CS 454 Final Project

Madeline Veric

December 12, 2023

1 AES Encryption and Decryption

1.1 Key Generation and Preparing a Message

To start off with AES, we first need to generate a key. In order to generate a key for AES encryption and decryption, we need to get a random byte string of length N where N is the number of bytes per block. In this case, we use 16 since the program follows an AES-128 bit structure. We can then make any message by taking a string literal and turn it into a sequence of octet bytes by applying the byte operator to it, which can be seen below.

```
# Key used for ECB Mode
key = get_random_bytes(16)

# Keys used for CBC, CFB, and OFB Modes.
iv_cbc = get_random_bytes(AES.block_size)
iv_cfb = get_random_bytes(AES.block_size)
iv_ofb = get_random_bytes(AES.block_size)

# Nonce used for CTR Mode.
nonce_ctr = get_random_bytes(AES.block_size // 2)

# Plaintext string message to encrypt, replace with what you want it to be.
plaintext = b"This is a multiple-block long message.  It spans multiple blocks and is already the proper size."
```

Figure 1: Key and Message Generation in Main

1.2 ECB Encryption and Decryption

ECB mode or Electronic Code-book is a AES mode where each block of plain-text is encrypted independently of any other block. It is semantically insecure due to its correlation between blocks. The encryption for this method expects data to have a length multiple of the block size (16 in this case due to our key being 16 bytes of length). We use ECB because it is simple but does not hide its pattern well and identical plain-text blocks produce identical cipher-text blocks.

```
def ecb_encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))
    return ciphertext

def ecb_decrypt(ciphertext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
    return plaintext
```

Figure 2: Definition of ECB Encryption and Decryption Functions

```python
# ECB Mode
ecb_ciphertext = ecb_encrypt(plaintext, key)
ecb_decrypted = ecb_decrypt(ecb_ciphertext, key)
print(f"ECB Decrypted: {ecb_decrypted.decode()}")
```

Figure 3: Use of ECB encryption and decryption functions in main.

1.3 CBC Encryption and Decryption

CBC mode or Cipher block Chaining is a AES mode where each block of plain-text is XORed with the previous cipher-text block before encryption. The only exception to this is when we originally use this mode and we need to read in an iv or Initialization Vector Value in order to start the encryption mode. It is mainly used to provide diffusion while makes identical plaint-text blocks produce different cipher-text blocks and offers better security overall compared to ECB.

```python
def cbc_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(plaintext), block_size):
        block = plaintext[i:i+block_size]

        # Padding the block before XOR
        padded_block = pad(block, block_size)

        xor_result = bytes(a ^ b for a, b in zip(padded_block, previous_block))
        encrypted_block = cipher.encrypt(xor_result)
        ciphertext += encrypted_block
        previous_block = encrypted_block

    return ciphertext

def cbc_decrypt(ciphertext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    decrypted_text = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(ciphertext), block_size):
        block = ciphertext[i:i+block_size]
        decrypted_block = cipher.decrypt(block)
        xor_result = bytes(a ^ b for a, b in zip(decrypted_block, previous_block))
        decrypted_text += xor_result
        previous_block = block

    return unpad(decrypted_text, block_size)
```

Figure 4: Definition of CBC Encryption and Decryption Functions

```python
# CBC Mode
iv_cbc = get_random_bytes(AES.block_size)
cbc_ciphertext = cbc_encrypt(plaintext, key, iv_cbc)
cbc_decrypted = cbc_decrypt(cbc_ciphertext, key, iv_cbc)
print(f"CBC Decrypted: {cbc_decrypted.decode()}")
```

Figure 5: Use of CBC encryption and decryption functions in main.

1.4 CFB Encryption and Decryption

CFB or Cipher Feedback is a AES mode where the block cipher turns into a stream cipher. This happens because each bytes of plain-text is XORed with a byte taken from a key-stream which results in the cipher-text. The previous cipher-text block is then encrypted and XORed with the plain-text to produce the next block. The key-stream mentioned is obtained on a byte basis, that is that the plain-text is broken into segments from anywhere between 1 byte up to the entire size of the block. Each segment is then encrypted with the block cipher the last piece of cipher-text produced so far in order to obtain the key-stream. If it is not possible, then we have the IV stream to go back to, such as in the case where in the 1st cycle there is no cipher-text yet. It's main purpose is to provide a way for encryption of individual bits or bytes which makes it possible to stream data. It also limits errors to the block that the error is contained in as well.

```python
def cfb_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(plaintext), block_size):
        encrypted_block = cipher.encrypt(previous_block)
        block = plaintext[i:i+block_size]
        xor_result = bytes(a ^ b for a, b in zip(block, encrypted_block))
        ciphertext += xor_result
        previous_block = xor_result

    return ciphertext

def cfb_decrypt(ciphertext, key, iv):
    return cfb_encrypt(ciphertext, key, iv)  # CFB decryption is the same as encryption
```

Figure 6: Definition of CFB Encryption and Decryption Functions

```python
# CFB Mode
iv_cfb = get_random_bytes(AES.block_size)
cfb_ciphertext = cfb_encrypt(plaintext, key, iv_cfb)
cfb_decrypted = cfb_decrypt(cfb_ciphertext, key, iv_cfb)
print(f"CFB Decrypted: {cfb_decrypted.decode()}")
```

Figure 7: Use of CFB encryption and decryption functions in main.

1.5 OFB Encryption and Decryption

OFB or Output Feedback is an AES mode where it shares a similar key-stream pattern to CFB. Each byte of plain-text is yet again XORed with a byte taken from the key-stream in order to result with the cipher-text. The difference is that the key-stream is obtained by recursively encrypting the IV value. Its purpose is to allow for pre-computation of the key-stream which makes it suitable for streaming data. Errors in this mode are also only propagated in its own block and do not affect decryption beyond the point of error.

```python
def ofb_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(plaintext), block_size):
        encrypted_block = cipher.encrypt(previous_block)
        block = plaintext[i:i+block_size]
        xor_result = bytes(a ^ b for a, b in zip(block, encrypted_block))
        ciphertext += xor_result
        previous_block = encrypted_block

    return ciphertext

def ofb_decrypt(ciphertext, key, iv):
    return ofb_encrypt(ciphertext, key, iv)  # OFB decryption is the same as encryption
```

Figure 8: Definition of OFB Encryption and Decryption Functions

```python
# OFB Mode
iv_ofb = get_random_bytes(AES.block_size)
ofb_ciphertext = ofb_encrypt(plaintext, key, iv_ofb)
ofb_decrypted = ofb_decrypt(ofb_ciphertext, key, iv_ofb)
print(f"OFB Decrypted: {ofb_decrypted.decode()}")
```

Figure 9: Use of OFB encryption and decryption functions in main.

1.6 CTR Encryption and Decryption

CTR or Counter is an AES mode that turns the block cipher into a stream cipher as well. Each byte of plain-text is XOR-ed with a byte from the key-stream which results in the cipher-text. However, the key-stream is generated with a sequence of counter blocks with ECB mode. A counter block is exactly as long as the cipher block (16 bytes in this case) and consists of two separate pieces. The first piece of the counter block is a fixed nonce which is set at initialization and functions as our IV value or the value at which the counter starts. The second value of a counter block is the counter variable which gets increased by one for subsequent counter blocks and is big endian encoded. The main purpose of counter is to provide an efficient way to encrypt along with having a way to have random access and streaming applications.

```python
def ctr_encrypt(plaintext, key, nonce):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    for i in range(0, len(plaintext), block_size):
        counter_block = nonce + (i // block_size).to_bytes(8, byteorder='big')
        encrypted_counter_block = cipher.encrypt(counter_block)
        block = plaintext[i:i+block_size]
        xor_result = bytes(a ^ b for a, b in zip(block, encrypted_counter_block))
        ciphertext += xor_result

    return ciphertext

def ctr_decrypt(ciphertext, key, nonce):
    return ctr_encrypt(ciphertext, key, nonce)  # CTR decryption is the same as encryption
```

Figure 10: Definition of CTR Encryption and Decryption Functions

```python
# CTR Mode
nonce_ctr = get_random_bytes(8)
ctr_ciphertext = ctr_encrypt(plaintext, key, nonce_ctr)
ctr_decrypted = ctr_decrypt(ctr_ciphertext, key, nonce_ctr)
print(f"CTR Decrypted: {ctr_decrypted.decode()}")
```

Figure 11: Use of CTR encryption and decryption functions in main.

2 Introducing Errors and Expectations

I will introduce errors in the cipher-text using a bit-flip method. This was done using a function called introduce error that takes in a cipher-text block, select a random bit in a random block, and flips said bit. This is the preferred way for Error propagation because we can clearly see the error and how many blocks it will propagate to. We expect the following in terms of error propagation:

- ECB - ECB decryption should not propagate errors as each block functions independently.
- CBC - CBC decryption should propagate in Pi and Pi+1, or two blocks.
- OFB - OFB decryption should not propagate the error. There will be an error in one bit of one block in the cipher-text.
- CFB - CFB decryption should propagate until synchronization is restored, which means it will vary depending on which bit gets flipped.
- CTR - CTR decryption should not propagate the error, refer to OFB.

3 Error Propagation Analysis

As we can see in the following results, the error propagation analysis done in the previous section is correct and matches my own expectations. The results vary on compilation depending on where the error gets propagated in the three block message, so the rates of errors may be off by one sometimes if the error propagates to the last possible block of text. However, in the example shown below, we can see that it propagated in the 1st block and the expected results we're seen.

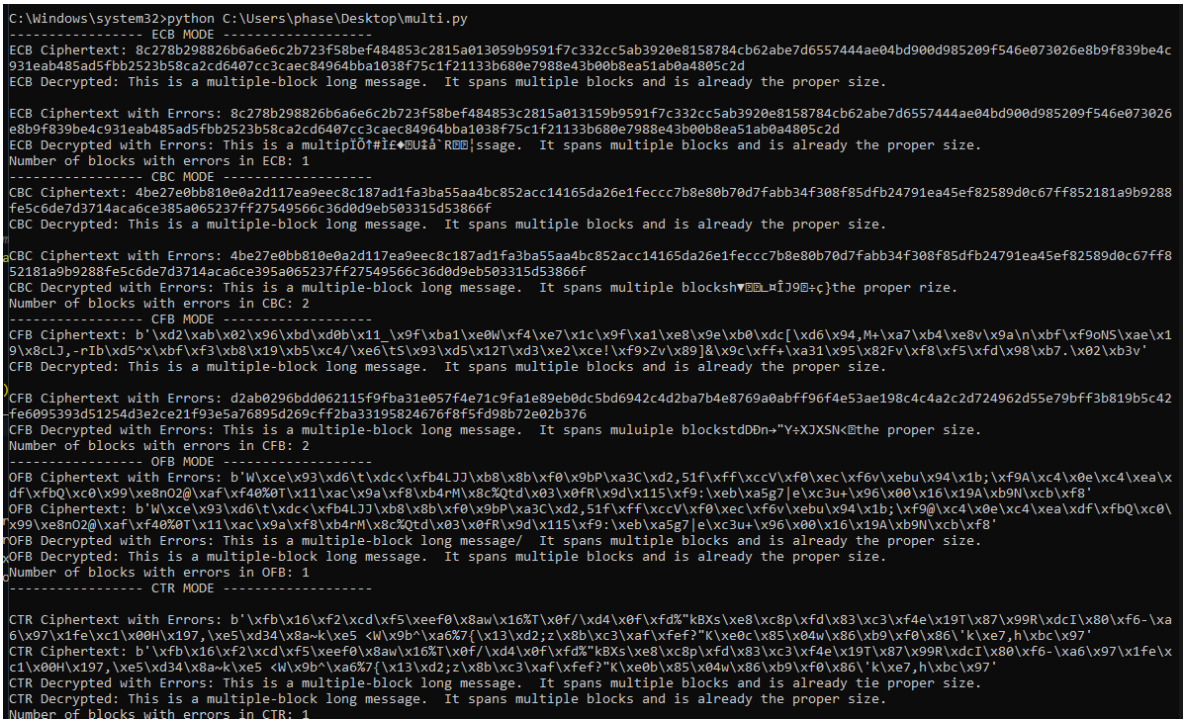

Figure 12: Error Propagation shown in action along with comparison of blocks.

Figure 13: Error Propagation shown in action along with comparison of blocks.

4 RSA Encryption,Decryption, and Analysis

RSA is a asymmetric algorithm, which means it takes two different keys in two different steps. These keys are called the public key which is given to everyone, and a private key which is kept private. RSA works by going off the idea that it is very difficult to factorize a large integer. It is mainly used for

```
print("----------------- ECB MODE --------------------")
ecb_ciphertext = ecb_encrypt(plaintext, key)
ecb_decrypted = ecb_decrypt(ecb_ciphertext, key)

# Introduce errors in ECB Mode
ecb_ciphertext_with_errors = introduce_error(ecb_ciphertext)
ecb_decrypted_with_errors = ecb_decrypt(ecb_ciphertext_with_errors, key)

print("ECB Ciphertext:", ecb_ciphertext.hex())
print("ECB Decrypted:", ecb_decrypted.decode())
print()

print("ECB Ciphertext with Errors:", ecb_ciphertext_with_errors.hex())
print("ECB Decrypted with Errors:", ecb_decrypted_with_errors.decode('latin1'))
error_block_count = count_error_blocks_in_plaintext(plaintext.decode(), ecb_decrypted_with_errors.decode('latin1'))
print("Number of blocks with errors in ECB:", error_block_count)

# CBC Mode
print("----------------- CBC MODE --------------------")
cbc_ciphertext = cbc_encrypt(plaintext, key, iv_cbc)
cbc_decrypted = cbc_decrypt(cbc_ciphertext, key, iv_cbc)

# Introduce errors in CBC Mode
cbc_ciphertext_with_errors = introduce_error(cbc_ciphertext)
cbc_decrypted_with_errors = cbc_decrypt(cbc_ciphertext_with_errors, key, iv_cbc)

print("CBC Ciphertext:", cbc_ciphertext.hex())
print("CBC Decrypted:", cbc_decrypted.decode())
print()

print("CBC Ciphertext with Errors:", cbc_ciphertext_with_errors.hex())
print("CBC Decrypted with Errors:", cbc_decrypted_with_errors.decode('latin1'))
error_block_count = count_error_blocks_in_plaintext(plaintext.decode(), cbc_decrypted_with_errors.decode('latin1'))
print("Number of blocks with errors in CBC:", error_block_count)
```

Figure 14: Error Propagation code of ECB and CBC showing Encryption and Decryption

confidentiality and authentication through digital signatures. Here are the generalized steps for RSA encryption and Decryption:

- Primes - We first need to start off by selecting two different prime numbers, p and q and their product n. We choose primes because it is difficult to determine the product of n and the two primes that compute to give us N.
- Public Key - We then need to calculate n (modulus) by multiplying the two prime numbers we choose. It is generally a good idea to have a n that spans for thousands of bits of data. The public key also need to have a component of e, which is an exponent usually decided by two participants.
- Secret Key - The RSA algorithm is known to use a Euler function of n to calculate the secret key, which is (p-1) * (q-1), following that they are both different prime numbers. We also need that the value of this Euler function is co-prime to e and only has a gcd of 1. The entire formula is know to be e x d = 1 mod (euler(n)).
- Encryption and Decryption - A message is encrypted with a public key by calculating m, which is known to be m to the e mod n, where m is the message you want to encrypt. Then, in order to decrypt with the private key, we will do m to e time d mod n. This only works since RSA exploits the property that any x to the a is equal to x to the b mod n where a is equal to b mod euler(n) and vice versa.

We can see in the figure above us that AES encryption and decryption is much faster than RSA encryption and decryption on average. This is mainly due to the fact that RSA is more computationally intensive than AES, and therefore much slower. RSA's main purpose is to encrypt only small amounts of data. RSA is also much more complex and tends to be more secure, but this security that it provides comes at the cost of being slower.

```python
# CFB Mode
print("---------- CFB MODE ----------")
cfb_ciphertext = cfb_encrypt(plaintext, key, iv_cfb)
cfb_decrypted = cfb_decrypt(cfb_ciphertext, key, iv_cfb)

cfb_ciphertext_with_errors = introduce_error(cfb_ciphertext)
cfb_decrypted_with_errors = cfb_decrypt(cfb_ciphertext_with_errors, key, iv_cfb)

print("CFB Ciphertext:", cfb_ciphertext)
print("CFB Decrypted:", cfb_decrypted.decode())
print()

print("CFB Ciphertext with Errors:", cfb_ciphertext_with_errors.hex())
print("CFB Decrypted with Errors:", cfb_decrypted_with_errors.decode('latin1'))
error_block_count = count_error_blocks_in_plaintext(plaintext.decode(), cfb_decrypted_with_errors.decode('latin1'))
print("Number of blocks with errors in CFB:", error_block_count)

print("---------- OFB MODE ----------")
# OFB Mode
ofb_ciphertext = ofb_encrypt(plaintext, key, iv_ofb)
ofb_decrypted = ofb_decrypt(ofb_ciphertext, key, iv_ofb)

ofb_ciphertext_with_errors = introduce_error(ofb_ciphertext)
ofb_decrypted_with_errors = ofb_decrypt(ofb_ciphertext_with_errors, key, iv_ofb)

print("OFB Ciphertext with Errors:", ofb_ciphertext_with_errors)
print("OFB Ciphertext:", ofb_ciphertext)

print("OFB Decrypted with Errors:", ofb_decrypted_with_errors.decode())
print("OFB Decrypted:", ofb_decrypted.decode())
error_block_count = count_error_blocks_in_plaintext(plaintext.decode(), ofb_decrypted_with_errors.decode('latin1'))
print("Number of blocks with errors in OFB:", error_block_count)

print("---------- CTR MODE ----------")
# CTR Mode
ctr_ciphertext = ctr_encrypt(plaintext, key, nonce_ctr)
ctr_decrypted = ctr_decrypt(ctr_ciphertext, key, nonce_ctr)
print()

ctr_ciphertext_with_errors = introduce_error(ctr_ciphertext)
ctr_decrypted_with_errors = ctr_decrypt(ctr_ciphertext_with_errors, key, nonce_ctr)

print("CTR Ciphertext with Errors:", ctr_ciphertext_with_errors)
print("CTR Ciphertext:", ctr_ciphertext)

print("CTR Decrypted with Errors:", ctr_decrypted_with_errors.decode())
print("CTR Decrypted:", ctr_decrypted.decode())
error_block_count = count_error_blocks_in_plaintext(plaintext.decode(), ctr_decrypted_with_errors.decode('latin1'))
print("Number of blocks with errors in CTR:", error_block_count)
```

Figure 15: Error Propagation code of CFB, OFB, and CTR showing Encryption and Decryption

```
C:\Windows\system32>python C:\Users\phase\Desktop\Time.py
AES Encryption Time: 0.0063572999788448215
AES Decryption Time: 0.006678699981421232
RSA Encryption Time: 0.5214917999692261
RSA Decryption Time: 3.1981970999622717
```

Figure 16: RSA and AES ECB Mode time comparison run n=1000 time with timeit Function

5 Conclusion

There was a lot of unexpected results that came from coding, especially when having to change from doing singular block messages to multiple block messages. The padding would have the chance to get corrupted and would usually cause errors in the encryption and decryption process because a change in padding lead to a change in size. There are also unexpected results when it comes to error propagation in CFB mode as it is not always consistent.

What I learned from this process is that AES encryption and decryption is quite the complex

process that involves many steps. While it is complex, it used to provide a great way to provide internet safety. However, there is also a lot that goes into which AES mode you choose and the advantages and disadvantages that comes with each mode. I also learned that RSA is very secure despite it being slower for larger data sets. Especially when dealing with multiple block codes, we can run into the slower and slower speeds the more we use RSA.

6 Sources

RSA Encryption vs AES Encryption: What Are the Differences? by Precisely
- RSA Step by Step by CrypTool
- RSA Algorithm Crptography bs Geeks for Geeks
- Modes of Operation of a Block Cipher by Bart Preneel Prof.
- Error Propagation in CBC Mode by Cryptography Stack Exchange
- What is the Point of the Nonce in CTR Mode by Information Security Stack Exchange
- Crypto.Random Pack by PyCryptodome.
- Classic modes of Operation by PyCryptodome
- Block Cipher Modes of Operations by Wikipedia
- The Use of Encryption mode with Symmetric Block Cipher by Rob Stubbs