

# CSCI 476: Computer Security

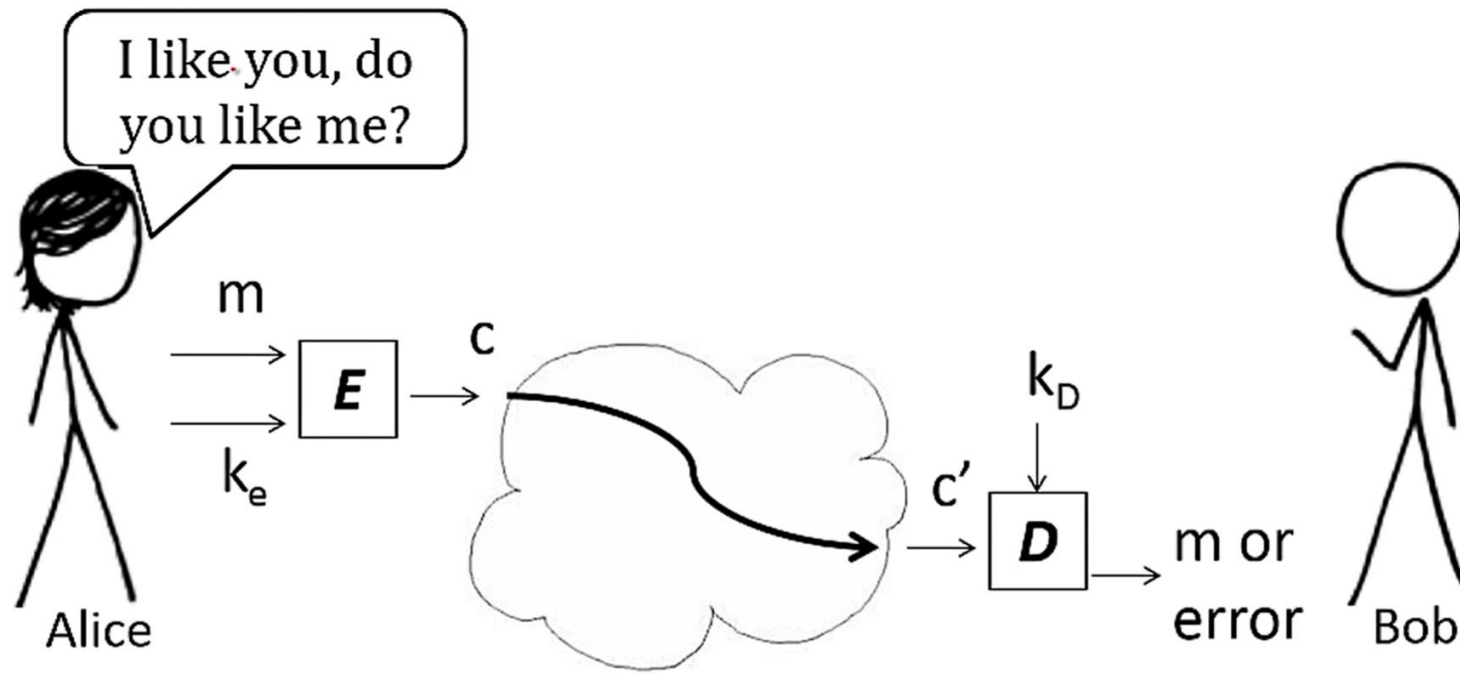
Secret Key Encryption/Symmetric Cryptography (Part 2)

Reese Pearsall  
Spring 2023

# Announcement

Lab 8 due date changed

4/16 → 4/19



## Cryptosystem

$m$ : Plaintext

$k_e$ : Encryption Key

$k_d$ : Decryption Key

$c$ : Ciphertext

$E$ : Encryption Program

$D$ : Decryption Program

Deterministic programs\*

The importance here is that the **keys** used for encryption/decryption are secret (ie not public knowledge)

The innerworkings of the encryption/decryption program *is* public knowledge though

# Block Cipher

Split in messages into fixed sized blocks, encrypt each block separately

Hello there world

01101000	01100101	01101100
01101100	01101111	00100000
01110100	01101000	01100101
01110010	01100101	00100000
01110111	01101111	01110010
01101100	01100100	00001010

Block 1

Block 2

Block 3

$\oplus$

$\oplus$

$\oplus$

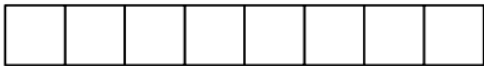


The specifics of this operation vary depending on your mode of encryption

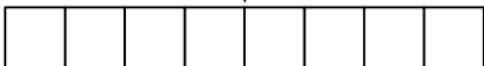
Key

*n bits*

Plaintext



Block Cipher  
Encryption



Ciphertext

*n bits*

**Decryption** is performed by applying the reverse transformation to ciphertext blocks

Important  
Properties

- Even small differences in plaintext result in different ciphertexts
- Blocks in plaintext that are the same will also have matching ciphertexts

# Block Ciphers

**AES** (Advanced Encryption Standard) and **DES** (Data Encryption Standard) are both symmetric block ciphers. The way they do block encryptions is slightly different

In AES: Key lengths can be 128, 192, or 256 bits. IN DES, key length can only be 56

Under the hood, these are rather complex ciphers, but each cipher involves multiple rounds of “encryption”

DES is older, broken and has known vulnerabilities, AES is the current widely-used block cipher

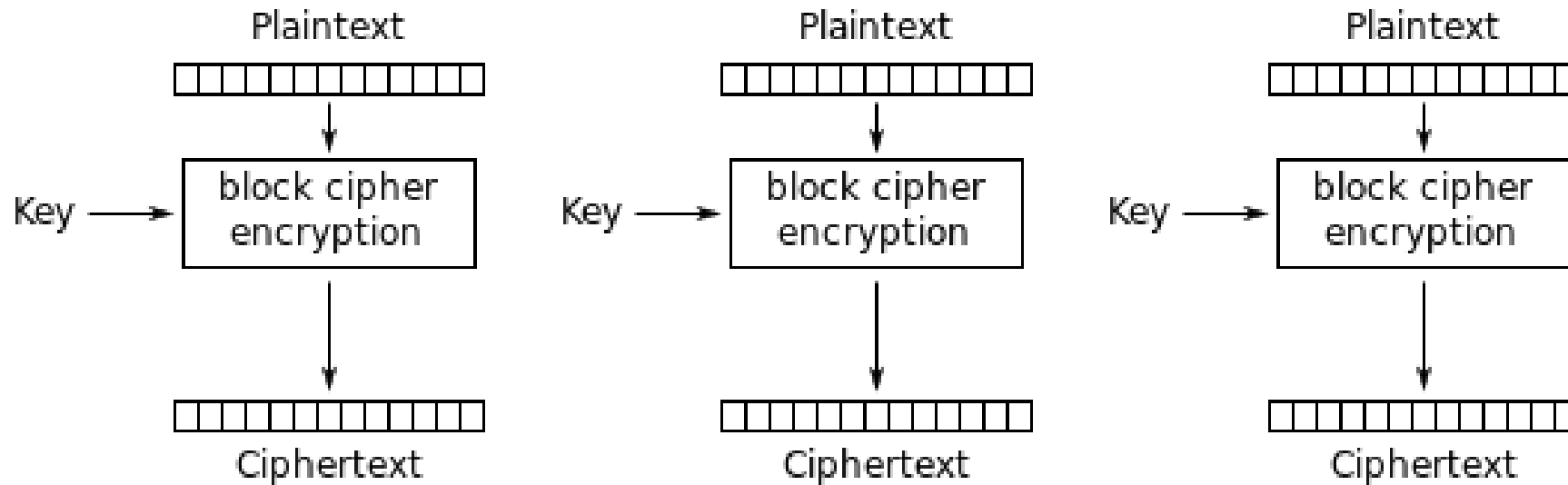
# Modes of Encryption

- Electronic Codebook (ECB)
- Cipher Block Chaining (CBC)
- Propagating CBC (PCBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

**All block ciphers!**

*But if we aren't careful about how we conduct encryption operations, we may accidentally reveal information about the plaintext*

# Electronic Codebook **ECB**



Electronic Codebook (ECB) mode encryption

**Notice:** For the same key, a plaintext always maps to the same ciphertext

# Using OpenSSL to encrypt w/ ECB

*Encrypt a .txt file*

```
openssl enc -aes-128-ecb -e -in plain.txt -out cipher.txt \
-K 00112233445566778899AABBCCDDEEFF
```

- ① Encrypt using AES (block cipher) with mode ECB using a 128-bit key
- ② **Encrypt**
- ③ Input file to be encrypted will be *plain.txt*
- ④ Output file created that contains the ciphertext will be *cipher.txt*
- ⑤ Key used for encryption will be 00112233445566778899AABBCCDDEEFF 32 characters in hex → 128 bits




# Using OpenSSL to encrypt w/ ECB

*Encrypt a .txt file*

```
openssl enc -aes-128-ecb -e -in plain.txt -out cipher.txt \
-K 00112233445566778899AABBCCDDEEFF
```

*plain.txt*

1 The FitnessGram Pacer Test is a multistage aerobic capacity test that progressively gets more difficult as it continues. The 20 meter pacer test will begin in 30 seconds. Line up at the start. The running speed starts slowly, but gets faster each minute after you hear this signal. [beep] A single lap should be completed each time you hear this sound. [ding] Remember to run in a straight line, and run as long as possible. The second time you fail to complete a lap before the sound, your test is over. The test will begin on the word start. On your mark, get ready, start.]



```
[11/09/22] seed@VM:~$ cat cipher.txt
0IeP0%0:00-=600
00=0090z050;NQ0000K0'0po0L?0\2tZ10NQ0i0K000'00mvsJ060L00000*p006n0
0000t0i0Zq000v0p00]00f"0000D0
0000[/0fp0,00p0hyç[000k>
000000|000>000g)k.0{0+V0;000d00000i
*z%VA;0000lf0v0?00u0$00Z%00T0GfZse
^
0000?C0!00c0JśK0i0Qb00 !C000U0u000>@000)9gm
;00p.~0f0^Ė0?0.0r^00"0000000[000z0;
[0![0 000000aç0_0000E&Di
60yN0?oc00w#0~0000w00?0)+80i03C5:0q00 p800000~/S0Q0[0~5'0+Y0uc0C00
04000aq1Y0000I0000uk00s0000%j070/FP00,x0>0i0X0^0T00zg00C00G000FR,
000fP@|0009h,0{H0g%600@e~0@eZDx'Gp]B/0[11/09/22] seed@VM:~$ █
```

# Using OpenSSL to encrypt w/ ECB

*Encrypt a .txt file*

```
openssl enc -aes-128-ecb -e -in plain.txt -out cipher.txt \
-K 00112233445566778899AABBCCDDEEFF
```

*Decrypt a .txt file*

```
openssl enc -aes-128-ecb -d -in cipher.txt -out new_output.txt \
-K 00112233445566778899AABBCCDDEEFF
```

```
[11/09/22]seed@VM:~$ cat cipher.txt
0IeP0%0:00-=600
00=0090z050;N00000K0'0po0L?0\2tZ10NQ0i0K000'00mvsJ060L00000*p006n0
0000t0i0Zq000v0p00]00f"0000D0
0000[/0fp0,00p0hyr[000k>
0000?C0!00c0J5K0i0Qb00 !C000U0u000>@000)9gm
;00p.-0f0^E0?0.0r^00"0000000[000z0;
[0![0 000000az0_0000E&Di
60yN0?oc00w#0~0000w00?0)+80i03C5:0q00 p800000^/S0Q0[0~5'0+Y0uc0C00
04000aq1Y0000I0000uk00s0000%j070/FP00,x0>0:0X0^0T00zg00C00G000FR,
000fP@|0009h,0{H0g%600@e~0@eZDx'Gp]B/0[11/09/22]seed@VM:~$ █
```



```
[11/09/22]seed@VM:~$ cat new_output.txt
The FitnessGram Pacer Test is a multistage aerobic capacity test that progressively gets
more difficult as it continues. The 20 meter pacer test will begin in 30 seconds. Line up
at the start. The running speed starts slowly, but gets faster each minute after you hea
r this signal. [beep] A single lap should be completed each time you hear this sound. [di
ng] Remember to run in a straight line, and run as long as possible. The second time you
fail to complete a lap before the sound, your test is over. The test will begin on the wo
rd start. On your mark, get ready, start.
-----
```

# Using OpenSSL to encrypt w/ ECB

*Encrypt a .txt file*

```
openssl enc -aes-128-ecb -e -in plain.txt -out cipher.txt \
-K 00112233445566778899AABBCCDDEEFF
```

*Decrypt a .txt file*

```
openssl enc -aes-128-ecb -d -in cipher.txt -out new_output.txt \
-K 00112233445566778899AABBCCDDEEFF
```

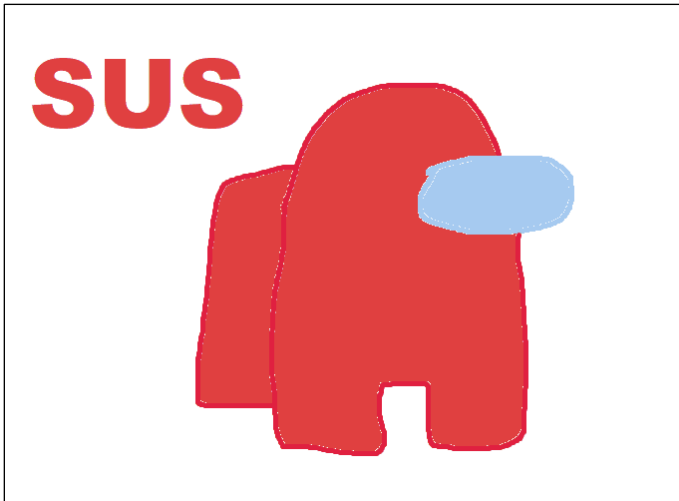
Changing the key used for decryption wont decrypt correctly!

```
[11/09/22]seed@VM:~$ openssl enc -aes-128-ecb -d -in cipher.txt -out new_output.txt -K 00
112233445566778899AABBCCDDEEFF
bad decrypt
140636099929408:error:06065064:digital envelope routines:EVP_DecryptFinal_ex:bad decrypt:
crypto/evp/evp_enc.c:583:
[11/09/22]seed@VM:~$ cat new_output.txt
v.00>X!0@.0~hy4c00A}00000(00tg{0M00q00u(00KU00h0%g0zmH0000(000
g'000]0005n00000kD000'L000a00070Vf0(000K0^200J/3;2Y0q00b000&w00-hQ000zY00R+000C0?00j00000
?0'0o00qj?0~A5J/;F.L/D?V00/00f00m00000M00t00H0Dr.#.0
0000i00s*0000&F/000Bv0w=
d0>00r00030i0000r0z
}d00dA00000]F000030000*:0ZX0/0?h0Y0md02W00w05桧0<0z0000r00|0020|0U0bb0[11/09/22]seed@VM:
~$
```

# Using OpenSSL to encrypt w/ ECB

*We can encrypt many things (everything on computers is just 0s and 1s). Let's try an image!*

sus.bmp



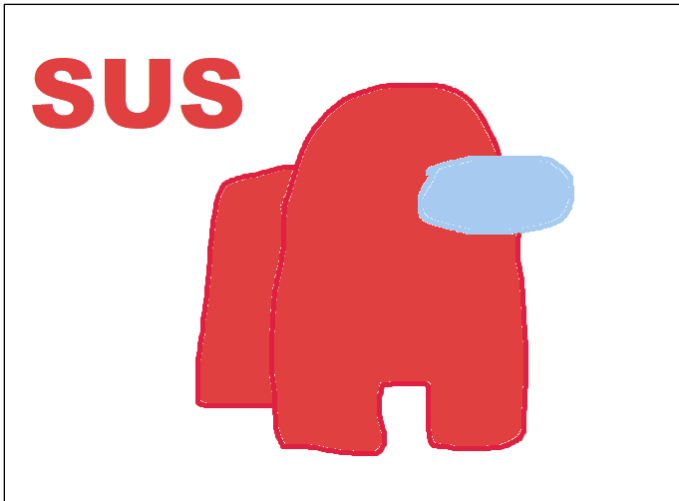
When encrypting images on the lab, make sure you use a **.bmp** image

(You can encrypt jpg and png, but you won't be able to follow the steps on the next few slides)

# Using OpenSSL to encrypt w/ ECB

*We can encrypt many things (everything on computers is just 0s and 1s). Let's try an image!*

sus.bmp



When encrypting images on the lab, make sure you use a **.bmp** image

(You can encrypt jpg and png, but you won't be able to follow the steps on the next few slides)

BMP files (and most files) have **headers**, which tell the OS what file type this sequence of 0s and 1s is

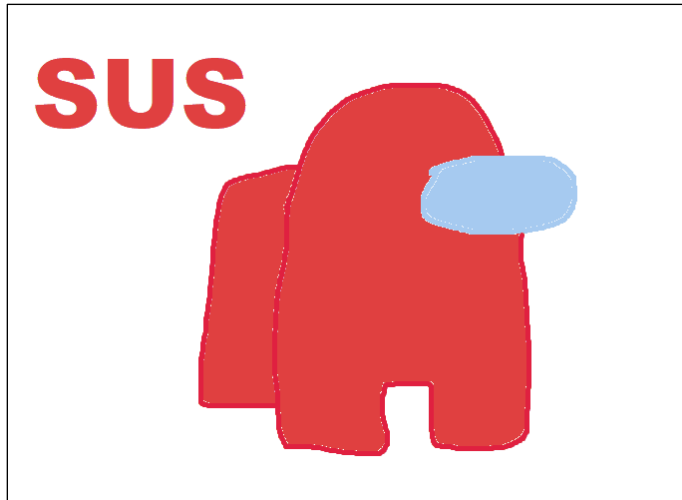
When we encrypt the image, the header will also get encrypted

The OS loads the encrypted image → Can't display it!

# Using OpenSSL to encrypt w/ ECB

*We can encrypt many things (everything on computers is just 0s and 1s). Let's try an image!*

sus.bmp



01011010101010111010110110101010101010101010100101

```

00100100111010111101110010100010011000000101101000011100110101011101110000010
001011110010010101101110110110111111111001100110011001001001000100010
01000001000000001010001101010001010100000101000000101000000111111010000010
111000100000100011011110111000101000000100000011000100010100111110000001001
00000000101011010010101010110011111011110111001000001001000011001101
100010101000100101001010111000000101011101100001100101000100010111101101
1101000000100100100111000010010010001000101111010001101001010000000100011
100111111001001001010010001001011101110111110001100001110110001101101010
110100111111011101100100101001001100100100101000010000001001001001010111
1000001010001001110001010001010001010010111011001110100111010010100000011
01010000001001010010100101111011110110010000100010101000010110000101
101011111111011100001100000100010000010011100110111000001000001011
10010000100011101110111000010010010111100111001001101100100010011100100
110101001111001010000000101111101110010010010000010011011001101010100100
0100010010000101110000010010001110010111101100101010000010010101001000
0010111110100001011011110110111001010100100000100000111011001010000010000
01010101101100110010010010110100100001000001011000101010010010101100101
001011100001111010101110010011101010111100010010101000101000010101010011
1101001001111111011010100101001010010100101110001011100010111011001011111101
0101100000101001111001110000110000111110001000101011100000101011000001010
1011010111010111010111100101111100001011111000001011101011111100111110111
001100000011110001011011101000000101010101011110000110010100010010101
10100001001100000010101010101010101001000001000010001001111111110101000101
0111101001101011110111110000111011101010000101101100100111101000101
11111010011100100101010111001011111100100011110100000110001001100100101
1010011100101111011001001011011010100001110100011111000001011010000101001
1100100010011110000100100010010111111000101110010001110100101101001010101
1000101011011101001010100001000001100010010101011000000110000100001011110
0011000111010000111111100010111100010110100101101010000111001011100101
000100010001010000101000110111110010101010101000011000101010010010110001
11011110100100101011101110111100010100100110100000101010100101110100101
1000011001011010100100001001011100010111100010111100010111100111

```

## Body of the image

# Header

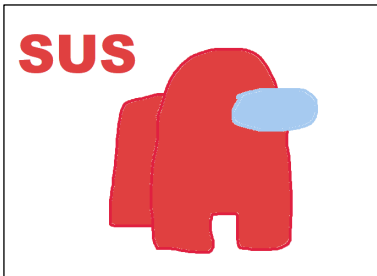
Offset	BMP marker		File size				Reserved				Offset of the pixel data				Header size	
00000000	42	4D	9E	D2	01	00	00	00	00	00	36	00	00	00	28	00
00000010	00	00	C8	00	00	Width	C7	00	00	Height	01	Planes	18	BPP	00	Compre
00000020	00	Size	68	D2	Image size	13	X pix per meter	13	Y pix per meter	00	Colors in					
00000030	Color table	00	Important colors	23	2E	Pixel	26	31	Pixel	28	33	Pixel	27			
00000040	33	6C	27	34	6D	29	34	6E	29	34	6F	29	34	6F	26	33
00000050	71	25	30	6F	25	30	6C	25	30	6B	27	31	6C	2B	35	6D
00000060	2E	37	70	29	35	6F	25	34	6F	21	31	6D	22	32	6B	23
00000070	32	69	26	33	6B	25	33	6D	27	35	6D	26	32	6B	25	31
00000080	6B	26	32	6B	29	35	6D	29	34	6E	25	2F	6B	24	2F	6A
00000090	24	2F	6B	29	33	6D	2D	37	70	27	32	6F	26	32	6B	26

**Fact:** The first 54 bytes of a BMP file will be the header

# Using OpenSSL to encrypt w/ ECB

*We can encrypt many things (everything on computers is just 0s and 1s). Let's try an image!*

sus.bmp



010110101010101110101101101101010101010101010100101

```
100101001101010111011100101100010011100000101101100001110011011010111011100000
0010111100100101011011101110111011111111001101011101100110010100100010010
0100000100000001010001010100010100101000000110010001100000011011110110000010
11100010000011000110111011100010110000010000001100010011001111000001001
00000000101101100101010101100111110111101111001000001011001000111001101
1000101010001001010101011110000001101101110001110110010001011111011010
11101000000100110010011100001100100010100010011110100011101001010000001100011
1001111110011001001001000100101110110111100011100011101100011101101110
110100111111101110011001010010011100100100101000100100000100101000110111
1000001011000100110001010001010010111011001110100111010010100000111
101100000101010100110010101111101111101111001000011001010100010110000101
11010111111111101110001110000100010000100111001110011011100000101000011011
1100100001100111101110110000100100101111001110010011101100100010011100100
11010110011100101100000001101111101110010010100010011011100111010100100
010100100100001101110000011001100011101100101010100010010101011000100
001011111010000110111011110111001100101010010000010000111011001100000110000
0110101011100110011001011001100101000011000010100001010110011001100110
001011100011110110101110010011101010111110001001010100010100001101010011
11010010011111110110101001001101001010100111100010111010001011111011011
01101100000101011001111011000111000111110001000101011100011010100000011010
1011101010111101011110111110001011110010001011110101110111100111
0011000000011110001011101110100000101010101010111000011100101000100110101
101000010011000001010101011011101001000010000100011111111101010001101
01111010011011011110110111100001101110111010100010111011001100111101000110
111110110011100101010111001101111110010001110100000110001010110011000110
1011001110011011110110001010111010101000111011000111110000011011000010001
1100110001001111100010011000100110111110001010111000100111011001110101
100010101110110010101000010000011000100101011100000111000010000101011110
00110001110100001111111000101111000101110101100011101010000111100111001101
00010001100010100001000011011110010101010101000011000101100100101110001
111011101100110010111011110011100010101011100011000101010111010101110011
100000110010111010100010000100101100010111100010111111001111100111
```



enc.bmp

```
10001100100101111000011101100001100001110001111011010000011000100111010001101
0011001010000001010100000101100001101000011111001011110101111010110000110110111
1000011111111010000010100000101101010000010110001110101010001001101001110001000
010001000110110011101100111001101010011100111010010011101110000001010001100001
1010010011100000110000010100011001111010100011010110100111101010100011110010100
1001010001010010010001001000011011110001011011101010111000111100011110010111
100001101000011111001001101010011001110010000110110000000001111010101100111
1101010110011110111000100000001110000000101000100101010001110100110101110100
1111000110010000011110111100010101101011101011010100000100110001010011001011
000101110110100111000100101111000101101011101000100011010001000110100011100000
01110000101101100111000100101111000101101011101000100011010011101100000
011100001011011001100100011110110110011000100111101000010010110101110000
000000010000111110011111011101011100011001101111011010100010001100000111
01000001010010001000110110111011011100000101100010011010111010011011100110
11101011111101000001000001000111011011101110100011010000010111000011101110
1110011000101100000100111111110111000110111101111011001100010100000100101
1001100110011001011101001011101111100101110000101010000000110100010100100
1101110110001101100111100100010110001011111101000100110110010100000011100
00100100101111001001111010101010011111010110001101000101000110110000111000001
11101010011011011001111010000101001000100011100010100000100001110110010000
01001000001010001101111011001011010000010011101110000100010010111001000100
0110111100101001100010110010111000011111011100101001011001110101111100111
0110100001001000100001001001000100000000011110000100101101011000101011
101010100010110101011010000011000001110110110000111010010100011111000110001
0110100110000000010000011100001010010100001011000001111111111001000010111000
1001100000010111000011011011011010100110001011101101110011100011000010010
11000010101001111101100100011101011001111110000011101100001011110111010000
1111111110101010010110010110010101000100001110100011110110000101000001111
1101100100101001110010110011011001100010110001011000010101101011010110101
100011010101110110010101000010000011000100101011100000111000010000101011110
00110001110100001111111100010111100010111010110001111010100001111001011001101
0001000110001010000101000111011110010101010101000011000101100100101110001
111011101100110010111011110001010101110001010101000010101101001011101001101
100000110010111010100010000100101100010111100001010110100101110100101110100110
```

Header AND  
image got  
encrypted

Step 2: Frankenstein together the encrypted image so our OS can open it

```
[11/09/22]seed@VM:~$ head -c 54 sus.bmp > header
[11/09/22]seed@VM:~$ tail -c +55 enc.bmp > body
[11/09/22]seed@VM:~$ cat header body > final.bmp
```

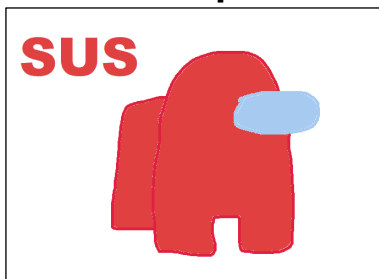
Take the first 54 bytes of the original image (header)  
Take everything after the 54<sup>th</sup> byte of the  
encrypted image (image)



# Using OpenSSL to encrypt w/ ECB

*We can encrypt many things (everything on computers is just 0s and 1s). Let's try an image!*

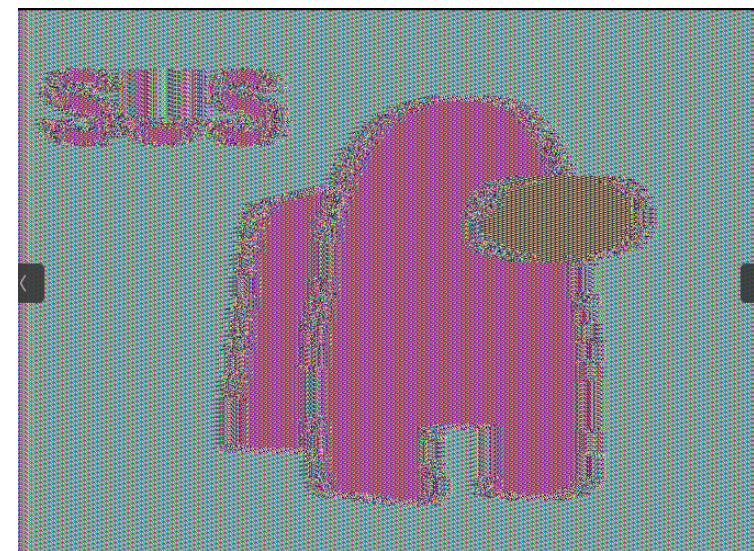
sus.bmp



*final.bmp*

```
01011010101010111010110110110101010101010100101
001100101000000101001000001010000110100011110010111010111010110000110110111
1000011111110100000101000001010101000001011000111010101000100110100110001000
010001000110110011101100111001101010011100111010010011101110000001010001100001
10100100111000001100000101000110011110101000110101101011110101010001111001010
1001010001010010001000101000010111100010101110101010111000111000011100010111
100001101100001111100100110101010011001110010000110110000000000111101010110011
11010101100111101110001000000111000000101000100101010001110100110101110100
111100011001000001111011110001010101011101011010100000100110001010011001011
000101110110100111000100101011110010110101100111101000100011010011101100000
011110000101011001001100100011101101100110001001111010000100101010101110000
000000010000111111001111101011100011001001101110110101001001100000111
0100000101001000100011011010111010101110000001010001001101011101001101100110
111010101111101000001000001000111011011101101000110100000010111000011101110
11100110001011000001001111111101110001101110111110101100010010100000100101
1001100110011001011110100101111011111001011100001010000000010100010100100
11011101100011010011110010001011000101111110100100110011010010100000011100
0010010010111110010011100111110101101011001111101011010001010001010011100010
0001101100010000011001101111100110000010100101000110110000111111111000001
1110101001101101100110111101000010100101000100011100010100000100001110110010000
01001000001010001101111011010010110100000100111011101000010001001111001000100
01101111001010011000101100101110000111110111001010010101011101011110011
01101100001001001000100000100100100000001000111100001001011010101000101011
101010100010110101001101000001100000111011011000001101001010001111000110001
0110100110000000010000011100001010010100001011000001111111110010000101111000
1001100000010101110000110110110110101001100010111011011100111000110000100010
110000101010011111011001000111010101100111110111100111000101011110111010000
11111111110101010001010010110101010001000011010001110101100001010001000001111
110110010010100111001001110100110110001011000011100000101010101010101010
100011101010011111110010001001011000110001110001000100000111100101101000111
00100001100101000011000011111001111001110101100100010101111001011011010111
010001011010111010011110011100000000001101111011100111100111000000000011011
11011100111100111000000000011011110111001111001110000000001101111011100111
110011100000000001101111011
```

[11/09/22] seed@VM:~\$ eog final.bmp



Our encrypted image!!!

Step 2: Frankenstein together the encrypted image so our OS can open it

```
[11/09/22] seed@VM:~$ head -c 54 sus.bmp > header
[11/09/22] seed@VM:~$ tail -c +55 enc.bmp > body
[11/09/22] seed@VM:~$ cat header body > final.bmp
```

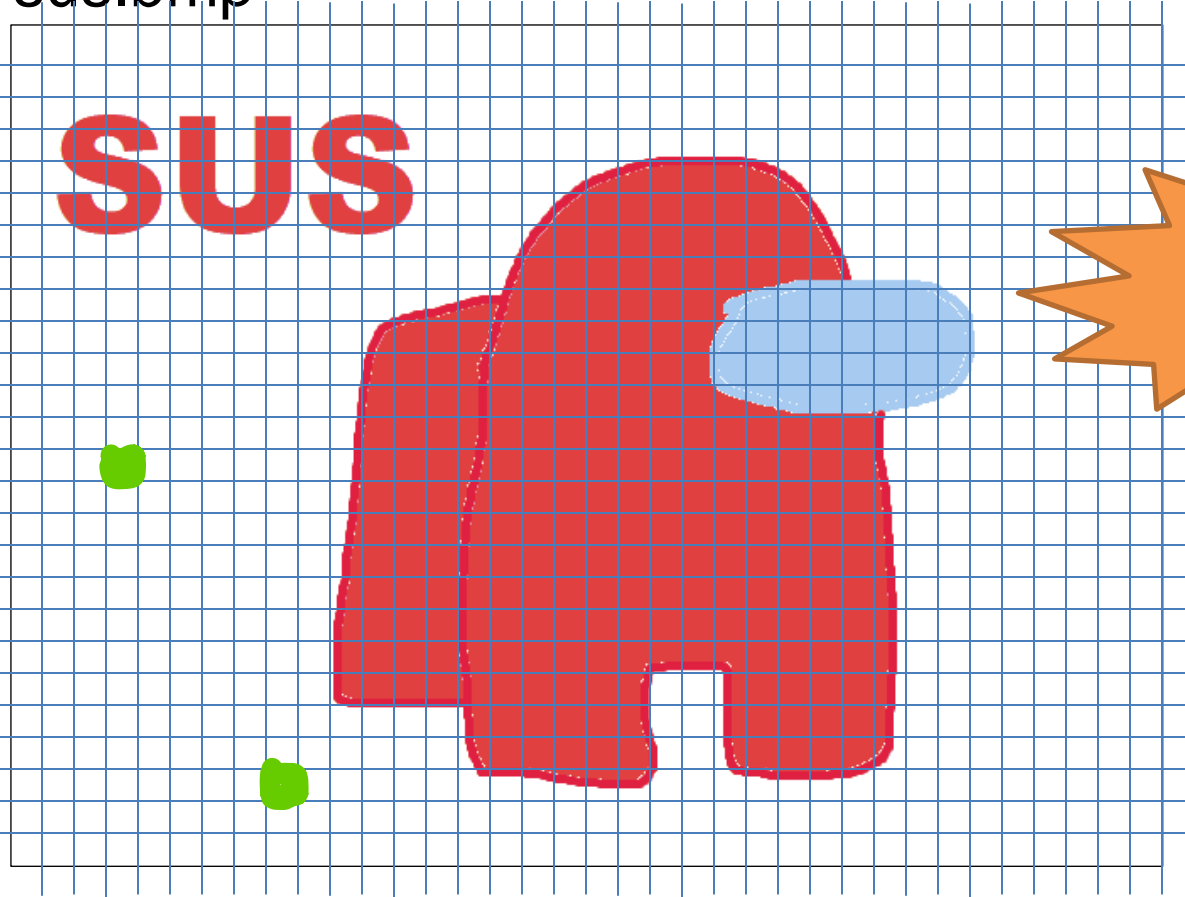
Take the first 54 bytes of the original image (header)  
Take everything after the 54<sup>th</sup> byte of the encrypted image (image)



# Using OpenSSL to encrypt w/ ECB

*Why does this suck?*

sus.bmp



Remember that ECB is a **block cipher** so it will encrypt the image “block by block”

Important Properties

- Even small differences in plaintext result in different ciphertexts
- **Blocks in plaintext that are the same will also have matching ciphertexts**

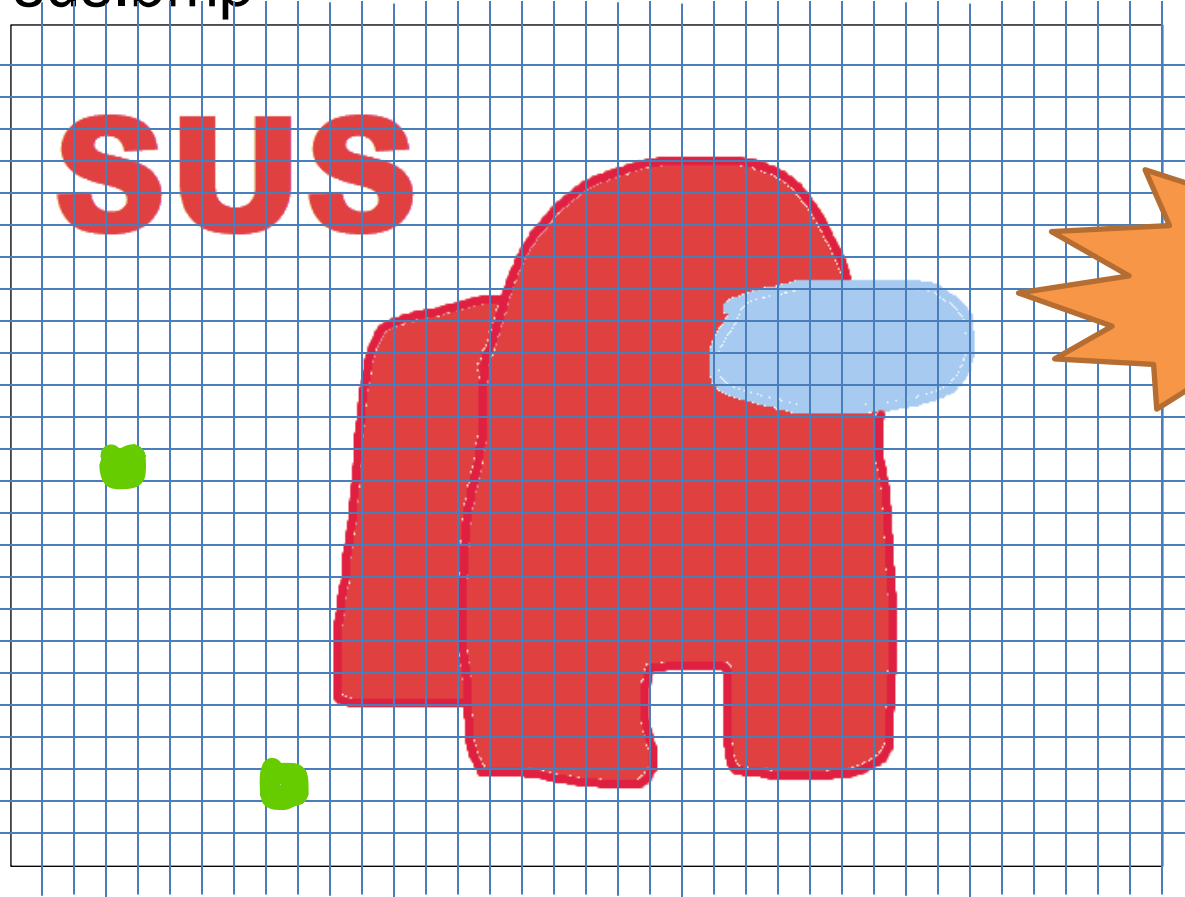
Dividing this image up, we can see that there are many blocks that are the exact same!

# Using OpenSSL to encrypt w/ ECB

*Why does this suck?*

Lesson learned: ECB can reveal information about our plaintext **after** encryption has occurred

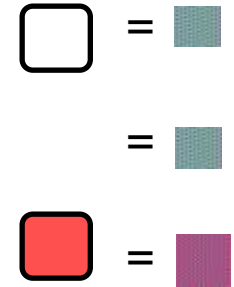
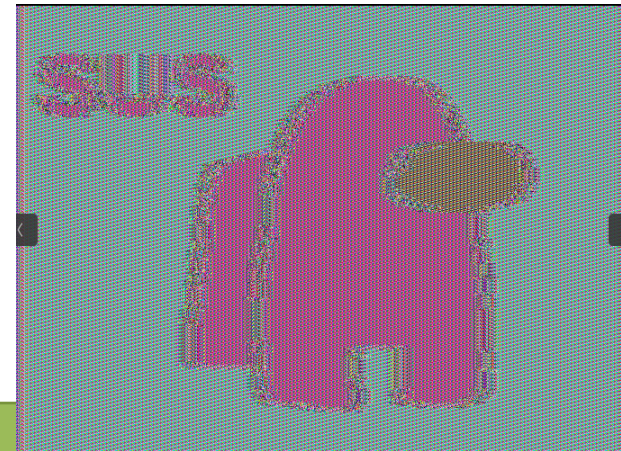
sus.bmp



Important Properties

Remember that ECB is a **block cipher** so it will encrypt the image “block by block”

- Even small differences in plaintext result in different ciphertexts
- **Blocks in plaintext that are the same will also have matching ciphertexts**



# Using OpenSSL to encrypt w/ ECB

Let retry this experiment on a more **complex** image

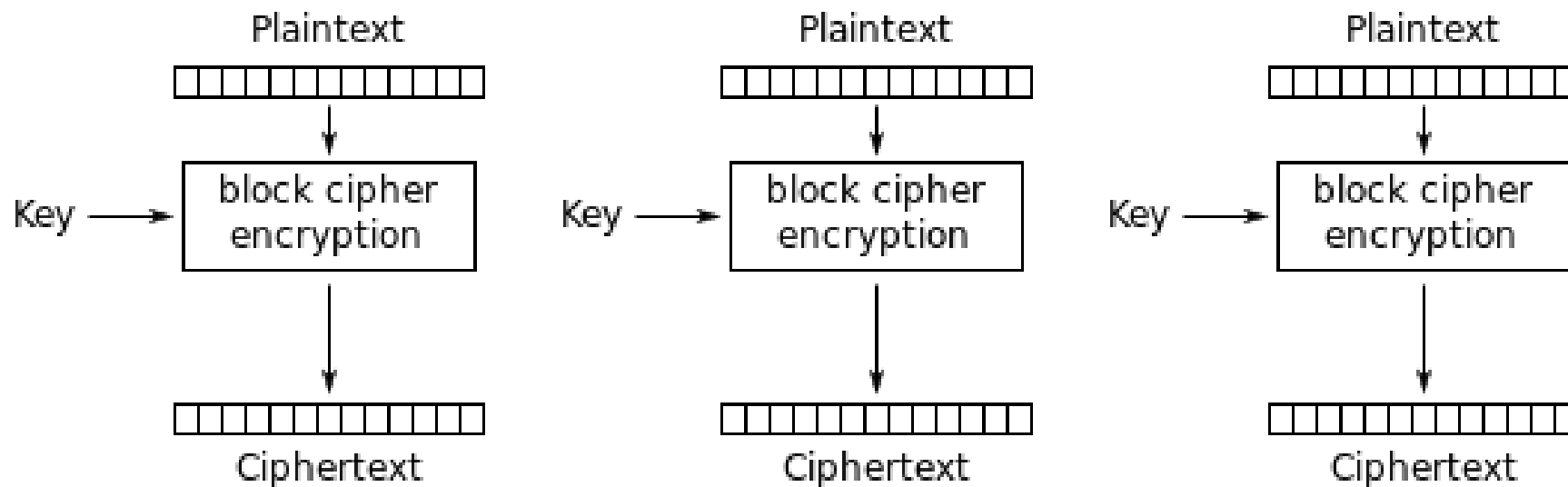
```
[11/09/22]seed@VM:~$ openssl enc -aes-128-ecb -e -in capy.bmp -out enc_capy.bmp -K 00112233445566778899AABBCCDDEEEE
[11/09/22]seed@VM:~$ head -c 54 capy.bmp > header
[11/09/22]seed@VM:~$ tail -c +55 enc_capy.bmp > body
[11/09/22]seed@VM:~$ cat header body > final_capy.bmp
[11/09/22]seed@VM:~$ eog final_capy.bmp
```

capy.bmp



We get much better encryption because the original image uses a lot more colors!

# Using OpenSSL to encrypt w/ ECB



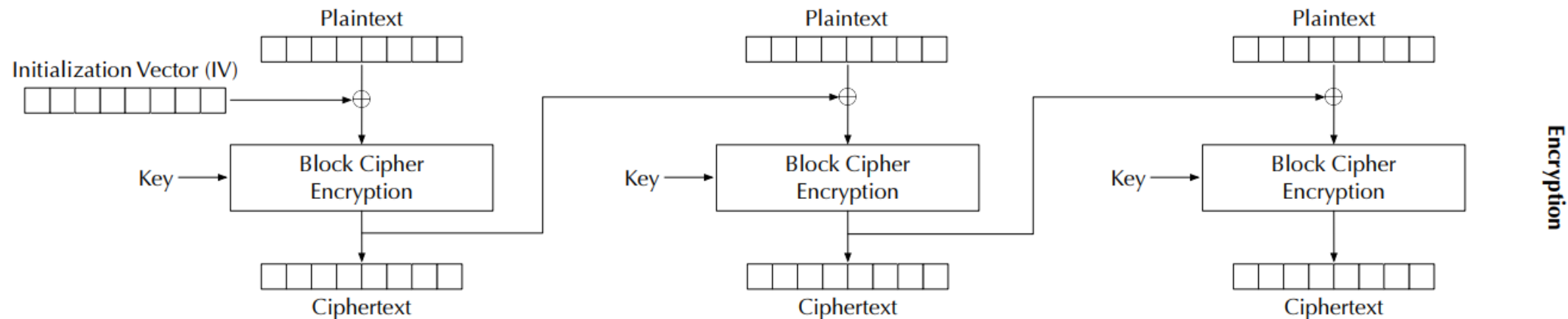
## Electronic Codebook (ECB) mode encryption

### Problem

ECB can reveal information about our plaintext if our blocks are similar!

**Solution:** Add some randomness to each block during encryption

# Cipher Block Chaining (CBC) Mode

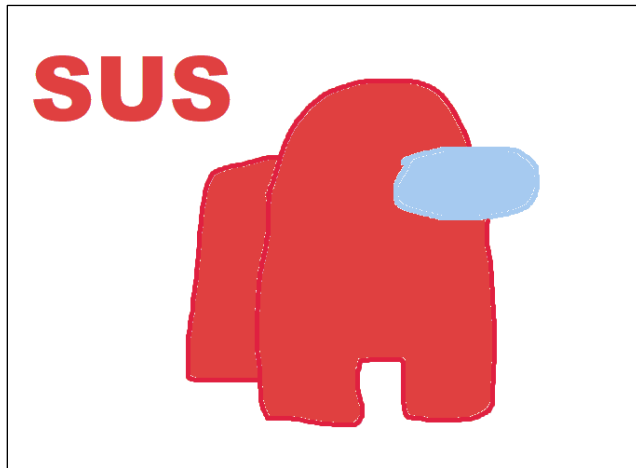


Introduces **block dependency**

$$C_i = E_K(P_i \oplus C_{i-1})$$

Introduces an **initialization vector (IV)** to ensure that even if two plaintexts are identical, their ciphertexts are still different because different IVs will be used

Using CBC to encrypt images??



???

You will do this on the lab.

# Using OpenSSL to encrypt w/ CBC

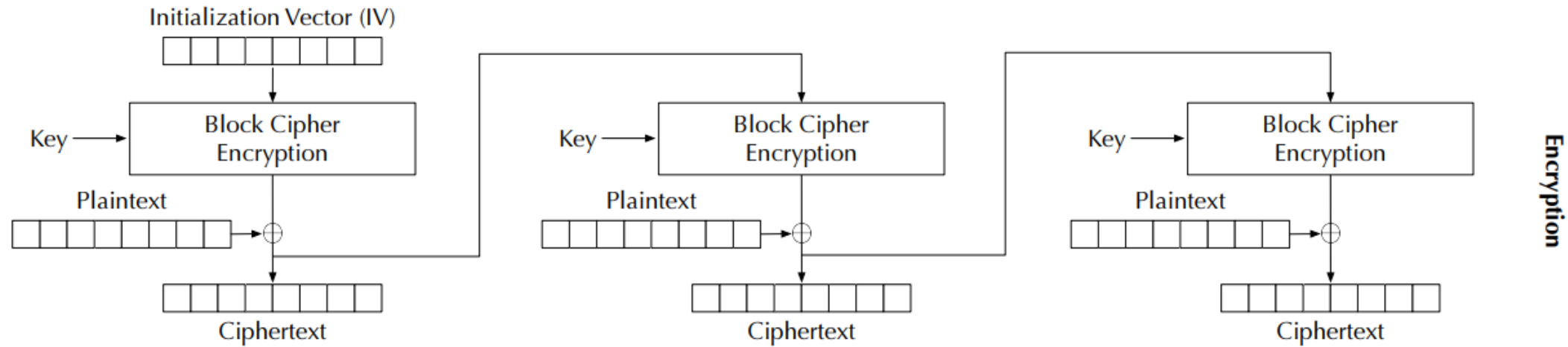
```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0F
```

```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher2.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0E
```

Let's encrypt the same file, but with different IVs



# Cipher Feedback (CFB) Mode



- Similar to CBC, but *slightly different*...  
...a block cipher is turned into a stream cipher!
- Ideal for encrypting real-time data.
- Padding not required for the last block.
- Encryption can only be conducted sequentially — *have to wait for all the plaintext*



# Comparing CBC vs CFB

```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0F
```

```
openssl enc -aes-128-cfb -e -in plain.txt -out cipher2.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0F
```

Any differences in output file sizes?

# Comparing CBC vs CFB

```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0F
```

```
openssl enc -aes-128-cfb -e -in plain.txt -out cipher2.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0F
```

```
[11/10/22]seed@VM:~$ ls -al | grep "cipher"  
-rw-rw-r-- 1 seed seed 576 Nov 10 00:36 cipher2.txt  
-rw-rw-r-- 1 seed seed 592 Nov 10 00:36 cipher.txt
```

Using CFB results in  
a smaller output file!  
(woah!)

# Padding

```
[11/10/22] seed@VM:~$ ls -al | grep "cipher"
-rw-rw-r-- 1 seed seed 576 Nov 10 00:36 cipher2.txt
-rw-rw-r-- 1 seed seed 592 Nov 10 00:36 cipher.txt
```

In a block cipher (where our block sizes is 4), what happens when we don't have a multiple of 4?

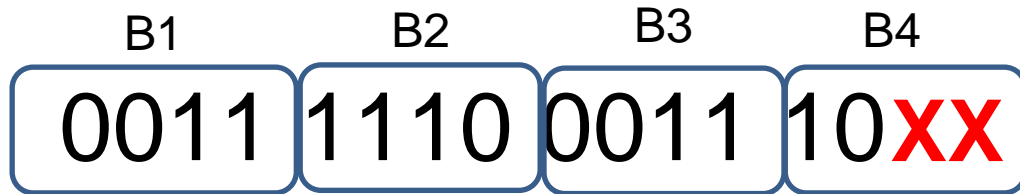


This block is not 4 digits... we need to add more so that our encryption method works!

# Padding

```
[11/10/22] seed@VM:~$ ls -al | grep "cipher"
-rw-rw-r-- 1 seed seed 576 Nov 10 00:36 cipher2.txt
-rw-rw-r-- 1 seed seed 592 Nov 10 00:36 cipher.txt
```

In a block cipher (where our block sizes is 4), what happens when we don't have a multiple of 4?



This block is not 4 digits... we need to add more so that our encryption method works!

Extra data or **padding**, needs to be added to the last block, so its size equals the cipher's block size

# Padding

Questions to answer:

1. *What* does the padding look like?
2. When decrypting, how does the software know *where* the padding starts?

# Padding Experiment #1

What happens when data is smaller than the block size?

```
[11/10/22]seed@VM:~/padding$ echo -n "123456789" > plain.txt  
[11/10/22]seed@VM:~/padding$ ls -ld plain.txt  
-rw-rw-r-- 1 seed seed 9 Nov 10 00:47 plain.txt
```

Plaintext is **9 bytes**

```
[11/10/22]seed@VM:~/padding$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt -K  
00112233445566778899AABBCCDDEEEE -iv 000102030405060708090A0B0C0D0E0F  
[11/10/22]seed@VM:~/padding$ ls -ld cipher.txt  
-rw-rw-r-- 1 seed seed 16 Nov 10 00:53 cipher.txt
```

Ciphertext is **16 bytes** (7 bytes of padding got added on!)

# Padding Experiment #2

## How does decryption software know where the padding starts?

```
openssl enc -aes-128-cbc -d -in cipher.bin -out plain3.txt \
-K 00112233445566778899AABBCCDDEEFF \
-iv 000102030405060708090A0B0C0D0E0F -nopad
```

```
[11/10/22]seed@VM:~/padding$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt -K
00112233445566778899AABBCCDDEEEE -iv 000102030405060708090A0B0C0D0E0F
[11/10/22]seed@VM:~/padding$ openssl enc -aes-128-cbc -d -in cipher.txt -out result.txt -
K 00112233445566778899AABBCCDDEEEE -iv 000102030405060708090A0B0C0D0E0F -nopad
[11/10/22]seed@VM:~/padding$ ls -ld result.txt
-rw-rw-r-- 1 seed seed 16 Nov 10 02:05 result.txt
```

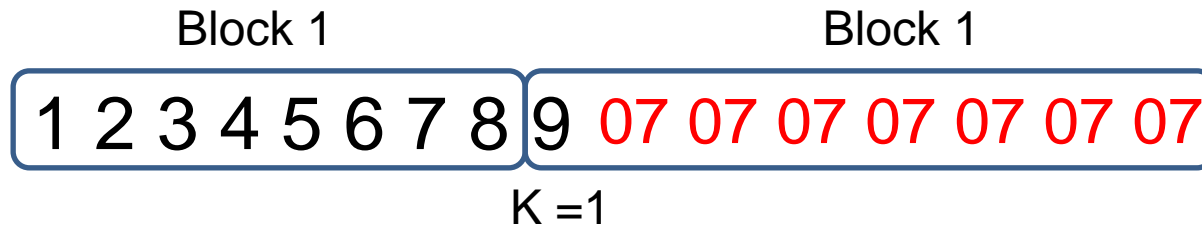
7 bytes of 0x07 are added as padding data

```
[11/10/22]seed@VM:~/padding$ xxd -g 1 plain.txt
00000000: 31 32 33 34 35 36 37 38 39                123456789
[11/10/22]seed@VM:~/padding$ xxd -g 1 result.txt
00000000: 31 32 33 34 35 36 37 38 39 07 07 07 07 07 07 07 123456789.....
```

# Padding Experiment #2

How does decryption software know where the padding starts?

```
[11/10/22]seed@VM:~/padding$ xxd -g 1 plain.txt
00000000: 31 32 33 34 35 36 37 38 39                123456789
[11/10/22]seed@VM:~/padding$ xxd -g 1 result.txt
00000000: 31 32 33 34 35 36 37 38 39 07 07 07 07 07 07 07 123456789.....
```



B = 8 characters

In general, for block size B and last block w K bytes,

B-K bytes of value B-K are added as the padding



# Padding Experiment #3

What if the size of the plaintext is a multiple of the block size? And the last seven bytes are all 0x07?

Block 1

Block 1

1 2 3 4 5 6 7 8 9 07 07 07 07 07 07 07 07

```
$ xxd -g 1 plain3.txt
00000000: 31 32 33 34 35 36 37 38 39 07 07 07 07 07 07 07 07

$ openssl enc -aes-128-cbc -e -in plain3.txt -out cipher3.bin \
-K 00112233445566778899AABBCCDDEEFF \
-iv 000102030405060708090A0B0C0D0E0F

$ openssl enc -aes-128-cbc -d -in cipher3.bin -out plain3_new.txt \
-K 00112233445566778899AABBCCDDEEFF \
-iv 000102030405060708090A0B0C0D0E0F -nopad

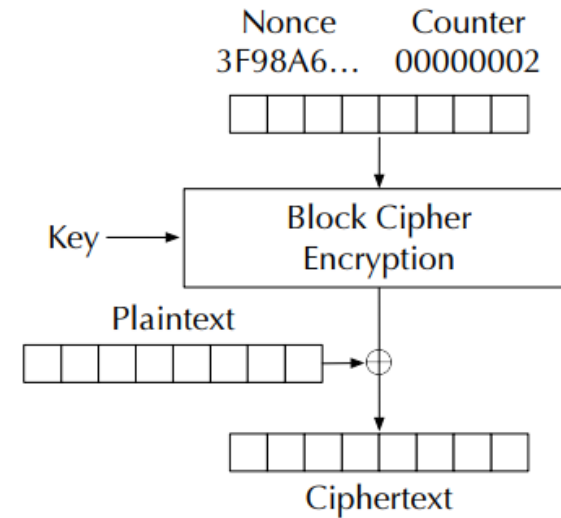
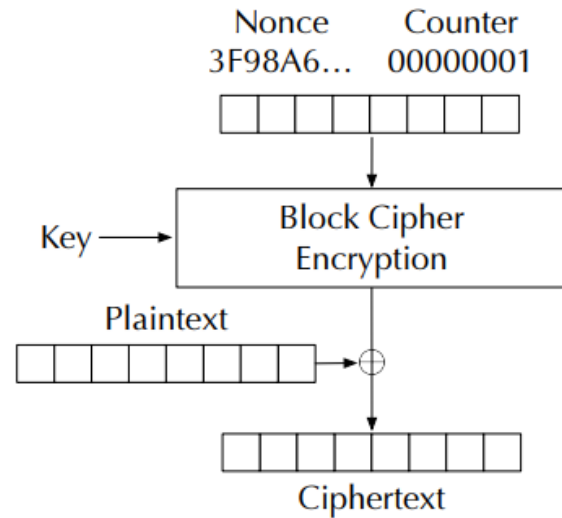
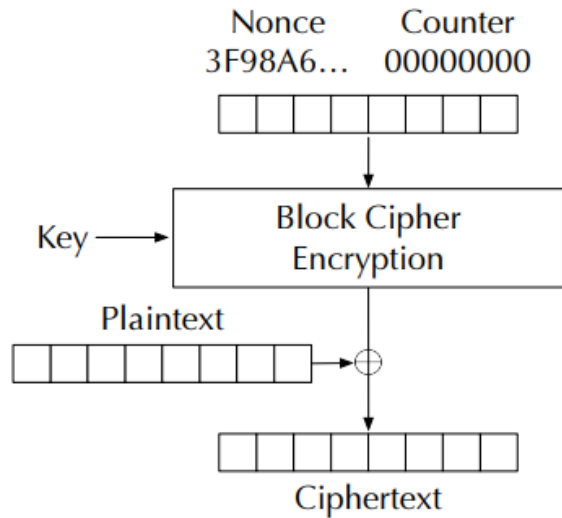
$ ls -ld cipher3.bin plain3_new.txt
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 cipher3.bin
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 plain3_new.txt

$ xxd -g 1 plain3_new.txt
00000000: 31 32 33 34 35 36 37 38 39 07 07 07 07 07 07 07 07
00000010: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

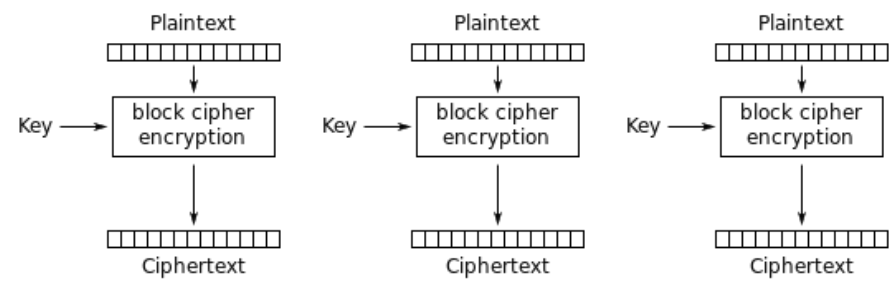
- Size of plaintext (plain3.txt) is **16 bytes**
- Size of decryption output (plain3\_new.txt) is **32 bytes** → a new, full block is added as the padding
- In PKCS#5, if the input length is already an exact multiple of the block size  $B$ , then  $B$  bytes of value  $B$  are added as the padding.

# Counter(CTR) Mode

- Use a counter to generate the key streams
- No key stream can be reused; the counter value for each block is prepended with a randomly generated value called a **nonce** (same idea as the IV)

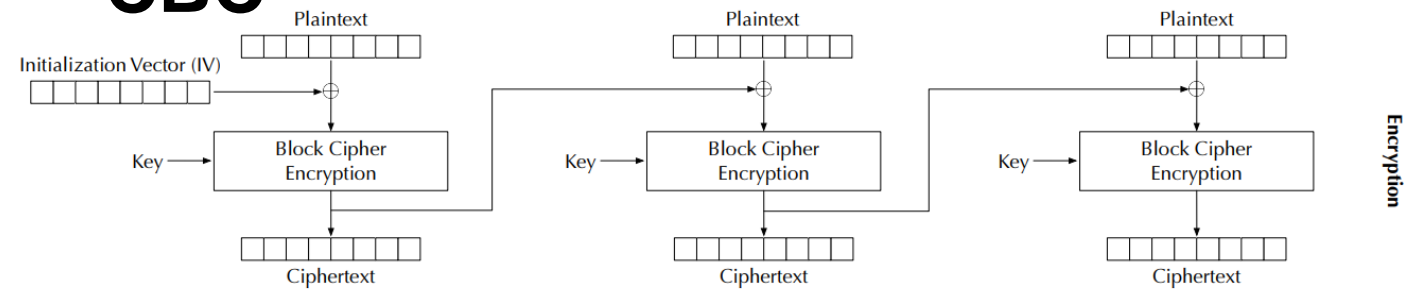


# Modes of Encryption

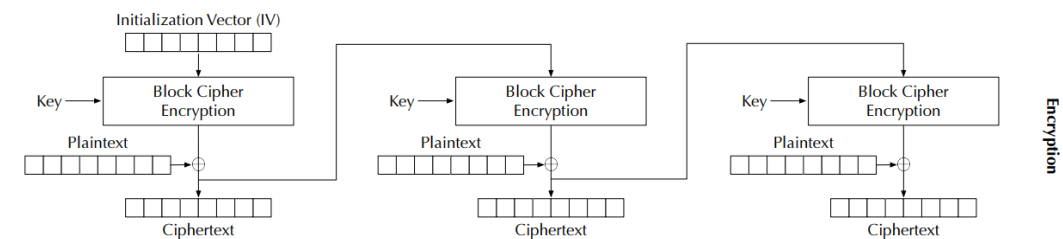


Electronic Codebook (ECB) mode encryption

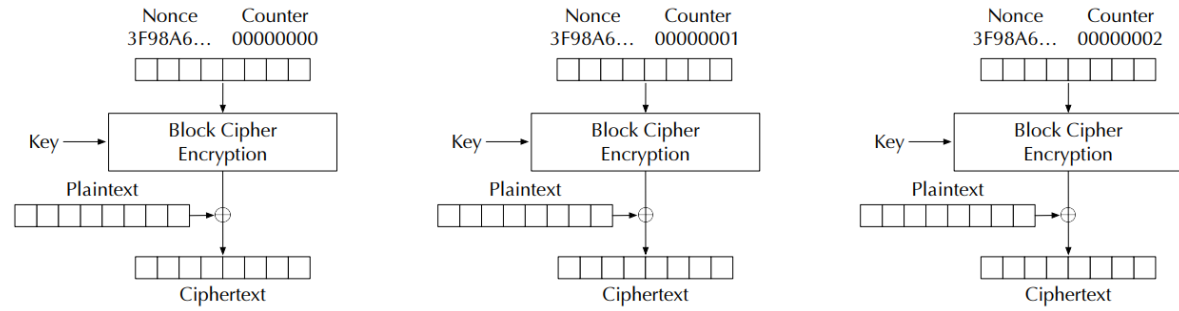
## CBC



## Cipher Feedback (CFB) Mode



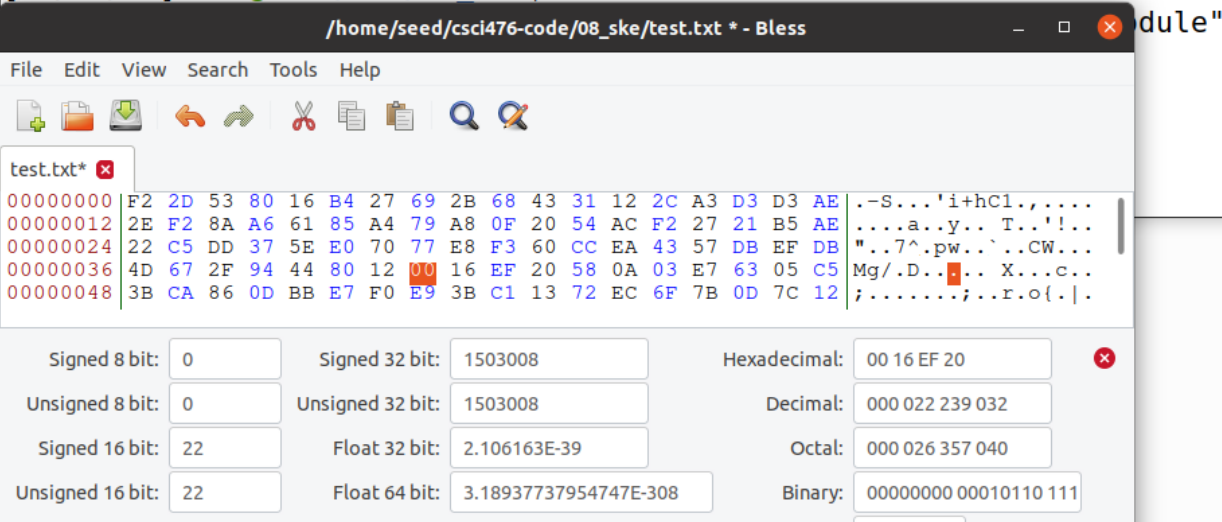
## Counter(CTR) Mode



You will explore these in the lab

# Corrupting a Ciphertext + Recovering

```
[04/10/23]seed@VM:~/.../08_ske$ openssl enc -aes-128-ecb -e -in nevermore.txt -out test.txt -K 00112233445566778899AABBCCDDEEFF
[04/11/23]seed@VM:~/.../08_ske$ sudo bless test.txt
```



Let's change a byte in the ciphertext using the `bless` hex editor

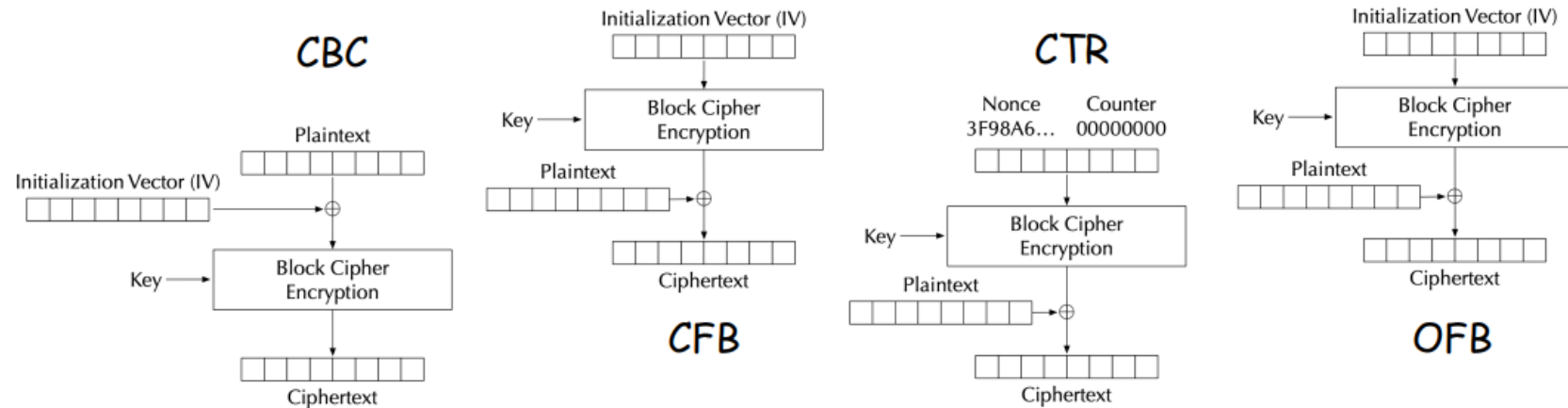
When decrypting the ciphertext, we can see

```
[04/10/23]seed@VM:~/.../08_ske$ cat test.txt
Once upon a midnight dreary, while We pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore—
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my front door.
“’Tis some visitor,” I muttered, “tapping at my front door—
Only this and a little bit more.”
```

```
Ah, disc@#0f0w0^0+00wDer it was in the beak December;
And each separate dying ember wrought its ghost upon the ground.
Eagerly I wished the marrow;—vainly I had sought to barrow
From my books surcease of sorrow—sorrow for my lost Lenore—
For the rare and radiant maiden who the angels name Lenore—
Nameless here for some more.
```

# Initialization Vectors and Common Mistakes

- Initialization Vectors have the following requirements:
  - IV is supposed to be stored or transmitted in plaintext
  - IV should not be reused -> uniqueness
  - IV should not be predictable -> pseudorandom
- Some modes w/ IVs:

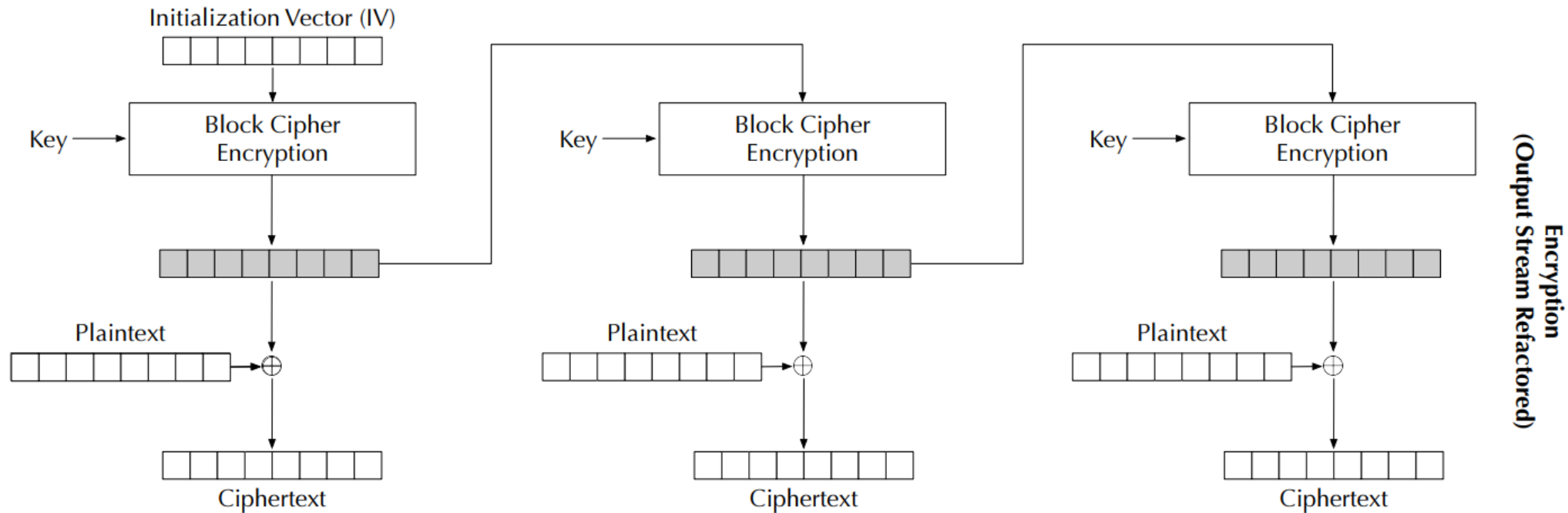


# IV should not be reused...

## Scenario:

- Suppose attacker knows some info about plaintexts ("known-plaintext attack")
- Plaintexts encrypted using AES-128-OFB and the same IV is repeatedly used...

**Attacker Goal:** Decrypt other plaintexts

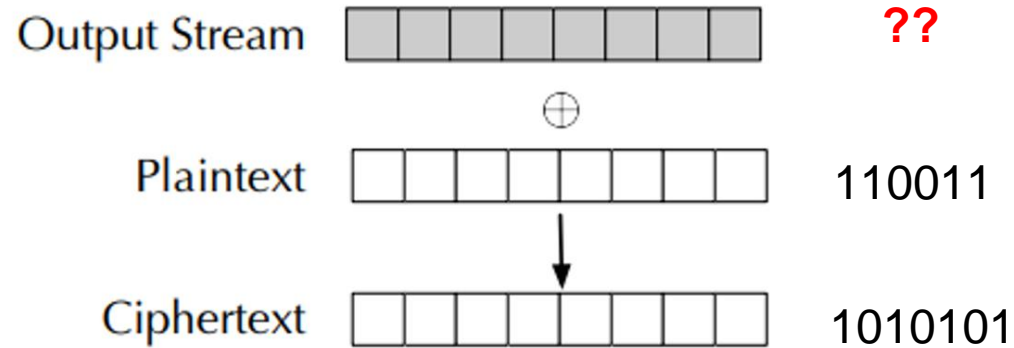


# Chosen Plaintext Attack:

Suppose we have the plaintext: 110011

And the ciphertext from that plaintext: 101010

Can we recover information about the key used? Can we decrypt other plaintexts?

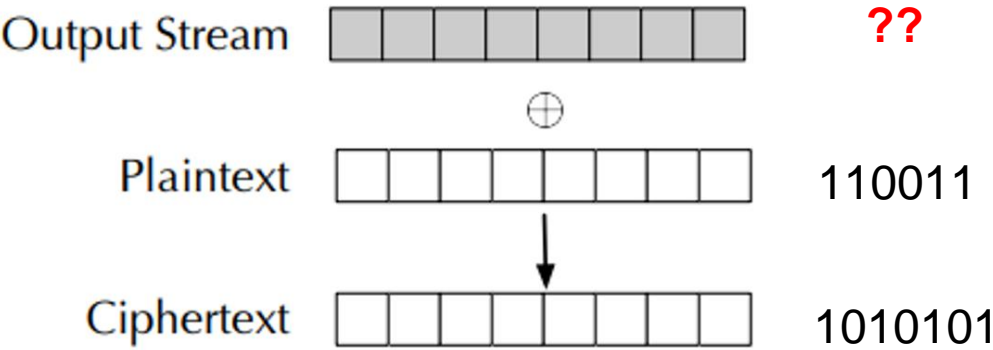


# Chosen Plaintext Attack:

Suppose we have the plaintext: 110011

And the ciphertext from that plaintext: 101010

Can we recover information about the key used? Can we decrypt other plaintexts?



We can XOR P and C to key our key/IV value!

A handwritten calculation showing the XOR of the plaintext and ciphertext to recover the key:

$$\begin{array}{r} 110011 \\ \oplus 101010 \\ \hline 011001 \end{array}$$

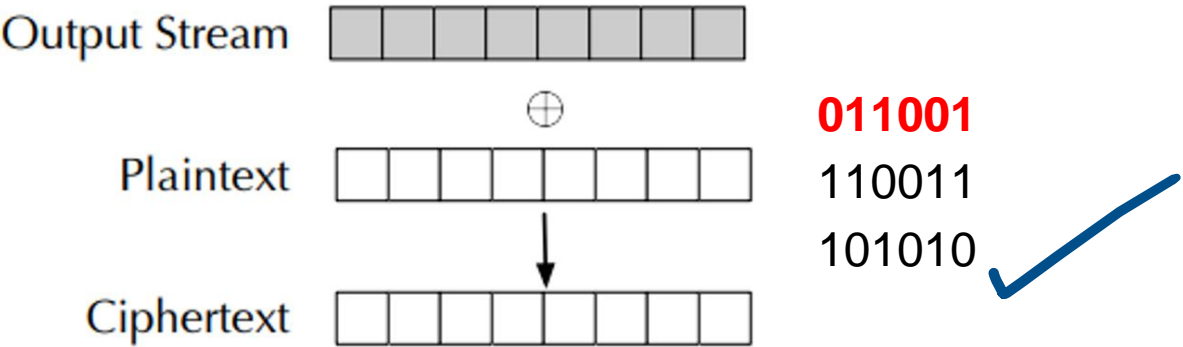


# Chosen Plaintext Attack:

Suppose we have the plaintext: 110011

And the ciphertext from that plaintext: 101010

Can we recover information about the key used? Can we decrypt other plaintexts?



We can XOR P and C to key our key/IV value!

A handwritten calculation showing the XOR of the plaintext '110011' and ciphertext '101010'. The numbers are written in blue ink, with a blue circled plus sign ( $\oplus$ ) to the left. A horizontal blue line separates the inputs from the result. The result, '011001', is written in green ink below the line.

Knowing that an encryption scheme uses the same IV + key .... (you will do this on the lab)