Applied Crypto

Block Cipher and Modes of Operations

This project is focused on the block cipher and the various modes of operations.

Problem 1 This is not a code exercise. Students do it manually

Let E be a block cipher of block length 4 that $E_k(b_1b_2b_3b_4) = (b_2b_3b_1b_4)$.

a) The following table is used to convert (or encode) an English plaintext into a bit string.

A	B	C	D	E	F	G	H
0000	0001	0010	0011	0100	0101	0110	0111
I	J	K	L	M	N	O	P
1000	1001	1010	1011	1100	1101	1110	1111

b) Given a plaintext message p = 'FOO', encrypt it manually using the following operation modes: 1) ECB, 2) CBC mode with IV = 1010, 3) CTR mode with ctr = 1010. Make sure you have the final ciphertexts in letters that are easy to transmit to the receiver.

c) The receiver received the ciphertexts in letters and decrypted them, respectively.

Problem 2 This is a python code exercise to implement the so called "Meet-in-the-Middle Attack" (https://en.wikipedia.org/wiki/Meet-in-the-middle_attack).

It is a cryptanalytic attack applicable to ciphers based on composition of multiple rounds of substitutions and permutations. It works by finding plaintext-ciphertext pairs that map to the same intermediate value after partial encryption/decryption. We can perform an exhaustive search on a simplified cipher by splitting the secret key into independent parts.

We are going to use the simplified AES (SAES). The class slides give the details of SAES. It starts with key expansion, then works on encryption to get ciphertext and on decryption to recover the plaintext. The key expansion generates three keys. It has 2 rounds.

The first key, Key0, is used for the add round key to the plaintext.

The second key, Key1, is used to perform Round 1 transformation on state, defined as encrypt_round1()

The third key, Key2, is used to perform Round 2 transformations on state, defined as encrypt_round2()

For the decryption,

Key2 is used to perform inverse Round 2 transformations on ciphertext, defined as decrypt_round2()

Key1 is used to perform inverse Round 1 transformations on state, defined as decrypt_round1()

Key0 is to do add round key.

So, the pseudo-code would be:

1. Get (Key0, Key1, Key2) from Key K using the key expansion.
2. Encrypt a plaintext to get the ciphertext.
3. Assume we can get Key0 easily. So we do not worry the initial add-round key operation. And we can assume the first round start with the plaintext.
4. Partially encrypt the plaintext with just Key1 to get intermediate state.
5. Partially decrypt ciphertext with just Key2 to get intermediate state.
6. Compare intermediate states to find matching Key1 and Key2

The following code skeleton may help students in their coding.

Meet in the middle attack code skeleton:

```
# Generate all possible keys for rounds 1 and 2
#n = size of key space
keys1 = []
keys2 = []
for k1 in range(n):
 for k2 in range(n):
  keys1.append(k1)
  keys2.append(k2)

# with the Key, real_key is given
cipher = encrypt(plaintext, real_key) # AES or SAES

# Encrypt plaintext with all keys for round 1
round1_outputs = []
for k1 in keys1:
 intermediate = encrypt_round1(plaintext, k1)
 round1_outputs.append(intermediate)

# Decrypt ciphertext with all keys for round 2
round2_inputs = []
for k2 in keys2:
 intermediate = decrypt_round2(cipher, k2)
 round2_inputs.append(intermediate)

# Match round1 outputs with round2 inputs to find keys
for i, r1out in enumerate(round1_outputs):
 for j, r2in in enumerate(round2_inputs):
  if r1out == r2in:
   round1_key = keys1[i]
   round2_key = keys2[j]
   print("Found keys:", round1_key, round2_key)
```

Encrypt_round and Decrypt_round skeleton:

```
# Round 1
def encrypt_round1(plaintext, k1):
```

```
 state = add_round_key(plaintext, k1)
 state = sub_bytes(state)
 state = shift_rows(state)
 return state

# Round 2
def decrypt_round2(ciphertext, k2):
 state = shift_rows(ciphertext, inverse=True)
 state = sub_bytes(state, inverse=True)
 state = add_round_key(state, k2)
 return state

# Base utility functions
def sub_bytes(state, inverse=False):
 # Substitute bytes using S-box
 return state

def shift_rows(state, inverse=False):
 # Shift rows of state matrix
 return state

def add_round_key(state, key):
 # XOR state with round key
 return state
```

The base utility functions are well defined. Students shall be able to implement them. It is also fine if students use existing code somewhere. They need to credit the source.

Code guideline (exhaustive search):

a) Implement the simplified AES.
b) Write functions to partially encrypt plaintext with just the first round key and partially decrypt ciphertext with just the last round key.
c) Execute an exhaustive search to generate all possible first and last round keys.
d) Match the intermediate states from partial encryption and decryption to find plaintext-ciphertext pairs that reveal the round keys.
e) Given a plaintext-ciphertext pair encrypted with the full key, use the attack to recover the simplified round keys.

Test scenario 1: select key size = 16, the block size =16. This is the typical case on the simplified AES as explained in the slides. It is easy to get the possible key space is 2^{16} = 65536. The match two keys using exhausted method would be 65536x65536 = 4,294,967,296. Clearly, your testing machine needs to be powerful. If you cannot get the results fast enough, summarize it and move to the test scenario 2. That is the nature of cryptoanalysis.

Test scenario 2: select key size = 8, the block size =8. This is the reduced simplified AES (rSAES). In this case, the possible key space is 2^8 = 256. The match two keys using exhausted method would be 256x256 = 65536, a huge reduction in test cases. And it should be manageable.

But we need to work on the 2x2 matrix with each element having 2 bits. Let me define the four functions, add_round_key, shift_row, sub_2bit and mixed_column. Note, this is for academic purposes only. I am not aware any of such rSAES existed. Here we use the GF(2^2) with m(x) = $x^2 + x + 1$.

Add_round_key – bit wise XOR

Shift_row: shift the second row one positon.

Sub_2bit: Using the similar formula from AES, we define A = $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ and b=$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, and the sub is defined as $Aa^{-1}+b$. My calculation gives the S_table

	0	1
0	01	00
1	11	10

Mix_column: define M = $\begin{pmatrix} 01 & 10 \\ 10 & 01 \end{pmatrix}$, the mixed column Mc where c=$\begin{pmatrix} b_0 b_1 \\ b_2 b_3 \end{pmatrix}$. We can get the substitution.

$b_0 \oplus b_2 \oplus b_3$	$b_1 \oplus b_2$
$b_0 \oplus b_1 \oplus b_2$	$b_0 \oplus b_3$

Answer the following questions.

1. Make sure you have the code for test scenario 1 and the code for test scenario 2 as well.
2. Answer if you can find the matched keys in test scenario 2, or both? You can use a driver call to test your codes. The Cas have their own test cases.
3. Analyze the time and memory complexity of the attack.
4. Discuss how increasing the number of rounds exponentially increases the difficulty of the attack.

Submission

Use the jupyter notebook format to write the report. Send the link in submission and also the exported pdf. Make sure you give access to the CAs and me. You can use google collab (https://colab.research.google.com/) with google doc or the deepnote (https://deepnote.com/) or any other hosting you are familiar with that supports the basic notebook functionality defined by jupyter notebook. If you do not want to host anywhere in the cloud, you can submit the .ipynb file plus the exported pdf file.

ms14306@nyu.edu

jfk292@nyu.edu

zc3133@NYU.edu