3660a89ecaf32466ef97 f5d3d58503b9699de785895a96fdbaaf 43b1cd7f598ece23881b00e3ed030688 7b0c785e27e8ad3f8223207104725dd4
Cipher:          3ad77bb40d7a3660a89ecaf32466ef97 f5d3d58503b9699de785895a96fdbaae 43b1cd7f598ece23881b00e3ed030688 7b0c785e27e8ad3f8223207104725dd4
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a 6106f8bf4703f6fcab18f369abc2498c 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
```

- In this example, I introduced an error in the second cipher text block. If you take a look at the expected cipher and the actual cipher in the second block, you'll notice the last hex value is an **e** instead of an **f.**
- This error did not propagate anywhere.
    - The error only affected the second block of plaintext.
    - The second block in the original plaintext and the second block in the decrypted cipher are completely different.
    - This makes sense since ECB does not tie any output to other input blocks. The ciphertext is fed back into AES to get back to the plaintext (which is why our decrypted cipher in block 2 is different)

Code example (Can be found in ecb.py function test_ecb())

```
ct = (encrypt_ecb(msg, key))
split = split_into_chunks(ct)#split the ciphertext
into
                            #an array of 16 byte
objects
split[1] = flip_bit((split[1]))
ct = concatenate_byte_objects(split)
```

The code here takes the ciphertext, splits it apart into an array (with each index containing a 128 bit block). I then index the second block and update it by calling flip_bit on it. I then take the array and turn it into a single byte object with concatenate_byte_bjects (a function I wrote in util.py). This lets me pass the updated ciphertext into the decrypt function to see how the error propagates.

CBC Error Analysis:

```
CBC Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f
Plaintext =      6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher: 7649abac8119b246cee98e9b12e9197d 5086cb9b507219ee95db113a917678b2 73bed6b8e3c1743b7116e69e22229516 3ff1caa1681fac09120eca307586e1a7
Cipher:          7649abac8119b246cee98e9b12e9197d 5086cb9b507219ee95db113a917678b3 73bed6b8e3c1743b7116e69e22229516 3ff1caa1681fac09120eca307586e1a7
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ac1bf824b43340c7fb5c9b68119bb006 30c81c46a35ce411e5fbc1191a0a52ee f69f2445df4f9b17ad2b417be66c3710
```

- In this example, I introduced an error in the second cipher text block. If you take a look at the expected cipher and the actual cipher in the second block, you'll notice the last hex value is a **3** instead of a **2.**
- This error then propagated from block two into block three of the plaintext.
    - Comparing the original plaintext to the decrypted cipher, blocks two and three are different.
    - Block two is completely different while block three just contains an error in the last bit.
        - The last hex value in the third block is an **e** instead of an **f**
    - This matches my expectations since CBC mode during decryption should only affect the current block and the one next to it. The current block will be completely different since the ciphertext is fed as input into AES, however, the next block should only contain errors in positions where the ciphertext has had a bit flipped. This is because the feedback is XORed with the output of the next block to get the plaintext rather than fed as input into AES.

Code example (Can be found in cbc.py function test_cbc())

```
ct = (encrypt_cbc(msg, key, iv))
split = split_into_chunks(ct)
split[1] = flip_bit((split[1]))
ct = concatenate_byte_objects(split)
```

The code here takes the ciphertext, splits it apart into an array (with each index containing a 128 bit block). I then index the second block and update it by calling flip_bit on it

CFB Error Analysis:

```
Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f
Plaintext =      6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  3b3fd92eb72dad20333449f8e83cfb4a c8a64537a0b3a93fcde3cdad9f1ce58b 26751f67a3cbb140b1808cf187a4f4df c04b05357c5d1c0eeac4c66f9ff7f2e6
Cipher:          3b3fd92eb72dad20333449f8e83cfb4a c8a64537a0b3a93fcde3cdad9f1ce58a 26751f67a3cbb140b1808cf187a4f4df c04b05357c5d1c0eeac4c66f9ff7f2e6
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e50 51e91061ce3a9043911119f073cadac4 f69f2445df4f9b17ad2b417be66c3710
```

- In this example, I introduced an error in the second cipher text block. If you take a look at the expected cipher and the actual cipher in the second block, you'll notice the last hex value is an **a** instead of a **b.**
- This error propagated from block two into block three of the plaintext.
    - Comparing the original plaintext to the decrypted cipher, block 3 is completely different.
    - However, you'll notice that block two only has the single bit flip error rather than the whole block being affected by the ciphertext error.
        - The last hex value in the original plaintext is a **1** while the decrypted value is a **0**
    - This matches my expectations since in CFB, the ciphertext isn't fed into the same block that it was generated from, it is only XORed. The subsequent block does use the previous ciphertext as input, however. This is why we see the 3rd block of plaintext being completely altered in the example above but the second block of plaintext just contains a single bit flip.
    - This is the opposite of what we see in CBC

.Code example (Can be found in cfb.py function test_cfb())

```
ct = (encrypt_cfb(msg, key, iv, 128))
split = split_into_chunks(ct)
split[1] = flip_bit((split[1]))
ct = concatenate_byte_objects(split)
```

The code here takes the ciphertext, splits it apart into an array (with each index containing a 128 bit block). I then index the second block and update it by calling flip_bit on it

OFB Error Analysis:

```
Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | IV = 000102030405060708090a0b0c0d0e0f
Plaintext =      6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  3b3fd92eb72dad20333449f8e83cfb4a 7789508d16918f03f53c52dac54ed825 9740051e9c5fecf64344f7a82260edcc 304c6528f659c77866a510d9c1d6ae5e
Cipher:          3b3fd92eb72dad20333449f8e83cfb4a 7789508d16918f03f53c52dac54ed824 9740051e9c5fecf64344f7a82260edcc 304c6528f659c77866a510d9c1d6ae5e
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e50 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
```

- In this example, I introduced an error in the second cipher text block. If you take a look at the expected cipher and the actual cipher in the second block, you'll notice the last hex value is a **4** instead of a **5.**
- This error did not propagate anywhere.
    - Comparing the original plaintext to the decrypted cipher in the second block, we see that there is also only a 1 bit difference.
    - The difference can be seen in the last hex value which is a **0** instead of a **1**
    - This matches my expectations because the ciphertext for one block is not used to decrypt blocks that come after it. The ciphertext for a block is simply XORed with the output of the current block to get the plaintext. This is why we see the decrypted cipher only contain the single bit flip "error" rather than the whole block being affected.

Code example (Can be found in ofb.py function test_ofb())

```
ct = (encrypt_ofb(msg, key, iv))
split = split_into_chunks(ct)
split[1] = flip_bit((split[1]))
ct = concatenate_byte_objects(split)
```

The code here takes the ciphertext, splits it apart into an array (with each index containing a 128 bit block). I then index the second block and update it by calling flip_bit on it

CTR Error Analysis:

```
Multi-block message test from AES Standard.
Input to Program:
Key = 2b7e151628aed2a6abf7158809cf4f3c | Nonce = f78a0908c9083734 | Counter = 0000000000000000
Plaintext =      6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
Expected Cipher:  4a22cfef8843bb96e1ddfe8b1960472a 7930b7193726abf52480fdb058653367 d2ff5e93611574ac6997de2c8f5e6d1d 4b251767ac762e72a8552da6068c5894
Cipher:          4a22cfef8843bb96e1ddfe8b1960472a 7930b7193726abf52480fdb058653366 d2ff5e93611574ac6997de2c8f5e6d1d 4b251767ac762e72a8552da6068c5894
Decrypted Cipher: 6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e50 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c3710
```

- In this example, I introduced an error in the second cipher text block. If you take a look at the expected cipher and the actual cipher in the second block, you'll notice the last hex value is a **6** instead of a **7.**
- This error did not propagate anywhere.
    - Comparing the original plaintext to the decrypted cipher in the second block, we can see that there is only a 1 bit difference.
    - The difference can be seen in the last hex value which is a **0** instead of a **1**
    - This matches my expectations because CTR does not use the ciphertext as feedback for subsequent blocks. It also does not use the ciphertext as input back into AES, it XORs the ciphertext with the output block to get the plaintext. This is why the only affected part of our decrypted cipher is the last bit not the whole second block.

Code example (Can be found in crt.py function test_ctr())

```
ct = (encrypt_ctr(msg, key, nonce, counter))
split = split_into_chunks(ct)
split[1] = flip_bit((split[1]))
ct = concatenate_byte_objects(split)
```

The code here takes the ciphertext, splits it apart into an array (with each index containing a 128 bit block). I then index the second block and update it by calling flip_bit on it

RSA Encryption/Decryption

The first step to using RSA encryption with pycryptodome is to generate an RSA key pair and store these values in files. The following code snippet generates an RSA key pair and stores the private and public key in their own respective files.

```
key = RSA.generate(2048)
private_key = key.export_key()
file_out = open("private.pem", "wb")
file_out.write(private_key)
file_out.close()

public_key = key.publickey().export_key()
file_out = open("public.pem", "wb")
file_out.write(public_key)
file_out.close()
```

In order to encrypt an arbitrary amount of data we will use the hybrid encryption scheme as described by pycryptodome documentation under the **Encrypt data with RSA** example. It involves using RSA for asymmetric encryption of an AES session key. A full example of this code can be found later in the Time Consumption Comparison section.

Step 1:
Import the “recipients” public key and generate a 16 byte session key.

```
data = b'Test data to encrypt'
recipient_publickey = RSA.importKey(open('public.pem').read())
session_key = get_random_bytes(16)
```

Step 2:
We then need to generate an rsa cipher with our public key with PKCS1 OAEP so we can encrypt the session key we created in step 1. This simulates passing an encrypted session key to another user.

```
# encrypt the session key with the public RSA key
cipher_rsa = PKCS1_OAEP.new(recipient_publickey)
enc_session_key = cipher_rsa.encrypt(session_key)
```

Step 3:
Now we create an AES cipher using the unencrypted session key. This AES cipher is used to encrypt the data. We also generate a MAC tag to verify our data with and the nonce used during the encryption so we can use it again later when we decrypt the ciphertext.

```
# encrypt the data with AES session key
cipher_aes = AES.new(session_key, AES.MODE_EAX)
# get the ciphertext and also a MAC tag to verify the file
ciphertext, tag = cipher_aes.encrypt_and_digest(data)
nonce = cipher_aes.nonce
```

Step 4:
Now on our “receiver side” we need to import the private key. We then need to generate an rsa cipher with our private key with PKCS1 OAEP so we can decrypt the encrypted session key. We want to make sure we use the same session key so our AES decrypt works properly.

```
private_key = RSA.importKey(open('private.pem').read())
# decrypt session key with the private RSA key
cipher_rsa2 = PKCS1_OAEP.new(private_key)
session_key2 = cipher_rsa2.decrypt(enc_session_key)
```

Step 5:
Now we can generate a new aes cipher using the decrypted session key and also the nonce used.

```
# decrypt the data with AES session key
cipher_aes2 = AES.new(session_key2, AES.MODE_EAX, nonce)
data_decode = cipher_aes2.decrypt_and_verify(ciphertext, tag)
```

data_decode will be the original plaintext

Time Consumption Comparison

The code below is what I used to test the different algorithms. Both functions, test_eax and test_rsa are defined in the rsa.py module I created. They both encrypt the same 512 bit message. This code will run each function 500 times and then print the average function execution time.

```
aes_exec_times = timeit.timeit(test_eax, number=500)
print(f"AVG AES execution time: {aes_exec_times / 500}s")
rsa_exec_times = timeit.timeit(test_rsa, number=500)
print(f"AVG RSA execution time: {rsa_exec_times / 500}s")
```

Program Output

```
AVG AES execution time: 0.00022341319999999998s
AVG RSA execution time: 0.0309243414s
```

Dividing the average execution time of RSA by AES $\frac{AVG\ RSA\ time}{AVG\ AES\ time}$, we get approximately 136. This shows that encryption with RSA is quite a bit slower than AES by itself. It makes sense though because RSA key size is 2048 bits so encryption and decryption is inherently slower. It also involves prime calculations while AES is a lot of operations on matrixes.

Below are the two test functions

```
def test_eax():
    key = convert_to_bytes('2b7e151628aed2a6abf7158809cf4f3c')
    msg = convert_to_bytes('6bc1bee22e409f96e93d7e117393172aae2d8
    cipher = AES.new(key, AES.MODE_EAX)
    ciphertext, tag = cipher.encrypt_and_digest(msg)
    nonce = cipher.nonce

    d_cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    plaintext = d_cipher.decrypt_and_verify(ciphertext, tag)
```

```python
def test_rsa():
    data = convert_to_bytes('6bc1bee22e409f96e93d7e117393172aae2d8a571e
    recipient_publickey = RSA.importKey(open('public.pem').read())
    session_key = get_random_bytes(16)

    # encrypt the session key with the public RSA key
    cipher_rsa = PKCS1_OAEP.new(recipient_publickey)
    enc_session_key = cipher_rsa.encrypt(session_key)

    # encrypt the data with AES session key
    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    # get the ciphertext and also a MAC tag to verify the file
    ciphertext, tag = cipher_aes.encrypt_and_digest(data)
    nonce = cipher_aes.nonce


    private_key = RSA.importKey(open('private.pem').read())
    # decrypt session key with the private RSA key
    cipher_rsa2 = PKCS1_OAEP.new(private_key)
    session_key2 = cipher_rsa2.decrypt(enc_session_key)

    # decrypt the data with AES session key
    cipher_aes2 = AES.new(session_key2, AES.MODE_EAX, nonce)
    data_decode = cipher_aes2.decrypt_and_verify(ciphertext, tag)
```

Conclusion

Key Findings:

- CFB Mode is a difficult mode to implement because of the bitshifting and combining segment values. For example, a segment value of 8 creates ciphertext values of 8 bits. This was difficult to work with because all the other modes create fixed ciphertext output of 128 bits. Padding also works differently because of this, etc.
- Different modes of AES are good at different things. For example, ECB mode is really simple to use and is very fast. It also allows for parallelization.
    - ECB, OFB, and CTR mode all don't propagate errors. The other two modes do propagate errors though.
- Encrypting with PKCS#1 OAEP only allows you to encrypt messages of a few hundred bytes.
    - Because of this limitation we have to use RSA in addition with AES to encrypt data of arbitrary sizes
- I was expecting errors in the ciphertext to have a much greater impact on the plaintext.
    - After working with the diagrams of each mode though I realized that for these 5 modes, the errors only propagate at most 2 blocks.

Lessons Learned

- I learned how to work extensively with bytes objects in python
- I learned how to manipulate individual bits, how to create new byte objects, and in general how to perform lower level bit manipulations.
- I learned how to provide type hints with python and organize files into modules
- I learned that when working with a lot of hex values it can get very confusing. Drafting out examples on paper and annotating the diagrams of each mode is very helpful for figuring out errors and what to code.
- Naming variables as closely to the names as they are in the mode diagrams is also very useful so that the logic in the code flows as similarly as the diagram.

Unexpected Results

- I did not expect RSA encryption to be 100 times slower than AES. I did expect it to be slower because the key's are much larger and there are intermediate steps of encryption, but not by this much.
- Errors in some modes only affect the ciphertext slightly. For example, I implemented a single bit flip to show how that could change an entire plaintext. However, in some modes, the bit flip in the ciphertext only showed up as a bit flip in the plaintext rather than changing the entire block.
    - This makes sense though, since if we flip a bit in the ciphertext it will only affect that bit in the plaintext. This is due to the XOR operation "undoing" anything that was previously XORed.

Citations

General Documentation

https://xilinx.github.io/Vitis_Libraries/security/2021.2/guide_L1/internals/cfb.html
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf
https://csrc.nist.gov/csrc/media/projects/cryptographic-algorithm-validation-program/documents/aes/aesavs.pdf
https://pycryptodome.readthedocs.io/en/latest/src/introduction.html
https://docs.python.org/3/

Stack Overflow

https://stackoverflow.com/questions/6624453/whats-the-correct-way-to-convert-bytes-to-a-hex-string-in-python-3
https://stackoverflow.com/questions/58437353/how-can-we-efficiently-check-if-a-string-is-hexadecimal-in-python
https://stackoverflow.com/questions/2340319/python-3-string-to-hex
https://stackoverflow.com/questions/58966410/how-to-increment-a-python-bytes-object
https://stackoverflow.com/questions/20910653/how-to-shift-bits-in-a-2-5-byte-long-bytes-object-in-python
https://stackoverflow.com/questions/15254195/python-how-to-add-space-on-each-3-characters
https://stackoverflow.com/questions/8220801/how-to-use-timeit-module
https://stackoverflow.com/questions/28130722/python-bytes-concatenation

Youtube Videos

Counter Mode - Applied Cryptography
Cipher Feedback Mode(CFB) | Algorithm Modes in Cryptography
Cipher Feedback (CFB)