SP2023 Week 13 • 2023-04-20

Crypto III: Block Ciphers (AES)

Sagnik Chakraborty

Announcements

- Only 3 more meetings! Whaaaaattttt
- No meeting this Sunday
- Java Rev next Thursday with Suchit, Pete & Hassam

ctf.sigpwny.com

sigpwny{sauce_box}

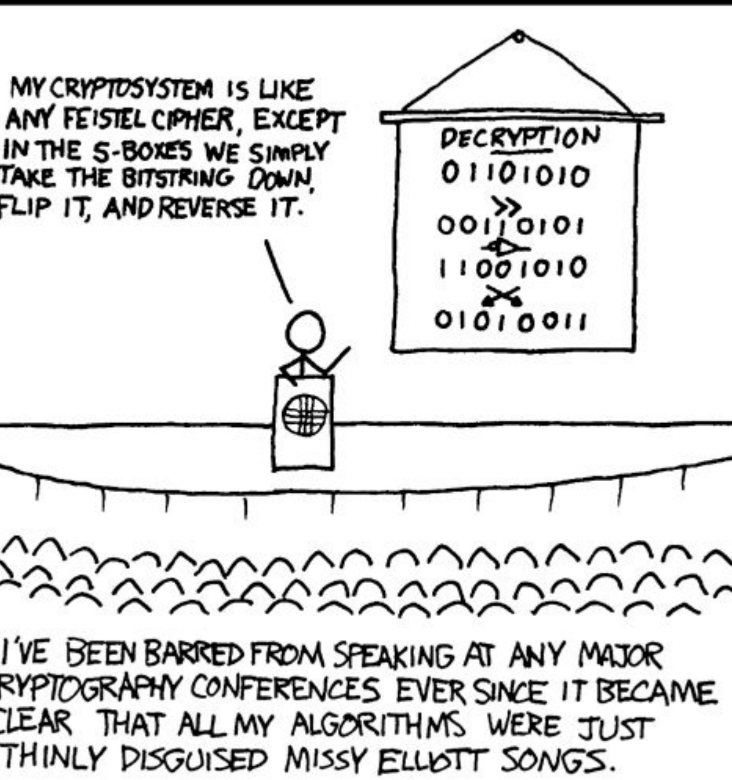


Table of Contents

- What are block ciphers
- AES algorithm, modes
    - Linear/Differential Cryptanalysis, Side channels (PO)
- An understanding of AES tends to lend itself well to other block ciphers!

Block Ciphers

- A type of deterministic encryption algorithm that operates on a plaintext of fixed length
- Typically, the plaintext is divided into “blocks” of 16 or 32 bytes
- An algorithm is used to transform each block into an enciphered block, and then the results are joined together to form a ciphertext

AES

- Block cipher that operates on a fixed block length of 16 bytes (128 bits)
- There are a total of 3 different bitlengths for the keys: AES-128, AES-192, AES-256
- For the sake of simplicity and due to its widespread use, we will stick with AES-128 for now. But the same ideas extend to higher key dimensions
- Encryption for AES-128 consists of 10 rounds of encryption, AES-192 is 12 rounds, AES-256 is 14 rounds

AES

There are 4 main components to AES encryption

- SubBytes uses a global substitution lookup table called the SBOX to substitute a set of bits in the current block, adding nonlinearity to the encryption
- ShiftRows shifts the rows of the current block by a certain offset amount, providing diffusion in the vertical direction.
- MixColumns applies a matrix multiplication operation to each column of the current block, providing diffusion in the horizontal direction.
- AddRoundKey performs a bitwise XOR operation between the current block and a round key derived from the cipher's key schedule, adding confusion to the process.

Math in a galois field - GF(2^8)

- GF(2^8) is represented by a characteristic polynomial P(x) = $x^8+x^4+x^3+x+1$
- Operations with bytes is analogous to polynomial operations!
  - So for example, \x34 = \b00100010 corresponds to x^5+x modulo P(x)
- Example Multiplication within GF(2^8):
  - {\x53} * {\xCA} = $(x^6+x^4+x+1)(x^7+x^6+x^3+x)$

$$= (x^{13}+x^{12}+\ldots+x^8+x^6+x^5+\ldots+x^2+x) \bmod (x^8+x^4+x^3+x+1)$$

= (11111101111110) mod (100011011)

= (000000001) mod (100011011)

= {\x01}

Math in GF(2^8)

```
 11111101111110 (mod) 100011011
^100011011
 01110000011110
 ^100011011
  0110110101110
  ^100011011
   010101110110
   ^100011011
    00100011010
     ^100011011
      000000001
```

Why AES uses GF(2^8)

- All elements in GF(2^8) are singular bytes, which allows efficient byte-by-byte operations
- AES operations are invertible in GF(2^8), enabling fast encryption and decryption
- Using a simpler modulus (eg. generic pow of 2) can cause loss of the high bit during multiplication by 2, making operations complex.
- The linear and affine operations in AES are non-commutative and prevent simple matrix multiplication

AES from a symbolic perspective

- Consider that we have the input 128 by 128 bit block represented as a 4 byte by 4 byte message block (32 bits/col)

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

- Notice that this is actually in column major order. We keep track of this array and call it the state array

AES from a symbolic perspective

- AES also keeps track of collections of 4 bytes, known as **words**.
- As with the input array, we can also consider the 128-bit key as a 4 byte by 4 byte array.
- AES uses a substitution-permutation network where substitutions occur at the byte level while permutations occur at the word level
- Each of the 4 column words from the input bits are expanded into a key schedule of 44 words.

AES from a symbolic perspective

$m_0 m_2 \ldots m_{128}$

AddRoundKey — $w_0 - w_3$

Round 1 — $w_4 - w_7$

Round 2 — $w_8 - w_{11}$

Encryption

Round 10 — $w_{40} - w_{43}$

$c_0 c_2 \ldots c_{128}$

key schedule

$m_0 m_2 \ldots m_{128}$

Round 10 — $w_0 - w_3$

Round 9 — $w_4 - w_7$

Round 8 — $w_8 - w_{11}$

Decryption

AddRoundKey — $w_{40} - w_{43}$

$c_0 c_2 \ldots c_{128}$

AES from a symbolic perspective

- Step 1: SubBytes:
  - A 16 by 16 global look up table called the SBOX is used to find a replacement byte given the current byte in the state array
    - State[byte] ← Sbox[byte] for each byte in the state
  - Entries of the are developed with the notion of multiplicative inverses in the field GF(2^8)
  - This multiplicative inversion is followed by an affine transformation which we must have because 0 will never have a multiplicative inverse (corresponds to the 0x63 byte in the Rijndael SBOX)
    - This also prevents you from stringing multiplicative inverses together
  - Overall, this is the only source of nonlinearity in AES: very important

AES from a symbolic perspective

- Step 2: ShiftRows:
  - The purpose of this step is to simply scramble the byte order within each 128-bit block.
  - Since we simply just permute the bytes within the rows, this operation is linear and invertible. Below is the representation of this on our state array

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \longrightarrow \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

AES from a symbolic perspective

- Step 3: MixColumns:
  - Each byte in the word columns are replaced by a function acting on that word
  - Each byte specifically is replaced by 2 times that byte + 3 times the next byte in the column plus the byte that follows and then the next
    - Keep in mind that we are still working in $GF(2^8)$ so each “plus” represents a simple XOR and we multiply in a similar spirit from earlier

$$\begin{bmatrix} x02 & x03 & x01 & x01 \\ x01 & x02 & x03 & x01 \\ x01 & x01 & x02 & x03 \\ x03 & x01 & x01 & x02 \end{bmatrix} \times \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

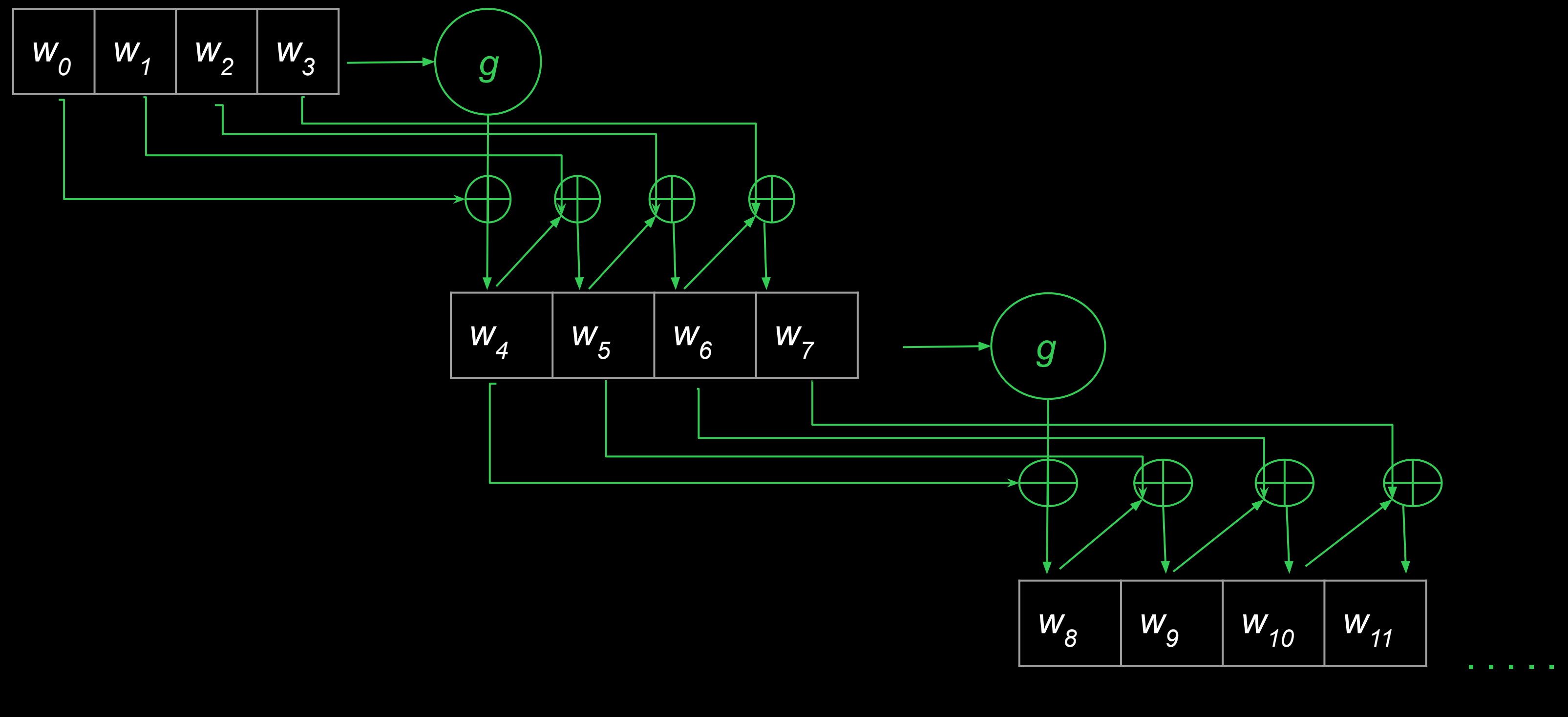
AES from a symbolic perspective

- Key Expansion and Step 4: AddRoundKey
    - From our key matrix

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix} \xrightarrow{\text{Key scheduling}} \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & \dots & w_{43} \end{bmatrix}$$

    - We originally have a group of 4 words from our key that we expand into 43 via a similar fashion to the next slide

AES from a symbolic perspective



Round keygen algorithm

- Suppose we have 4 words of the round key for the ith round,

  [w_i w_{i+1} w_{i+2} w_{i+3}]
- As from the figure from the last slide, if we want to find w_{i+6} w_{i+7} w_{i+8} and w_{i+9} we just follow the figure from above:

$$w_{i+5} = w_{i+4} \oplus w_{i+1} \qquad w_{i+6} = w_{i+5} \oplus w_{i+2} \qquad w_{i+7} = w_{i+6} \oplus w_{i+3}$$

- This leaves us with finding w_{i+4}
- The beginning of each round key is found by $w_{i+4} = w_i \oplus g(w_{i+3})$
- The group function g:
  - Performs a one-byte circular left shift on the argument 4 byte word
  - Performs a byte substitution from the SBOX
  - XORS the bytes from the previous step with a round constant

RCON array

- Let RCON[i] be the round constant for the ith round, the word [RC[i], x00, x00, x00]
- The RC part follows a simple DP:

$$RC[1] = x01$$

$$RC[i] = (x02 \cdot RC[i-1])_{GF(2^{**}8)}$$

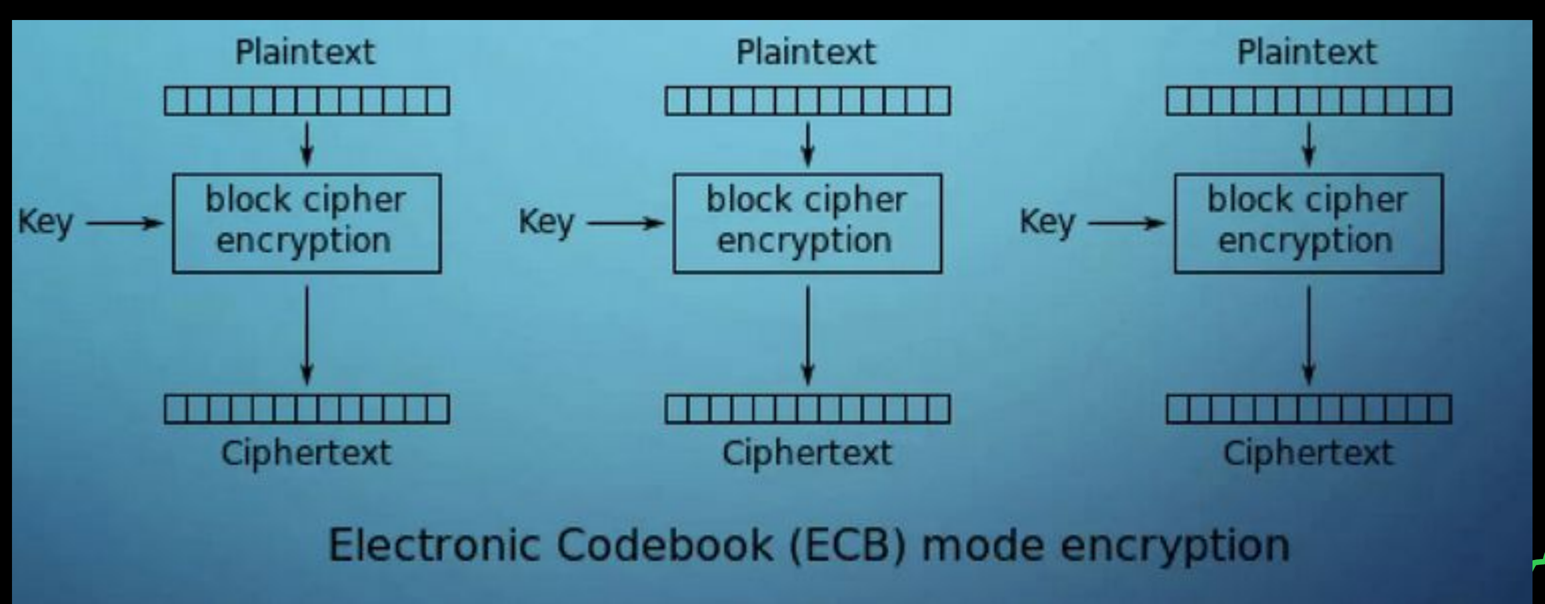
AES encryption

```
Encrypt(msg=M, key=K):=
    # key expansion
    round_keys = key_expansion(K)

    # initial round
    state = add_round_key(M, round_keys[0])

    # main rounds
    for i = 1 … 9:
        state = sub_bytes(state)
        state = shift_rows(state)
        state = mix_columns(state)
        state = add_round_key(state, round_keys[i])

    # final round
    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, round_keys[10])

    output ciphertext C = state
```

AES vs DES

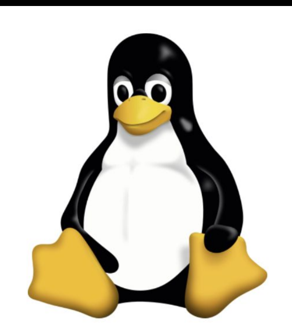
- AES is a key-alternating general permutation/substitution network operated block cipher, while DES is based on the Feistel structure of encryption
- In AES, each round applies a diffusion-achieving transformation to the entire block, followed by the round key.
- The transformation in AES may involve a combination of linear and nonlinear steps.
- In contrast, DES transforms only one half of the block in each round, based on S-boxes and the round key.
- Key alternating ciphers are amenable to theoretical analysis of security.

Cipher Modes - ECB

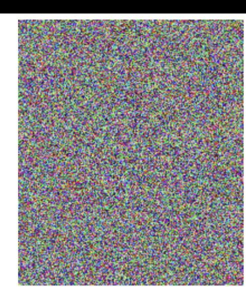


Cipher Modes - AES

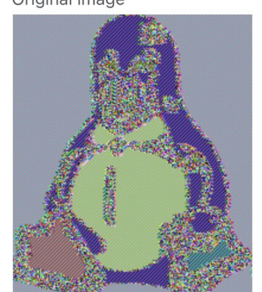
- Since each block of the input is encrypted independently (and not reliant on previous outputs/inputs), it is less secure



Original image

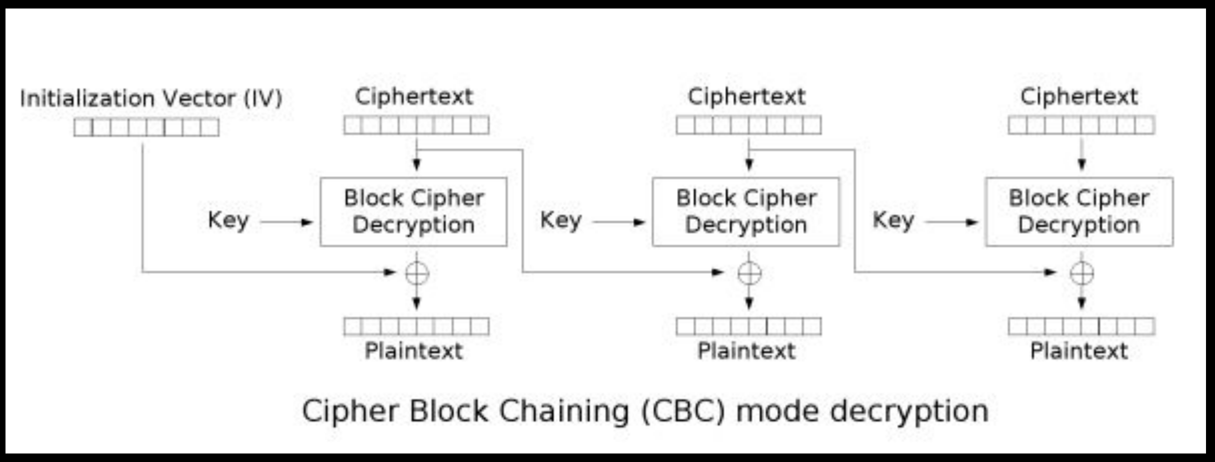


Modes other than ECB result in pseudo-randomness



Using ECB allows patterns to be easily discerned

Cipher Modes



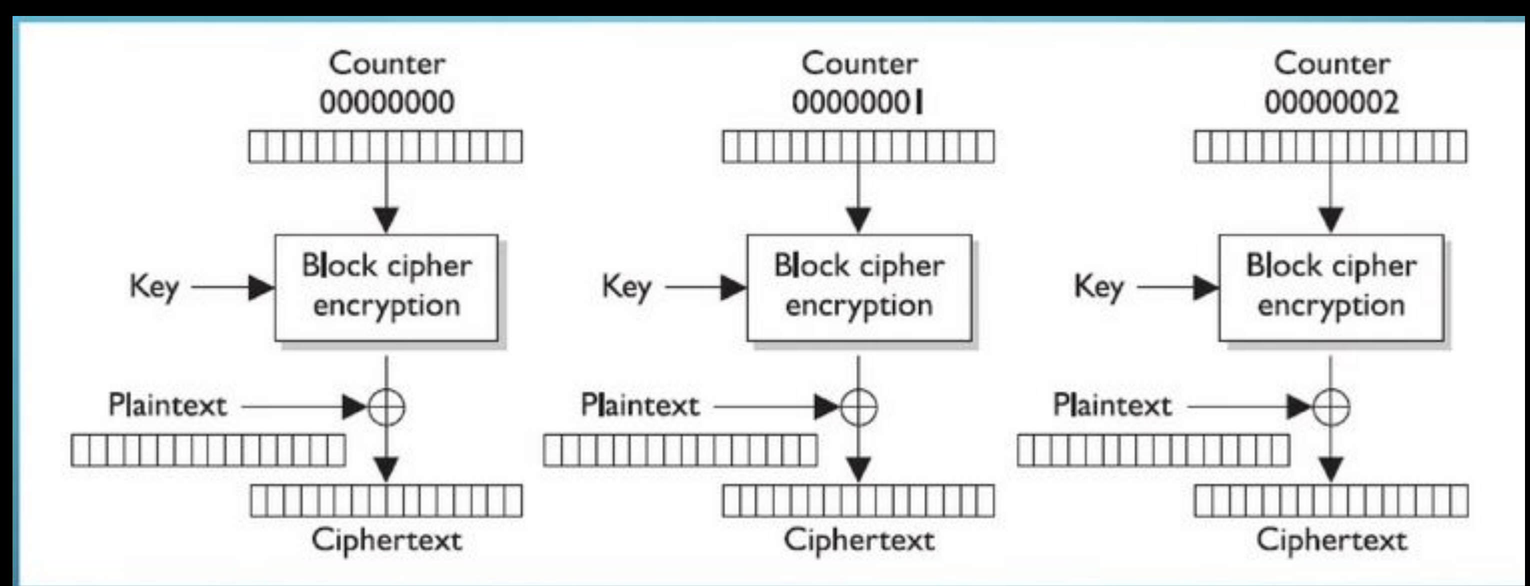
Cipher Modes

CFB(t)/OFB(b)

Plaintext Plaintext
Algorithm Keystream Algorithm Keystream
IV
Ciphertext Ciphertext

Plaintext Plaintext
Algorithm Keystream Algorithm Keystream
IV
Ciphertext Ciphertext

Cipher Modes - CTR



How to tell what mode's been used

- Check for repeated bytes in the encryption, and if the ciphertext is of a multiple length of 16 bytes. This likely means that ECB was used
- If there is no sign of any repeated bytes but the ciphertext is still of a multiple length of 16 bytes, then either CBC or ECB encryption has likely been used
- If you have a black box encryption oracle available, try sending 1 byte to the oracle. If you get back 1 byte, then this has likely been one of the stream modes (OFB/CFB), but if you get 16 bytes, then it's one of the whole-block modes

Possible Attack Vectors: Differential Cryptanalysis

- If the cipher exhibits some sort of non-random behavior based on how the plaintext bits change and if you can trace some equivalent transformation in the ciphertext, then it’s likely that the implementation is suspect to differential attacks

- In the context of AES, “differentials” are essentially the XOR value between two bytes since subtraction and addition are treated the same in GF(2) and GF(2^8)

- The way a differential propagates through the encryption rounds is independent of the round keys themselves

Differential Cryptanalysis

- Consider 2 known plaintext bitblocks, P_1 and P_2, and let the known XOR (diff) between them be ΔP
- Now suppose we now retrieve the corresponding enciphered blocks C_1 and C_2 and have the known diff be ΔC
- Let the final round output $C_1 = C_1' \wedge K$ and $C_2 = C_2' \wedge K$ where C_1' and C_2' are the SBOX outputs and K is the round key

$$\Delta C = C_1 \wedge C_2 = C_1' \wedge K \wedge C_2' \wedge K = C_1' \wedge C_2'$$

- Thus, the final round differential is NOT dependent on the output of the round keys but rather the SBOX

Possible Attack Vectors: Linear Cryptanalysis

- Secure S-Boxes in block ciphers are designed to be resistant towards 2 kinds of cryptanalysis: linear and differential
  - If an S-Box linear, the output bitvector ***y*** of the substitution can be expressed as the bitwise XOR-sum of some linear combination of the input bitvector ***x***
  - Basically, there exist some vector ***b*** and some matrix in GF(2) **A** such that the output bitvector
    - $y = A \cdot x \oplus b$
  - **If this is the case, then we can possibly represent the AES/DES encryption as an affine transformation!!!**

```
P.<x> = PolynomialRing(GF(2))
T.<z> = GF(2^8, modulus=x^8 + x^4 + x^3 + x + 1)
PR = PolynomialRing(T, [f'm{i}' for i in range(16)])
Mgens = PR.gens()
def __shift_rows(self, M):
        s = [list(r) for r in M.rows()]
        s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[3][1], s[2][1], s[0][1]
        s[0][2], s[1][2], s[2][2], s[3][2] = s[1][2], s[0][2], s[3][2], s[2][2]
        s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[0][3], s[3][3], s[2][3]
        return Matrix(s)

def __mix_columns(self, M):
    S = Matrix(T, [
        [T.fetch_int(2), T.fetch_int(3), T.fetch_int(1), T.fetch_int(1)],
        [T.fetch_int(3), T.fetch_int(2), T.fetch_int(3), T.fetch_int(1)],
        [T.fetch_int(1), T.fetch_int(1), T.fetch_int(2), T.fetch_int(3)],
        [T.fetch_int(1), T.fetch_int(1), T.fetch_int(1), T.fetch_int(2)],
    ])
    return S*M
```

Linearity of AES without SubBytes and AddRoundKey

These two operations are completely linear, and thus it is possible to represent the encryption as a matrix transform with the ciphertext $\boldsymbol{y} = \boldsymbol{Ax}$

In this toy implementation, **x** can be represented as a single vector of elements m_0 to m_{15}, each of which are elements of $GF(2^8)$

When we call ShiftRows and MixColumns, for every single element of **x**, we apply some sort of transformation on each bit as a polynomial in $GF(2^8)$ acting on each of the message elements.

The AddRoundKey scheduling/expanding operations on the key

and the substitution rounds from the linear s-box correspond to an additional Affine component

Check for linearity of an SBOX

- It's very easy to check for the linearity of an sbox
- For all possible *i,j* within 0 to 255,

  if ***S[i^j^0] = S[i] ^ S[j] ^ S[0]***, then the sbox is linear, and it is possible to bring the entirety of AES into the form

  $A \cdot x \oplus b$

General Idea for extracting A and b

- If you have an sbox that you have proven to be linear, the entirety of the AES cipher (with other things constant) is basically an affine transformation over GF(2)
- Simulate the encryption using a symbolic representation: sagemath, github, and past CTF chals are your friends!!!

Possible Attack Vectors: Padding Oracle Attack

- If a plaintext has been encrypted in AES-CBC Mode, then you can implement a kind of side-channel attack to send modified ciphertexts that have been intentionally tampered with
- Suppose we have an oracle available to us that can provide us insight into whether or not a padding scheme input is valid or not
- If we are able to modify an initialization vector, the oracle can return to us whether or not the given IV was “accepted” based on if the ciphertext padding was valid

PO Attack

So we basically have [0x3C] ^ [x] = 0x01 which has been returned as valid from the server.

By the definition of XOR, this must mean that [x] = 0x01 ^ 0x3C = 0x3D

From there, we just need to xor this byte with the corresponding byte from the CT to get the plaintext byte!

Possible Attack Vectors: Padding Oracle Attack

Of course, we have tools that can automate this process

Tools:

Bletchley: https://code.blindspotsecurity.com/trac/bletchley

PadBuster: https://github.com/GDSSecurity/PadBuster

POET: http://netifera.com/research/

Python-Paddingoracle:
https://github.com/mwielgoszewski/python-paddingoracle

Resources for block cipher chals

- Guess the research paper!
- https://crypto.stackexchange.com/
- SageMath
- further explanation of AES symbolically: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf

sigpwny{sauce_box}