SIGPwny

SP2025 Week 03 • 2025-02-13

Block and Stream Ciphers

Sagnik and George

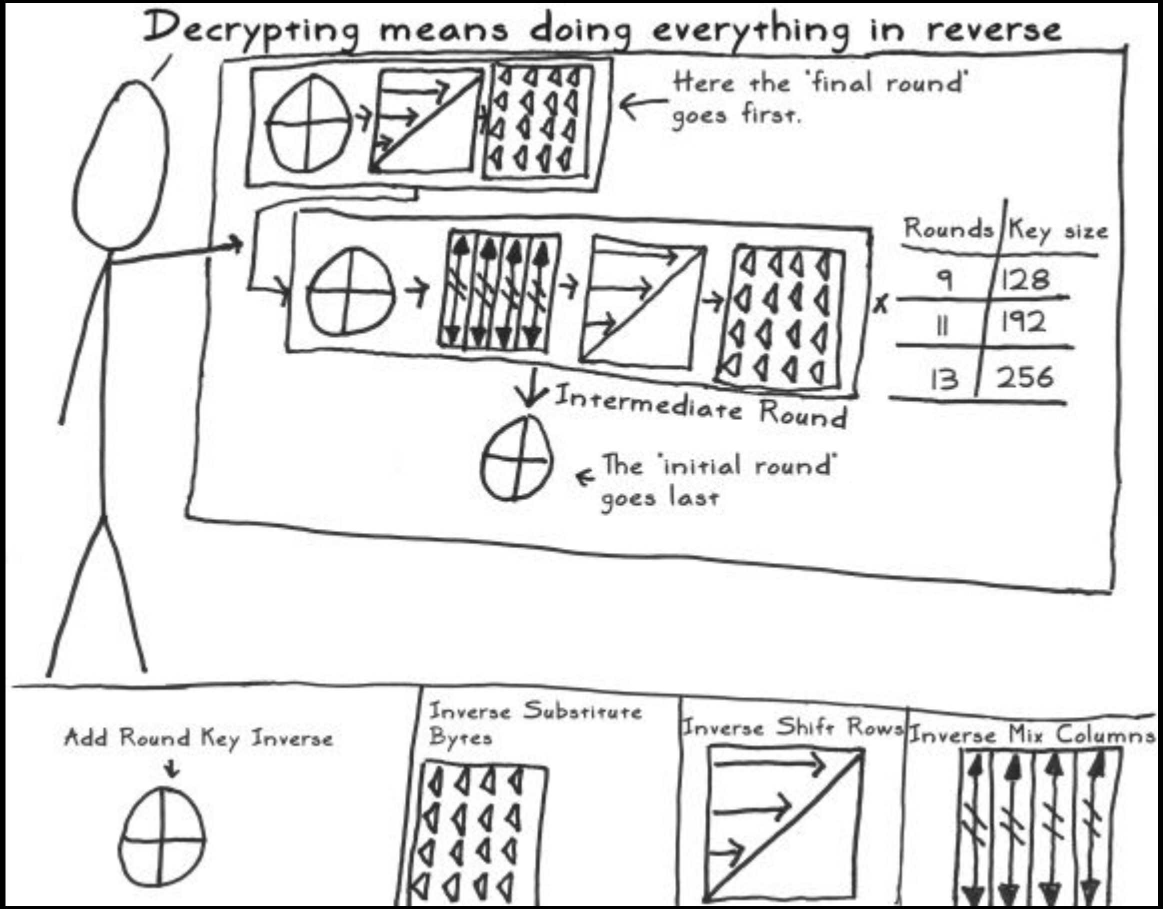
Announcements

2025-02-16 • **This Sunday**

- SIGPwny x SIGPolicy: Cyberwarfare
- Learn about how cyberwarfare affects governments, enterprises and consumers

ctf.sigpwny.com

sigpwny{5ub_5h1ft_4dd_r3p34t}



Pseudorandomness

- We say a sequence of symbols is **pseudorandom** if it seems to look completely random yet has been created by a deterministic, completely repeatable process

- It’s actually provably impossible to turn a short random string into a long random string: **pseudorandom generators** are used to turn a shorter random string into a long string that looks random

Pseudorandomness

- A **PRG** $G : \{0,1\}^n \rightarrow \{0,1\}^{n+s}$ is a mapping such that it is very hard for any polynomial-time “guesser” to guess the output of the PRG given the input string
- So such guessers can’t tell it apart from the output of a truly random function
- Formally, we say for all PPT adversaries A:

$$|P[A(G(k) = 1] - P[A(f) = 1]| \leqslant negl(n)$$

for some truly random function f, uniform random $k : \{0, 1\}^n$, and a negligible function $negl$

So do we security?

- Because a PRG is not *truly* random and is deterministic, it cannot be actually secure
- For this reason, we introduce *probabilistic* encryption:
    - The idea is that **encrypting the same plaintext multiple times gives a different ciphertext**
- $Enc_k(r, m) \rightarrow r, c$ where some random *r* is chosen differently every time Enc is invoked; receiver who gets *r* can then decrypt

Stream Cipher

- A probabilistic encryption algorithm building on top of PRGs where the cryptographic key and algorithm are applied to each binary digit in an input (treated as a data stream)
- The key supplied as input into the PRG is known as the **keystream**
- General Example:
  - Calculate keystream with some random IV: $G(iv, k)$
  - Encrypt message (byte or bit level) $m \in \{0,1\}^{n+s}$: $c = (iv, G(iv, k) \oplus m)$
  - Decrypt with $m = G(k, iv) \oplus c$, discarding IV

Stream Cipher

- Idea for the Stream Cipher: make it difficult for cryptanalysis while still maintaining power efficiency
- Longer, pseudorandomly generated keystream makes it resistant to brute force
- Since we call the algorithm on a smaller space of input (byte level over block level), it requires fewer lines of code and less power expenditure than block ciphers

Examples

- ChaCha20 : very popular used, low power stream cipher
- Rivest RC4: example of an insecure stream cipher, especially when you don’t discard beginning of keystream
- Chameleon, Fish, Helix
- many more

RC4

key schedule:

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylen]) mod 256
    swap(S[i],S[j])
endfor
```

PRG:

```
begin prg(with byte S[256])
    i := 0
    j := 0
    while GeneratingOutput:
        i := (i + 1) mod 256
        j := (j + S[i]) mod 256
        swap(S[i],S[j])
        output S[(S[i] + S[j]) mod 256]
    endwhile
end
```

what might go wrong here?

the key schedule is insufficient, as the first bytes of output reveal info about the original key

Stream Cipher weaknesses

- Two-time pad: if the keystream is used more than once
  - Say we have messages A,B of equal length and encrypt with same key K, then stream cipher produces keystream K with

    $$E(A) = A \oplus K,\ E(B) = B \oplus K$$

  - then if adversary knows $E(A),\ E(B),$ they compute

    $$E(A) \oplus E(B) = (A \oplus K) \oplus (B \oplus K) = A \oplus B$$

  - Having many such plaintexts may allow us to employ standard attacks i.e. crib dragging

- Bit flip attacks (more on that later)

- Chosen IV attack: if choosing particular values for the IV exposes a non-random pattern in the resulting keystream (via differential cryptanalysis), then the attack can reveal some bits in the keystream and reduce effective key length
  - This might weaken the key and allow for follow-up weak key attacks

Block Ciphers

- Another type of deterministic encryption algorithm building on pseudorandomness that operates on a plaintext of fixed length
- Typically, the plaintext is divided into “blocks” of 16 or 32 bytes instead of treated as a continuous stream of byte data
- 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 bit lengths 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

- 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.

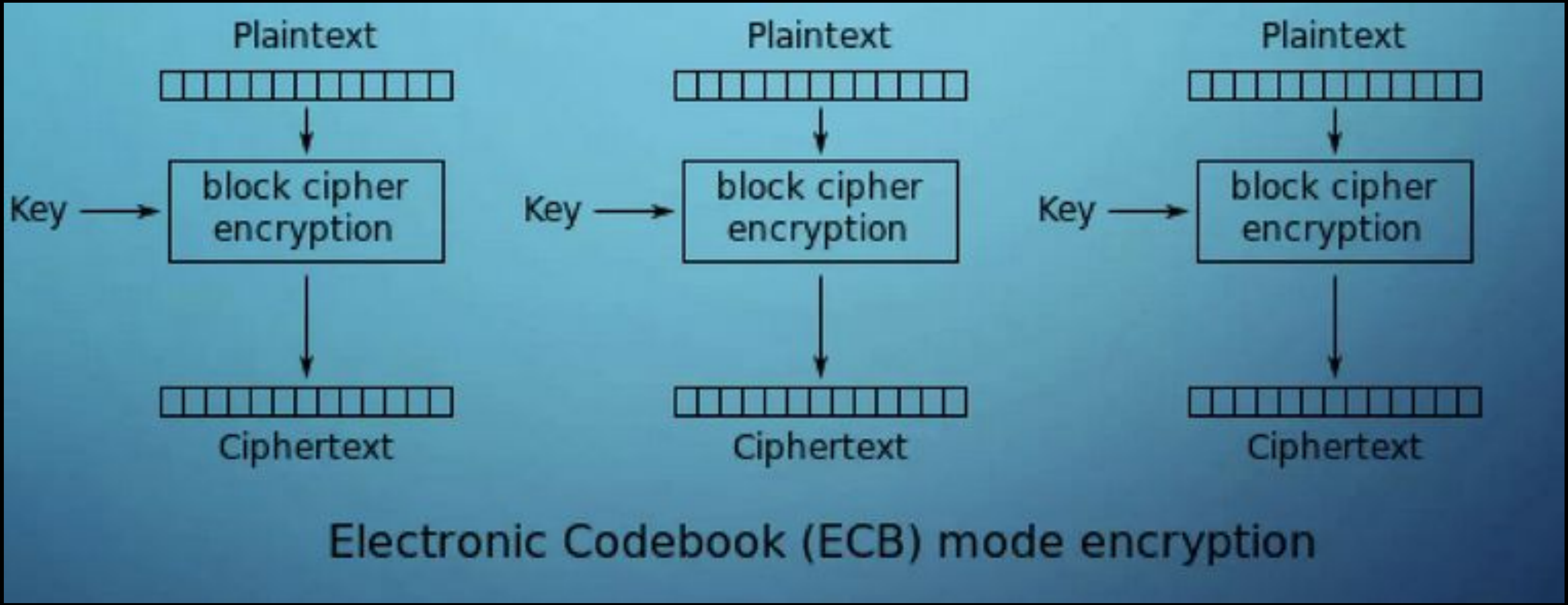
- MixColumns applies a matrix multiplication operation to each column of the current block, providing diffusion in the horizontal direction.

- 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.

Modes

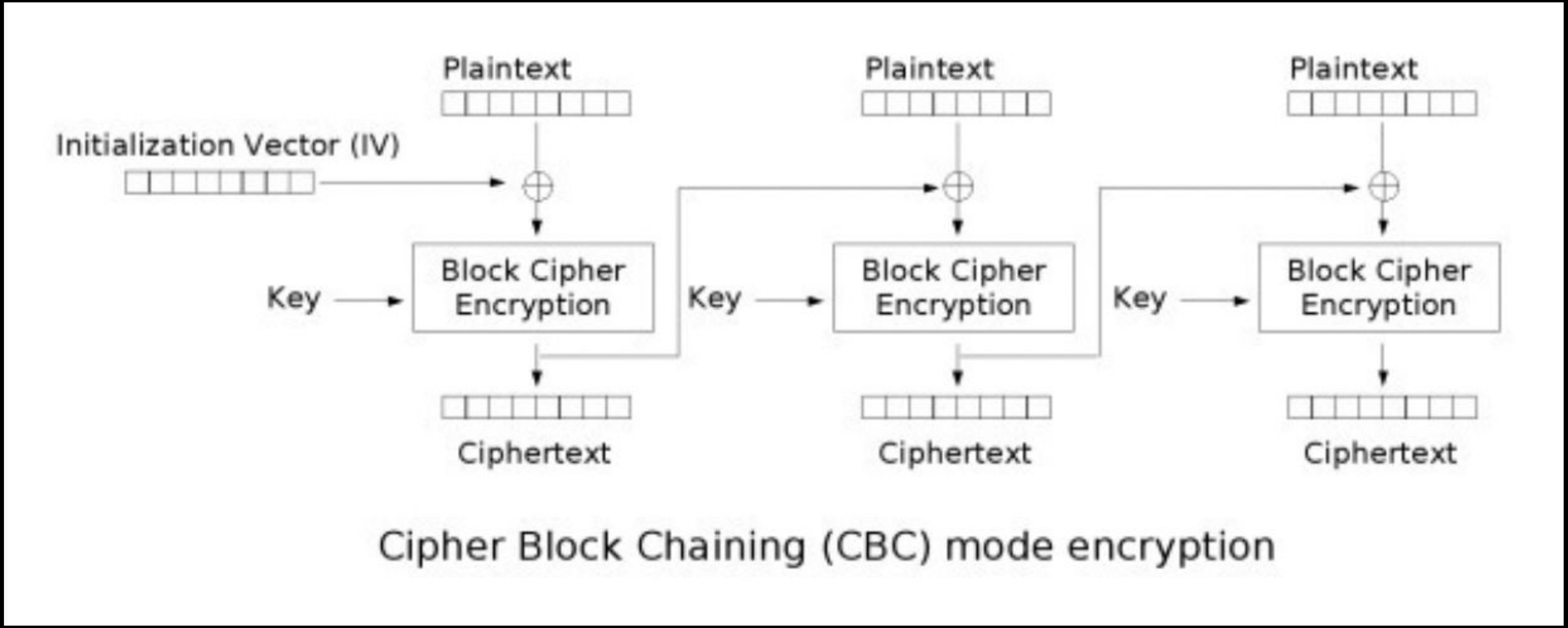
- Block ciphers like AES often have different *modes* of encryption governing how each block is encrypted: the algorithm itself is only good enough to encrypt one block of text, so we need to extend it to work on multiple blocks
- Example modes for AES: ECB, OFB, CTR, CBC, CFB

Cipher Modes - ECB



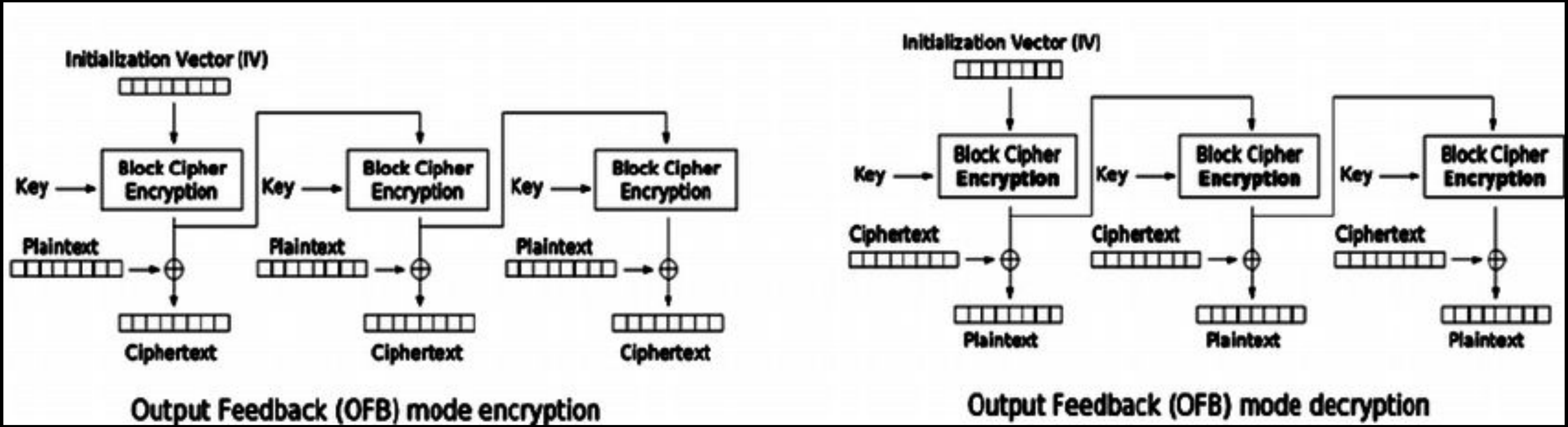
Electronic Codebook (ECB) mode encryption

Cipher Modes



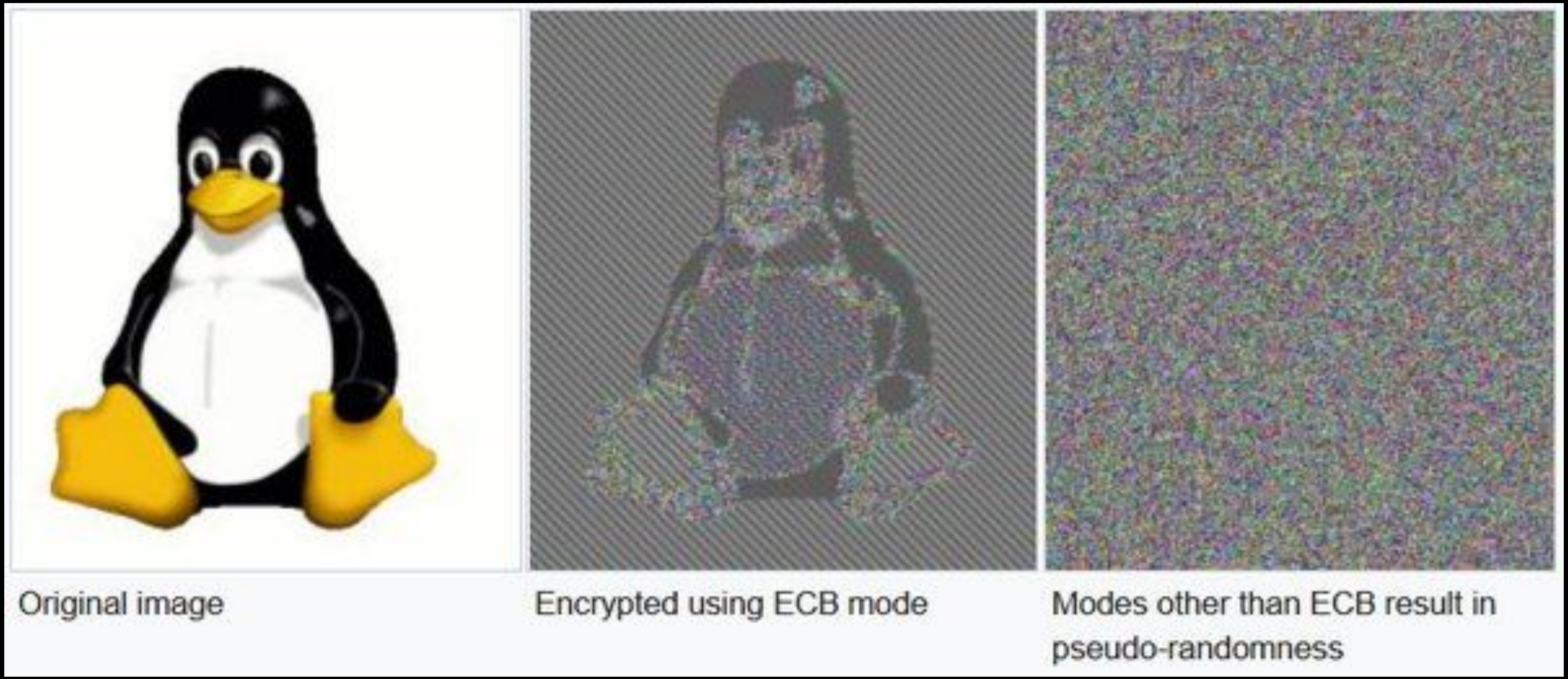
Cipher Modes

CFB/OFB mode (note: creates a keystream from key, IV)


Initialization Vector (IV)
Key
Block Cipher Encryption
Plaintext
Ciphertext
Output Feedback (OFB) mode encryption
Initialization Vector (IV)
Key
Block Cipher Encryption
Ciphertext
Plaintext
Output Feedback (OFB) mode decryption

Other modes include CTR mode (uses a nonce via a counter seeded with IV) and EAX mode (authenticated)

Why we need modes



Original image

Encrypted using ECB mode

Modes other than ECB result in pseudo-randomness

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

Differential Cryptanalysis (ECB)

- 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

Differential Cryptanalysis (ECB)

- 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 $C_1 = C_1' \oplus K$ and $C_2 = C_2' \oplus K$ where C_1' and C_2' are the SBOX outputs and K is the round key

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

Like the ECB penguin, this can reveal patterns within the CT that mirror the PT

Affine Transforms

- transformations upon a vector of the form $\boldsymbol{Ax + b}$
  - In the context of AES, this is represented by $\boldsymbol{y = A \cdot x \oplus b}$
- The transformation first mixes the bits linearly (via matrix A) and then shifts them (via vector b).
- Importantly, if A happens to be reversible, then it is possible to recover x from y: $\boldsymbol{x = A^{-1}(y \oplus b)}$
- Affine transformations are important as they allow efficient mixing
- However, being purely linear has its problems!!!

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 is linear, the output bitvector $\boldsymbol{y}$ of the substitution can be expressed as the bitwise XOR-sum of some linear combination of the input bitvector $\boldsymbol{x}$
    - Basically, there exist some vector $\boldsymbol{b}$ and some matrix in GF(2) **A** such that the output bitvector
        - $\boldsymbol{y} = \boldsymbol{A} \cdot \boldsymbol{x} \oplus \boldsymbol{b}$
    - **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
```

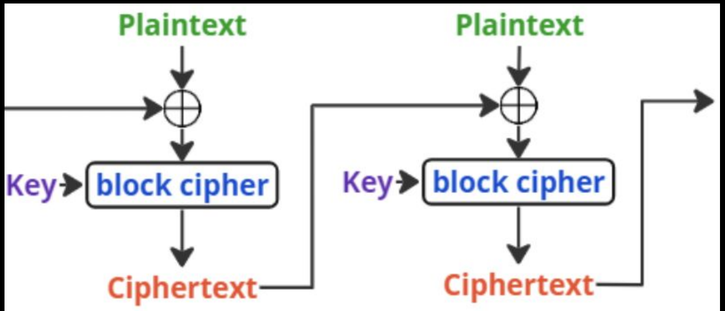
Check for linearity of an SBOX

- 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

  $\boldsymbol{A \cdot x \oplus b}$

Bit flip attack (CBC)

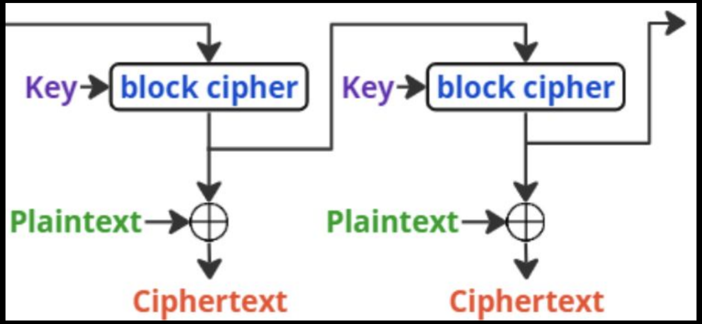


- The decryption formula for AES-CBC would be

$$P_i = D_K(C_i) \oplus C_{i-1}$$
$$C_0 = IV$$

- **Each block of plaintext is XORed with the previous block of ciphertext before being encrypted**
- Thus, if an attacker modifies a bit in the ciphertext of one block, the corresponding bit in the decrypted plaintext of the next block will be flipped
- This can be useful for things like privilege escalation

Bit flip attack (OFB)



- In a similar vein to CBC, a bit flip attack in the OFB mode can flip the bit in the corresponding plaintext

- However, if you notice closely for the CBC algorithm, the current block of changed ciphertext we decrypted returns gibberish, whereas a bit flip in OFB mode will not affect the next block

- This makes OFB more susceptible to bit flip attacks since we might still see the changed plaintext look like valid text

- The alternative CFB mode will flip the bit in the same block like OFB, except it will also clobber the next block, making it easier to authenticate the decrypted message and detect the bit flip attack

Padding Oracle Attack (CBC)

- 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

Padding oracle (CBC)

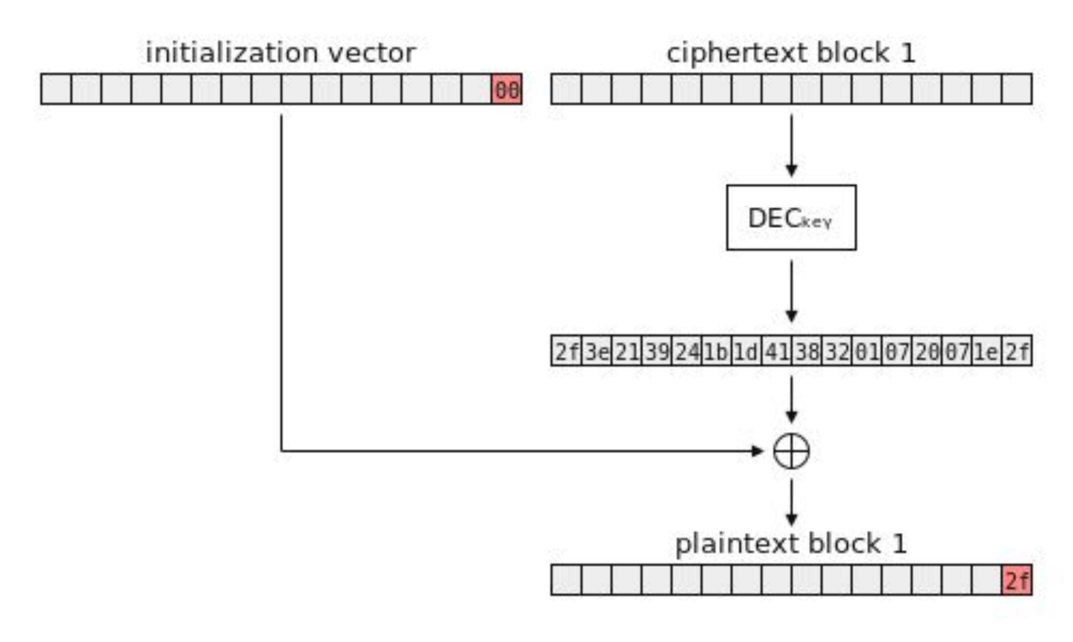
- Suppose you have cipher blocks C_1, C_2 and want to get the 2nd block's decryption P_2 , and PKCS#7 was used for padding
- Flip the last byte of C_1 to make C_1' and sends (IV, C_1', C_2) to the oracle
- The oracle then tells us if the padding of the last block P_2' was valid or not
- If the padding is correct, then we now know that the last byte of $P_2' = D_K(C_2) \oplus C_1'$ is $0x01$
- After finding the last byte of P_2, we can find the 2nd-to-last byte in a similar fashion by setting the last byte of P_2 to $0x02$ by setting the last byte of C_1 to $D_k(C_2) \oplus 0x02$
- Then modify the second-to-last byte until the padding is correct (0x02, 0x02)
- Rinse and repeat until all of P_2 is found

PO Attack (CBC)

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!



Padding Oracle Attack (CBC)

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

General approach

- AES-specific
    - CryptoHack! https://cryptohack.org/challenges/aes/
    - Avinash Kak's compsec lecture notes
- Guess the paper
- https://crypto.stackexchange.com/
- SageMath, PyCryptodome

Next Meetings

`2025-02-16` • **This Sunday**

- SIGPwny x SIGPolicy: Cyberwarfare

`2025-02-20` • **Next Thursday**

- PWN III: ROP
- Learn how to bypass W^X protections with code reuse attacks!

`2025-02-23` • **Next Sunday**

- Seminar meeting (TBD)

ctf.sigpwny.com

sigpwny{5ub_5h1ft_4dd_r3p34t}

Meeting content can be found at sigpwny.com/meetings.

SIGPwny