Diego Christopher Garay
Dr. Phoenix Fang
CPE 425 - 01
18th January 2023

Lab 1 - Symmetric Key Cryptography Implementation and Study

Github Repository: https://github.com/dcgaray/cpe425-lab1

Part 1[All code can be found in **encryption.py**]

Original Message: `secret_msg = b"this is the wireless security lab"`

AES Encrypted Message:

```
b"\x0fDC\x04\t'r\xc2L(\xf6\xc7\xe4\x1c\xae\xf9\x84\xd9\x82\x98\x19\xfb\xde\xbb\x99e\x00\x85q\
xeb\xd1\x8e\xd6”
```

RC4 Encrypted Message:

```
b'\x19MFWP:h\x90M#\xf6\x94\xb3\x06\xaf\xf9\x84\xd9\x83\x82N\xa8\xde\xb0\x987\x1a\x98(\xb
8\xd4\x87\xc0'
```

AES Cracking: In my approach for encrypting the original message with AES, I chose to use the EAX(Encrypt, Authenticate, and then Translate) [2] mode which is one of the most modern and secure versions of AES. Because I know AES has gone under various different revisions, and we are generally tasked with breaking the older version of AES, I thought it would be interesting to try one of the more modern modes of encryption. Apart from the general key used to encrypt the plaintext message, it also uses a “number only used once” (nonce) which makes decrypting the message without these features even harder. Generally, for this type of encryption method, one could brute-force crack the encrypted message by trying every possible permutation of numbers for the given key length. However, this is very time-consuming as well as computationally expensive. Moreso, given the use of the nonce, it would be nearly impossible to crack the message via brute-force. Therefore, I was not able to crack the encrypted message above.

RC4 Cracking: Similar to the AES encryption, I attempted to brute-force crack the RC4 encrypted message by trying every possible permutation of 40-bit length keys. Because the RC4 encryption method only uses a key(and no nonce) it is theoretically a lot easier to brute-force an encrypted message. However, even with this ease in cracking, it is still very time-consuming and computationally expensive to try every single key. I let my program run for about half an hour before I stopped it. As it is, my code will start at the lowest possible value b‘\x0\x0\x0\x0\x0’ for the key and work its way. Given that I knew the key itself was b’\xff\xff\xff\xff\xff’, I knew it would be a long time until my code would crack the message. However, I am confident my code would work, and I have thus made sure to include the function that does the cracking.

<u>Part 2</u>[All code can be found in symmetric.py]

	ECB	CBC	CFB	OFB	CTR
Pattern Preservation	Yes	No	No	No	No
Error Propagation	No	NO	Yes	Yes	Yes

<u>Pattern Preservation</u>

In order to see if any patterns were being preserved in ciphertexts, I ended up splitting all of our byte strings into byte arrays. Using this Byte Array, I created a dictionary object that held the amount of times that specific byte was encountered within a resulting ciphertext. Given that there are 255 possible values a single byte could take, I decided that if a byte showed up more than 4 times, it was being repeated uncharacteristically. Using this method, I was able to detect that the ECB mode of AES was preserving ciphertext patterns, while no other mode was. However, this is to be expected as ECB will break up a plaintext message into individual blocks, and these blocks will be encrypted using the same key. Therefore, if I had patterns within my plaintext, they would show up in the ciphertext.

<u>Error Propagation</u>

In order to test the different modes for Error Propagation, I first had to introduce errors into each mode's resulting ciphertext. To ensure that any error I introduced was guaranteed to propagate errors(if the AES Mode would propagate errors) I changed the 13th bit of the first block of ciphertext. Thus, if that block was then used to decrypt the subsequent blocks of ciphertext, the error would cascade and be visible.

In terms of my findings, I was not surprised to see that the ECB mode would not propagate errors as each block is individually encrypted, and a change in a singular block should not introduce a change in any other block. However I was surprised to see that changing a bit in the first block of the CBC encrypted message would not introduce any error. Because each block of ciphertext is XORed with the subsequent ciphertext block, I would assume an error would form. However, upon further research, this is actually an interesting property of CBC in which the ciphertext will actually heal itself if a bit is flipped, and at most an error would only propagate for 2 blocks.[3]

Questions

1.

	Advantages	Disadvantages
AES	● Because of the 128 bit key length, it is able to encrypt large chunks of data at once ● Given it's age, the modern version of AES is considered to be generally secure	● AES is a fixed-block cipher, and this can cause issues where information that is being encrypted/decrypted is not correctly padded which makes the algorithm more computationally expensive
RC4	● Because RC4 utilizes 128 bit keys, it can encrypt/decrypt information at a high rate ● It is a stream cipher and only encrypts/decrypts one byte at a time, it is useful for quickly encrypting/decrypting small amounts of data	● There are many known security vulnerabilities associated with RC4, with the most notable of these being that the first few bytes of the keystream are easily predicated

2. If i was encrypting a file, I would personally chose to use AES both because it known to be more secure than AES, but also because it would be able to handle medium-large files faster than RC4
3. Cracking AES and RC4 was somewhat frustrating. Because I know it was always possible to brute-force crack the algorithms, I was hoping it would be relatively simple to implement this. However, my computer simply does not have enough processing power to efficiently try every permutation of the keys used to encrypt the information. I did not give much time into trying to crack AES because of how much more randomization was involved, but for RC4 I was hoping a multi-threaded approach would work to crack the encrypted message in a small amount of time. However, when I tried to use 4 threads, my terminal crashed…Given my lack of experience with multi-threading, I took it as a sign that no matter how hard I tried, it would not be worth my time to look for a way to speed up the cracking when our key would not be randomized, and would alway be b'\xff\xff\xff\xff\xff'

4. Because of the different security vulnerabilities we demonstrated, none of these modes of AES are actually in use anymore. However, when they were used, they all served a variety of use cases, and each had their own unique features.

	Features	Uses
ECB	-The most primitive mode of AES -Each block of ciphertext is separately encrypted -Very vulnerable to repeating block attacks	-Easy to implement
CBC	-Each block of ciphertext is used to encrypt subsequent blocks of ciphertext -Requires an initialization vector	-Widely used in different protocols such a SSL, TLS, SSH, and IPSec
CFB	-Each block of ciphertext is used to encrypt subsequent blocks of ciphertext, but in smaller blocks of data than CBC -Requires an initialization vector	-Used SSL, TLS, SSH, and IPSec protocols
OFB	-Encrypts the plaintext one block at a time, and uses feedback in between each block to further introduce randomness that cannot be easily replicated into the cipher text -Requires an initialization vector	-Used heavily for datastreams, such as network traffic
CTR	-Encrypts the plaintext one block at a time, and uses a unique counter value for each block -Requires a Nonce	-Widely used in SSL, TLS, SSH, and IPSec

Works Cited

[1] Pycryptodome. “Classic Modes of Operation for Symmetric Block Ciphers¶.” Classic Modes of Operation for Symmetric Block Ciphers - PyCryptodome 3.170b0 Documentation, https://www.pycryptodome.org/src/cipher/classic#ecb-mode.

[2] Wikipedia. “Authenticated Encryption.” *Wikipedia*, Wikimedia Foundation, 30 Dec. 2022, https://en.wikipedia.org/wiki/Authenticated_encryption.

[3] Burda, Karel. (2006). Error Propagation in Various Cipher Block Modes. International Journal of Computer Science and Network Security. 6. 235 - 239.