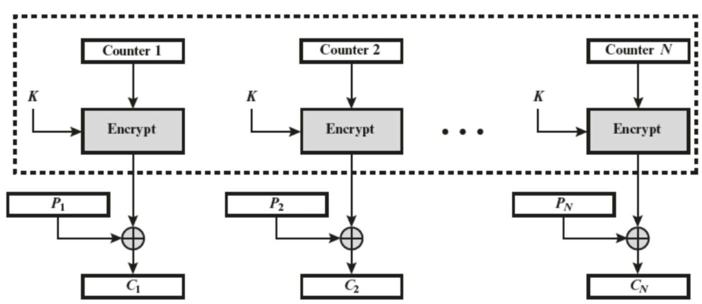
SIL 765 : Network Security
Assignment 1
Cryptanalysis

Mayank Badgotya
2021CS10583

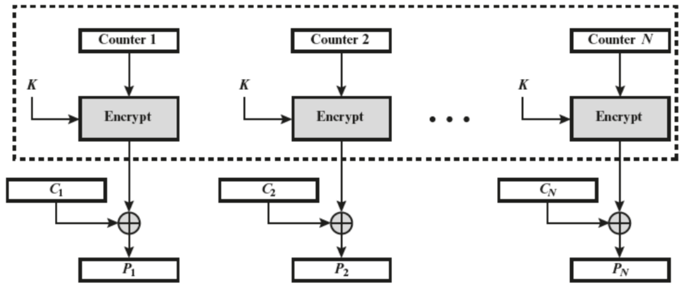
January 2024

1 AES:

- AES, which stands for Advanced Encryption Standard, is a widely used symmetric encryption algorithm.
- Steps taken in AES: KeyExpansion, SubBytes, ShiftRows, MixColumns, AddRoundKey.
- Above steps are done 10 times for a 128-bit key in our case, making ciphertext gibberish.
- We are given an AES machine working on CTR mode with the following cases:
  - 16 bit: In this 16 LSB are random and others are all zero.
  - 32 bit: In this 32 LSB are random and others are all zero.
  - 48 bit: In this 48 LSB are random and others are all zero.



Encryption in CTR Mode



Decryption in CTR Mode

- Due to the nonlinearity and randomness in AES. We will treat the AES mode as a black box.

2 Cracking the key:

2.1 16 bit:

- As $2^{16} = 65,536$ is easily computable in this time. We can brute force the key in $2^{16} = 65,536$ iterations.
- Obtained key: `b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xe9E'`
- I observed that 16 LSB's are `11101001 1000101`. Note number of ones and zeros are same.
- `Heuristic:` I made a harsh heuristic that the number of ones and zeros in the 16 LSB's will be the same.
- Comparing with and without heuristic brute force:

	Non-heuristic	Heuristic
Iterations	59717	12137
Time	2.374s	0.54s

2.2 32 bit:

- As $2^{32} = 4,294,967,296$ is easily computable in this time. We can brute force the key in $2^{32} = 4,294,967,296$ iterations.
- For this compute one a single core machine might take 1-2 days. So I parallelized the code and divided the key search space into 8 subspaces each spanning 2^{29} elements which ran for around 8 hours.
- Obtained key: `b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xed\xc5\xa5\x06'`
- 32 LSB's are `11101101 11000011 10100101 00000110`. Note the number of ones and zeros are consistent from previous observation.
- Comparing with and without heuristic brute force:

	Non-heuristic	Heuristic
Iterations	4,173,462,656	165,636,892
Time	8hrs	4hr

2.3 48 bit

- We can use the above heuristic to reduce the search space but it would take much more time to crack compared to the previous case.
- So I checked cryptanalysis which is explained in the below section.

3 Cryptanalysis:

It refers to observing the behavior between multiple plaintext and cipher text or exploiting mathematical properties on the model.

3.1 Brute Force

- Trying all possible keys. But this is not feasible for us as the key space is large.
- I tried this approach with a heuristic.

3.2 Differential Cryptanalysis

- Differential cryptanalysis examines the differences between pairs of plaintext-ciphertext pairs to deduce information about the key.
- In this approach we try to establish a relationship between responses of two messages where one message has just a few bits flipped from the other one.
- I tried this approach and was unable to find a solid relation with the key.

3.3 Linear Cryptanalysis and algebraic attacks

- Linear cryptanalysis exploits linear approximations in the relationship between the plaintext, ciphertext, and key.
- In this approach we check the internal S-box, matrix multiplication operation, and the key scheduling algorithm.
- All steps are invertible if the key is known. As xor is subtractive ,matrix multiplication is invertible and S-box elements are derived from `Galios field`.
- I looked into this but the number of variables grew in order of thousands so solving these linear equations would take time in `O(N^3)` where N is the number of variables which is infeasible to compute.

3.4 Side-Channel Attacks

- Side-channel attacks exploit information leaked during the execution of a cryptographic algorithm, such as power consumption, electromagnetic radiation, or timing information.
- As Oracle was on Gradescope I was unable to infer any extra information for this attack.

4 Code Explanation:

The following explanation is for the 16-bit case, other cases follows the same.

4.1 Non-heuristic brute force:

```
import pyaes

texts = [
    (b'abcdefghijklmno',b'WY\xa1\xb4\xfa\x86^Mu\x9a\x8c\x0b\xb1\x18\xd9'),
]

def crack(texts):
    iterations = 0
    N = 2**16
    for key in range(N):
        iterations += 1
        keystream = key.to_bytes(16,byteorder='big')
        passAll = True
        for plaintext,ciphertext in texts:
            decrypted = pyaes.AESModeOfOperationCTR(keystream).decrypt(ciphertext)
            if decrypted != plaintext:
                passAll = False
        if (passAll):
            return(key,iterations)
            return
    print("Failure!")
crack(texts)
```

- In this code I simply iterated over all keys which are denoted by variable N = 2^{16}
- Then iterated over all plaintext and ciphertext pairs and checked whether the key was valid for all texts.
- If it passes all pairs of texts successfully then return the key and iterations are done.
- Else echo 'Failure!'

4.2 Heuristic brute force:

```
import pyaes
L = list()

for i in range(256):
    for j in range(256):
        if ((bin(i)[2:].count('1')+bin(j)[2:].count('1')) == 8):
            L.append((i,j))

texts = [
    (b'abcdefghijklmno',b'WY\xa1\xb4\xfa\x86^Mu\x9a\x8c\x0b\xb1\x18\xd9'),
]

def crack():
    iterations = 0
    for (i1,i2) in L:
        iterations += 1
        key = 256*i1 +i2
        keystream = key.to_bytes(16,byteorder='big')
        passAll = True
        for plaintext,ciphertext in texts:
            decrypted = pyaes.AESModeOfOperationCTR(keystream).decrypt(ciphertext)
            if decrypted != plaintext:
                passAll = False
        if (passAll):
            print(key,iterations)
            return
    print("Failure!")
crack()
```

- This code block uses the heuristic mentioned above.
- First I took all pairs of numbers with are in the range [0,256] (i.e. 8 bits) and then cached all the pairs whose numbers of ones and number of zeroes count is same in binary notation.

- Notice here I picked the number whose number of one's count is 8 which automatically sets the number of zeroes to 8.
- This time we have to encode the pair of numbers together to a single number, which we can see in the code as $key = 256 * i1 + i2$
- Then iterated over all plaintext and ciphertext pairs and checked whether the key was valid for all texts.
- If it passes all pairs of texts successfully then return the key and iterations are done.
- Else echo 'Failure!'

5 More attacks tried:

5.1 Random attack:

- Choose a random key in the range and try it.

5.2 Cyclic input:

- I gave oracle a plaintext and got the corresponding ciphertext and then fed ciphertext to oracle and repeated this process 10 times.
- But no substantial information was revealed about key.

5.3 Frequency analysis:

- I checked what is the linear relation when we replace a ascii value by some amount.
- It was not constant, so checked what is frequency of linear difference (which was 3 in most cases for adjacent pair swaps when ascii value difference was 1 between characters). But couldn't infer any property of key through this.

6 Anomaly:

- I noticed an anomaly in oracle when I gave

  ```
  6;\xc3\x82\xc3\x90\xfa\x9e\xdeJ$\xf8\x8c\xb5(\x18 B\xb3\x91\n
      \x8a,\x8c\x90\x89\xcf\x99%cn\x12\x1e\xe8R\xb6
  ```

  as plaintext which is 288 bits and where the oracle automatically changed the plaintext to

  ```
  '6;\xc3\x83\xc2\x82\xc3\x83\xc2\x9␣0\xc3\xba\xc2\x9e\xc3\
      x9eJ$\xc3\xb8\xc2\x8c\xc2\xb5␣(\x18B\xc2\xb3\xc2\x91\n\xc2
      \x8a,\xc2\x8c\xc2\x90\xc2\x89\xc3\x8f\xc2\x99%cn\x12\x1e\
      xc3\xa8R\xc2\xb6'
  ```

  which is 448 bits long.

7 Note:

- As the code with which I cracked was not supposed to be submitted, in the `decipher_text.py`, I have used the `deciphered_key` directly for decrypting the message.
- You can find the code at Github

8 Refernces:

- Wikipedia
- AES review paper
- Cryptanalysis
- Pyaes
- Galios field