

# CSCI 476: Computer Security

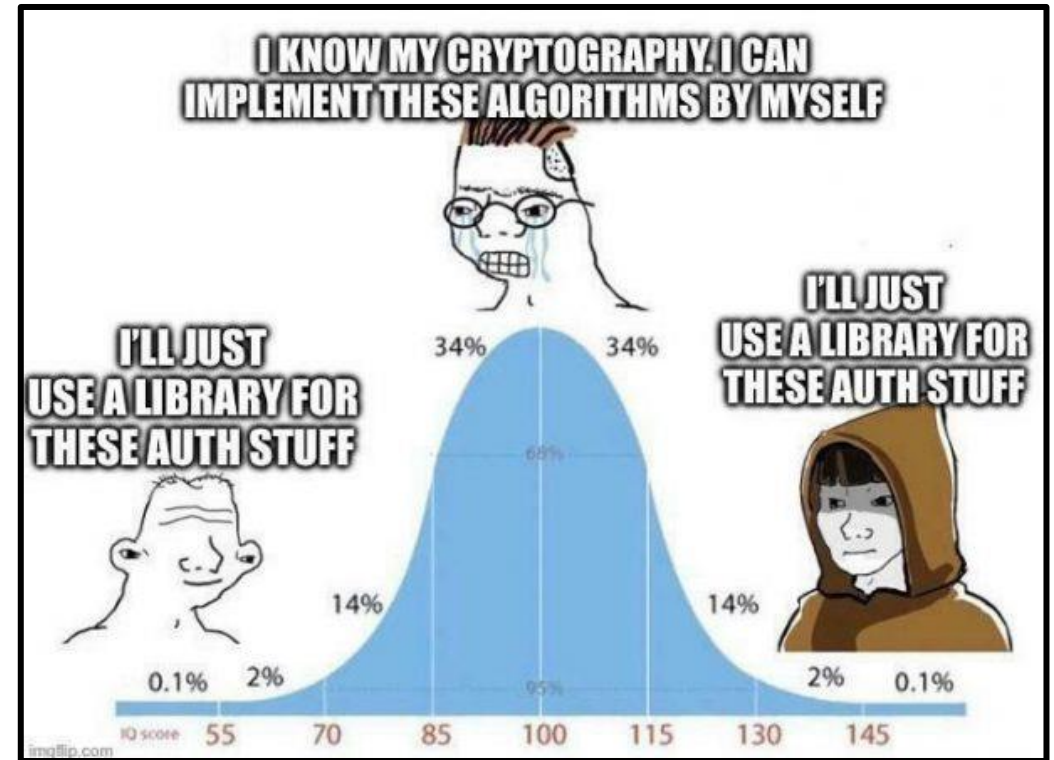
Secret Key Encryption/Symmetric Cryptography (Part 2)

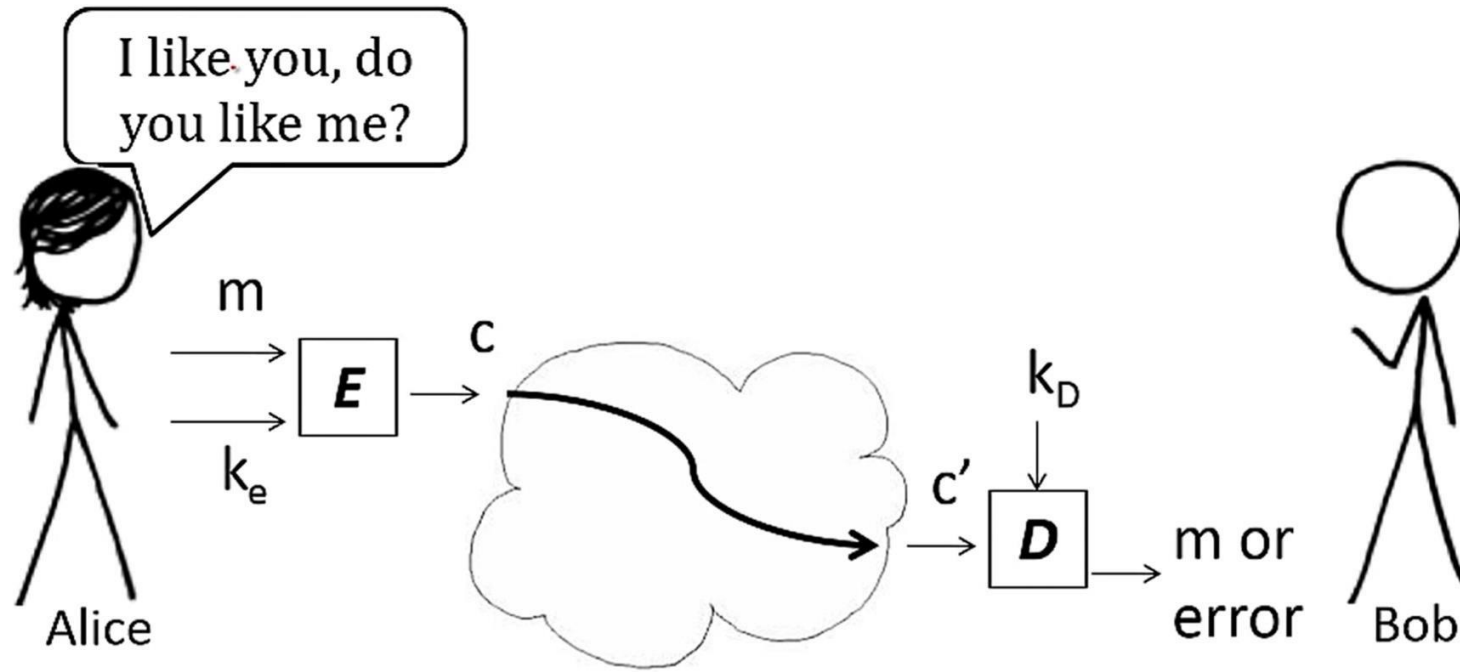
Reese Pearsall  
Fall 2024

# Announcement

Lab 7 due on **Sunday**

No in-person class on Tuesday  
→ I'll post a recording to the website





## 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

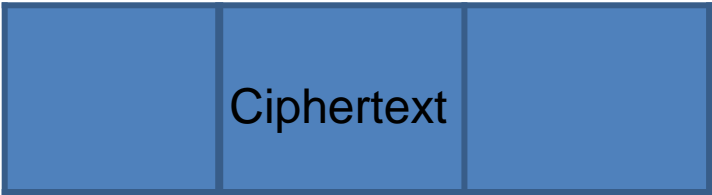
Block 2

Block 3

$\oplus$

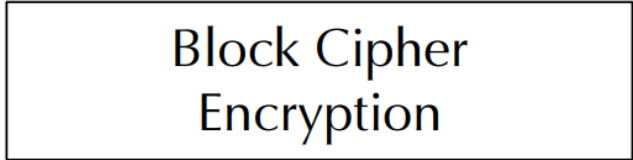
$\oplus$

$\oplus$



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

Key



*n* bits

Plaintext

Ciphertext

*n* bits

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



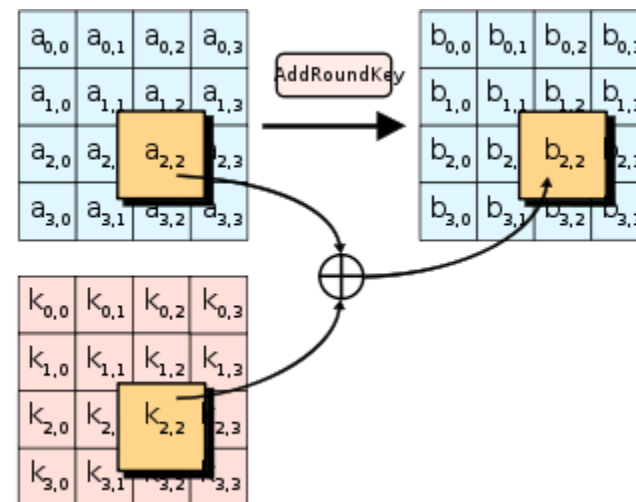
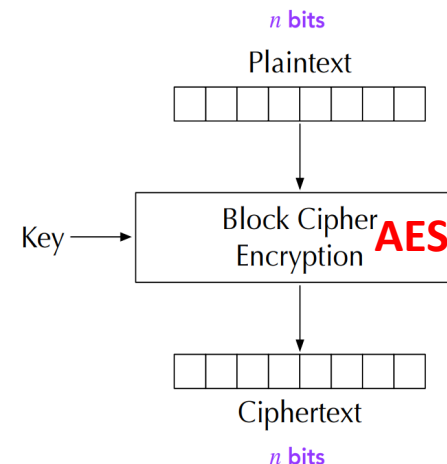
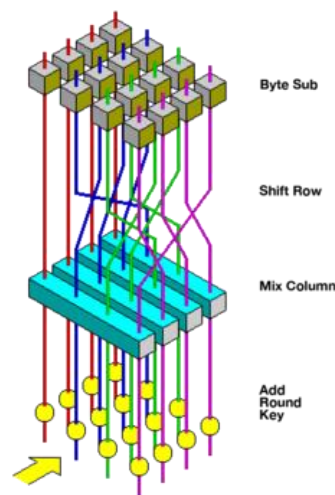
- Even small differences in plaintext result in different ciphertexts
- Blocks in plaintext that are the same will also have matching ciphertexts

# 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

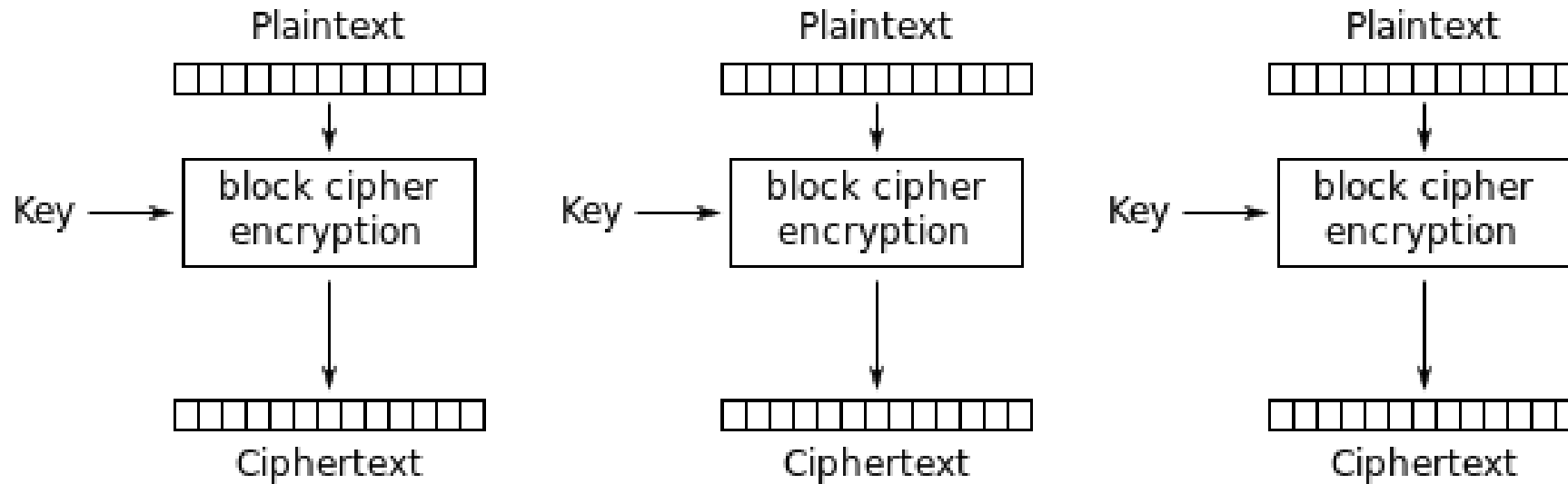
- Electronic Codebook (ECB)
- Cipher Block Chaining (CBC)
- Propagating CBC (PCBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

**All block ciphers!**

**Mode of Encryption** is used to determine how to handle encryption of a plaintext message that consists of several blocks

*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[/0fp0,00p0hyr[000k>
0000000|000>000g)k.0{0+V0;000d000000i
*z%VA;0000lf0v0?00u0$00Z%00T0GfZse
^
0000?C0!00c0JśK0i0Qb00 !C000U0u000>@000)9gm
;00p.~0f0^Ē0?0.0r^00"0000000[000z0;
[0![0 000000a0_0000E&Di
60yN0?oc00w#0~0000w00?0)+80i03C5:0q00 p800000^/S0Q0[0~5'0+Y0uc0C00
04000aq1Y0000I0000uk00s0000%j070/FP00,x0>0i0X0^0T00zg00C00G000FR,
000fP@|0009h,0{H0g%600@e~0@eZDx'Gp]B/0[11/09/22] seed@VM:~$ █
```

# 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\2tZ10N00i0K000'D0mvsJ060L00000*p006n0
0000t0i0Zq000v0p00'00f"0000D0
00000[/0fp0,00p0hyr(000k>
00000C0!00c0J5K0i0Qb00 !C000U0u000>@000)9gm
;00p.-0f0^E0?0.0r^00"0000000[000z0;
000000a0_0000E&Di
60yN0?oc00w#0-0000w00?0)+80i03C5:0q00 p800000^/S0Q0[0~5'0+Y0uc0C00
040000aq1Y0000I0000uk00s0000%j070/FP00,x0>0 0X0^0T00zg0f0C00G000FR,
0000fPe010009h,0-{H0g%000@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 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.
```

# 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
```

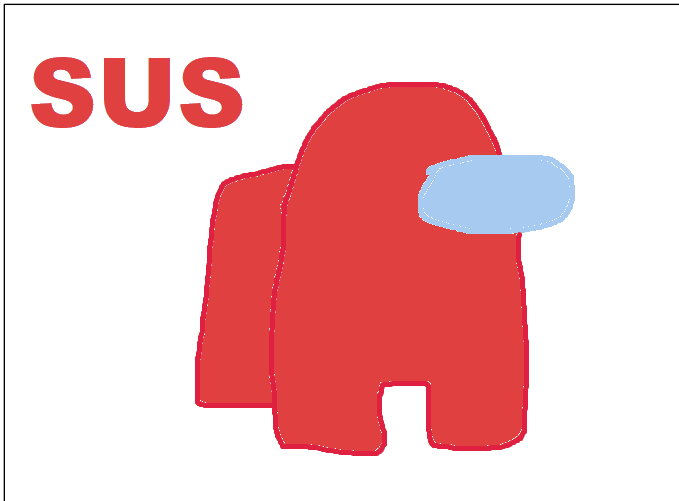
Changing the key used for decryption wont decrypt correctly!

```
[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>K!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换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



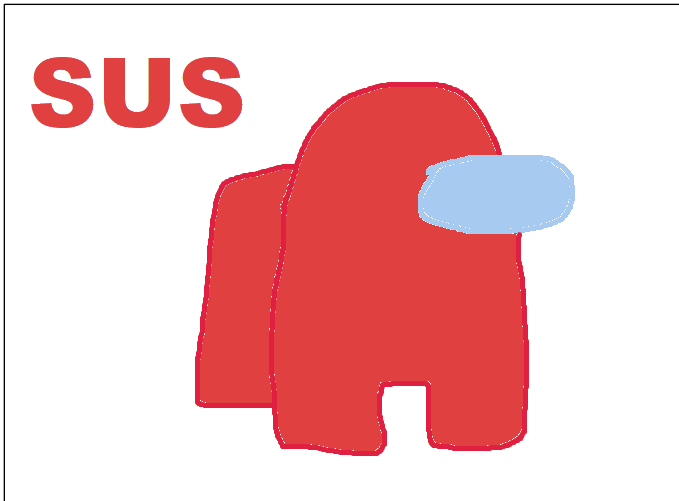
When encrypting images on the lab, make sure you use a **.bmp** image

(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

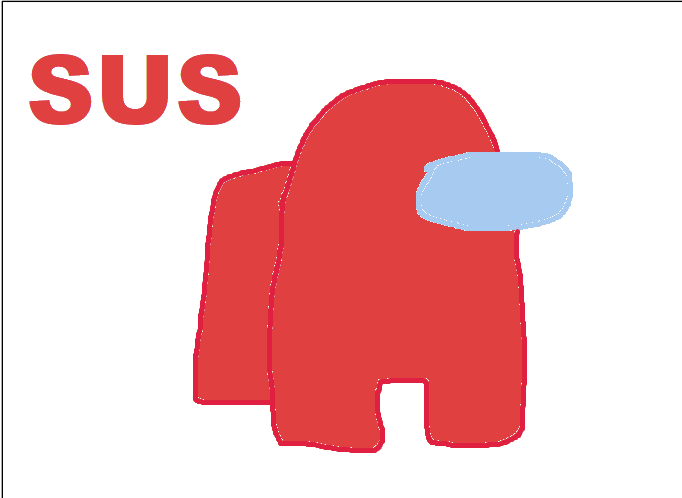
When we encrypt the image, the header will also get encrypted

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



01011010101011101011011011010101010101010100101

100101001101010111011100101100010011000001011011000011100110101011101100000  
0010111100100101010110111011011101111111100110101101100100100100010010  
0100000100000001010001101010001010010100000110010001100000011011110110000010  
1110001000001100011011101110001011000001000000110001001011001111000001001  
0000000010110110110010101101110011111011110111001000001011001000111001101  
1000101010001001010101010111000000110110110110001110110010001011111011010  
11101000001001100100111000011001100010100010011110100011101000101000001100011  
100111110011001001010010001100101110110111100011100011101100011101101110  
1101001111111011100110010110010011100100100110001001000001010010100011010111  
100000101100010011100010100010100100101010111011001110100111010010100000111  
10110000010101001100101011111011111011100010000110010101000101110000101  
1101011111111101110000111000001000100000100111001110011011100000101000011011  
11001000011001110111011100001001001011110011110010011101100100010011100100  
110101100111100101100000011011111101110010100110000110011011101010100100  
010100100100001101110000011001100011110110010101000010010101011000100  
00101111101000011011101110111001010100100001100001110110010100000110000  
01101011011100111001001100101110110010000110000101100100100000101101000110  
00101110001111011010110010011101010101111000100101010001010000110101010011  
1101001001111111011011001010011001010101001111000101110110010111111011  
0110110000010101001111011000111110001000101011100001101010000011010  
10111010101110101110111110010111110001011110010000110111010111011100111  
00110000000111110001011101110100000101010101011100001100101100010010101  
1010000100110000010101010110110110010000100001001111111101010001101  
011110100110110111101101111000011011101101100011011101100111101000110  
111110110011100100101010111001101111110010001110100000110001010110011000110  
1011001110011011110110001010110101010000110110001111100000110110000101001  
110011000100111110001001100010011011111100010101100100011101100010110101  
1000101011011001010100001000001100010010101110000011100001000010101110  
001100011101000011111111000101111000110110110010111011000011110010111001101  
00010001100010110000101000111011111001010101101000011000101100100101110001  
11101110110011001011101111000110100100111010000101010100101110100101  
1000011001011101010001000010010110001011111100111

Body of the image

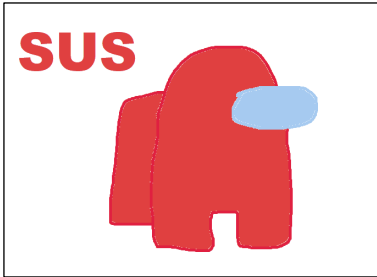
Header

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 00	C7 00 00 00	01 00 00 00	18 00
00000020	00 00	68 D2 00 00	13 00 00 00	13 00 00 00	00 00
00000030	00 00	00 00 00 00	23 2E 00 00	26 31 00 00	28 33 00 00
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!*



01011010101010111010110110110101010101010101010100101

[illegible]

enc.bmp

```

010001100100101111100001110110000110000111000111101011000001100001000111010000110011
0010010101000001010010000010100001010000011110010111010110101010100001010111
100001111111010000010100000101010100000101000111010101000100101001110001000
0100010001010100111010100111100101010101000111001101000100111011000001010001100001
1010010111000001100000110000100111101010001110101010111010101001111010100011110010100
100101000010100100010010100000101110000101011101010101100011100011100011110010111
10000111100001111001001110101001100111001000010100000000011110101010011
11010101001111011001000000011000000010100010010101000111010011010110100
111100010010000011101111000101010111010101010000010011000101001100101
00010111010100111000100010011110010101010011110100010001010011101010000
011110000010101001001010010001110101010010001001110100001001010101110000
00000000100001111100111110101010110001100010010111011010101001001000001011
010000010100100010001010101011010101100000001010001001010110100101011001010
11010101111101000001000001000111010111010101010001010000000101100001110110
11100110001010000100111111110110001101110111101010100010010000010010000100101
1001100110010010011110100111110111100101110000101010000000010100010100100
11011010100010100111100100101000101111101001001001010100101000001100
0010010010111100100110011110100101001111101010100010100010100010100011000010
0001101010001000001100101111110010000010100100001010000111111111000001
110101010010101010010111010000010100101000100011000101000001000011101010010000
0100100000100001011101010010100000101101101000010001001111001000010001110010010000
010111110010100100001010011110000111101100100010101001110101010110011
0101010000100100010000000100100100000000001111000010010101010101000101001
01010101000000000100000110000101001010000101000001111111110100100010111000
100110000001011100001101010101010101001000101101011011100111000110000100100
110000101000111101100100011101010100111101110011100010101111011010000
11111111110101010010100101010101000100001101001110101000010001000000111011
101100100100110001011010010101001010000110000010111000001010101000010101010
1000111010001111110100000100101000100011000100010000100001110011010000111
00100010100101000010000111100111101010101010100100010111001010101010111
0100010101011010011110011000000000110111011100111100111000000000011011
1101110011110011100000000011011101100111100110000000001110111011100111
11011100000000001110111011

```

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)

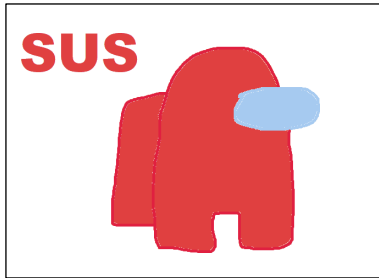


# Using OpenSSL to encrypt w/ ECB

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

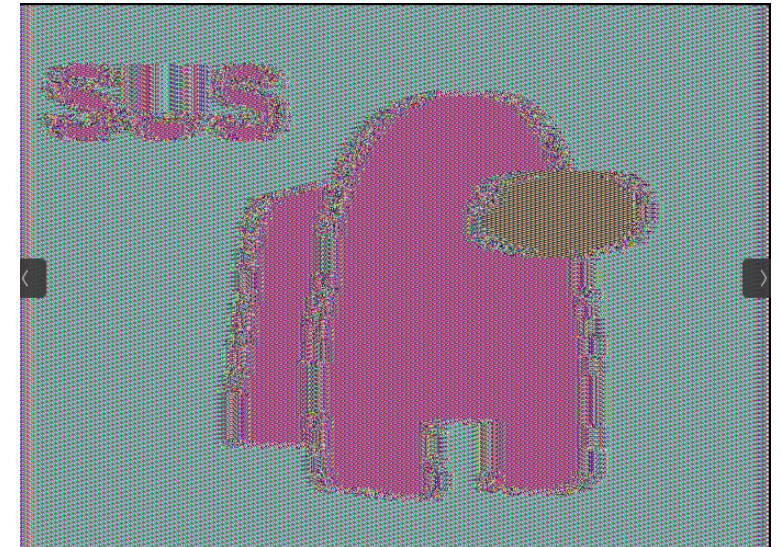
*final.bmp*

sus.bmp



```
01011010101010111010110110110101010101010100101
00110010100000010100100000101000011110010111010111010110000110110111
1000011111110100000101000001010101000001011000111010101000100110100110001000
0100010001101100111011001111001101010011100111010010011101110000001010001100001
101001001110000011000001010001100111101010001101011010111101010100011110010100
100101000101001000100010100001011110001010111010101011100011100011100010111
1000011011000011111001001101010100110011100100001101100000000001111010101100111
110101011001111101110001000000111000000101000100101010001110100110101110100
1111000110010000011110111100010101101011101011010100000100110001010011001011
0001011101101001110001001010111100101101011001111101000100011010011101100000
011110000101011001001100100011101101100110001001111010000100101010101110000
0000000100001111110011111010111000110010011011101110101001001100000111
01000001010010001000110110101110101110000001010001001101011101001101100110
111010101111101000001000001000111011011101101000110100000010111000011101110
11100110001011000001001111111011100011011101111101101100010010100000100101
1001100110011001011110100101111011111001011100001010000000010100010100100
1101110110001101100111100100010100010111111010010011001101100101010000011100
00100100101111100100111001111101011010110011111010110100011010001010011100010
000111011000100000110011011111001100000101001010001101100001111111111000001
111010100110110110011011101000010100101000100011100010100000100001110110010000
010010000010100011011101101001011010000010011101110100001000100101111001000100
011011110010100110001011001011100001111101110010100101101011101011110011
0110110000100100010000010010100100000010001111000010010110101011000101011
101010100010110101001101000001100000111011011000001101001010001111000110001
01101001100000000100000111000010100101000011111111110010000101111000
1001100000010101110000110110110110101001100010111011101110011100011000010010
110000101010011111011001000111010101100111110111100111000101011110111010000
11111111101010100101100101101010100010000110100011101101100001010001000001111
110110010010100111001001110100110110001011000011100000101111000001010101010
10001110101001111111001000010010110001100011000100010000011110011101000111
0010000110010100001100001111100111100111010110010001010111100101101010111
01000101101011101001111001110000000001101111011100111100111000000000110111
1101110011110011100000000011011110111001111001110000000001101111011100111
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
```

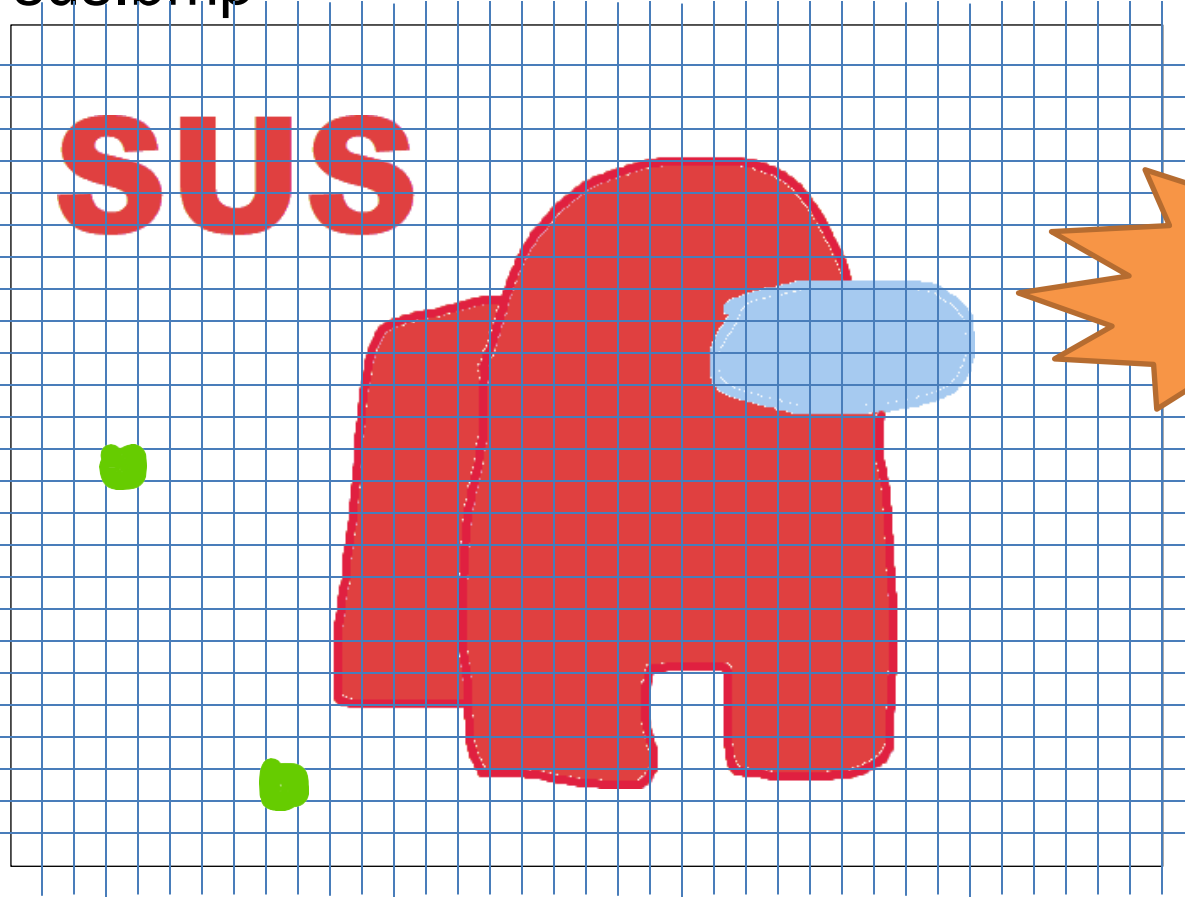
Take the first 54 bytes of the original image (header)  
Take everything after the 54<sup>th</sup> byte of the encrypted image (image)



# 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”

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

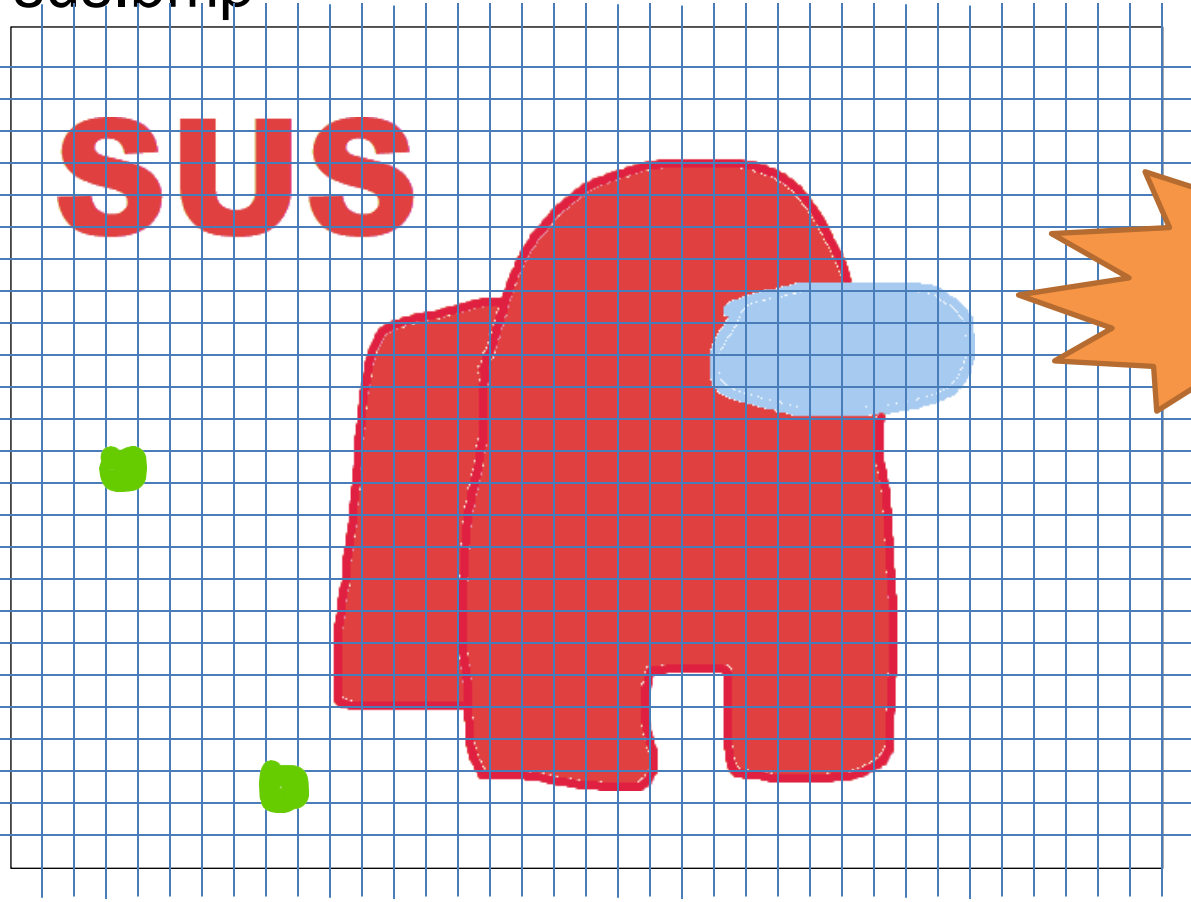
Dividing this image up, we can see that there are many blocks that are the exact same!

# Using OpenSSL to encrypt w/ ECB

*Why does this suck?*

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

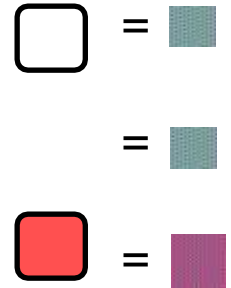
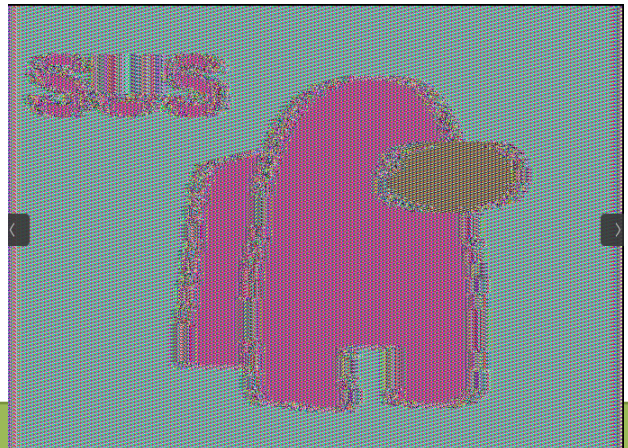
sus.bmp



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**

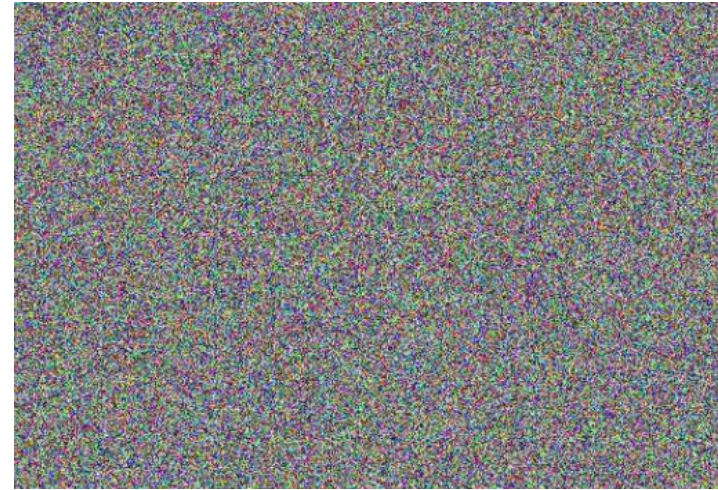


# Using OpenSSL to encrypt w/ ECB

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

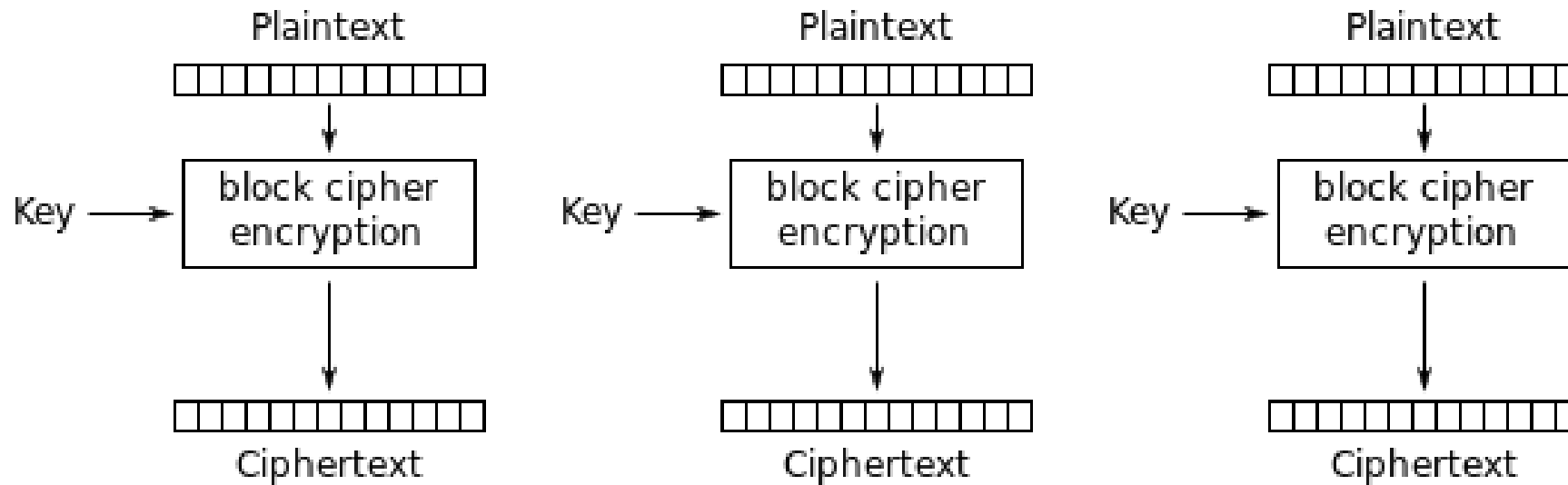
```
[11/09/22] seed@VM:~$ openssl enc -aes-128-ecb -e -in capy.bmp -out enc_capy.bmp -K 001122  
33445566778899AABBCCDDEEEE  
[11/09/22] seed@VM:~$ head -c 54 capy.bmp > header  
[11/09/22] seed@VM:~$ tail -c +55 enc_capy.bmp > body  
[11/09/22] seed@VM:~$ cat header body > final_capy.bmp  
[11/09/22] 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



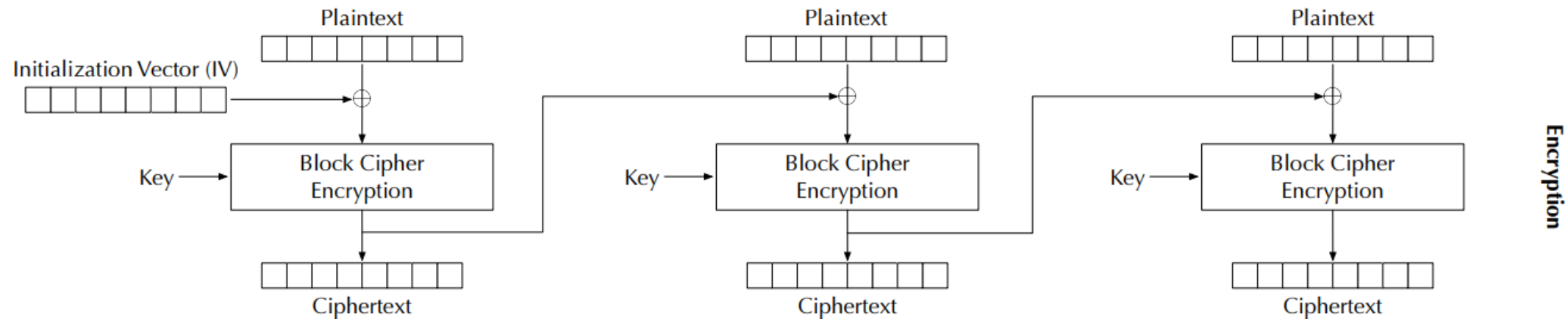
## Electronic Codebook (ECB) mode encryption

### 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

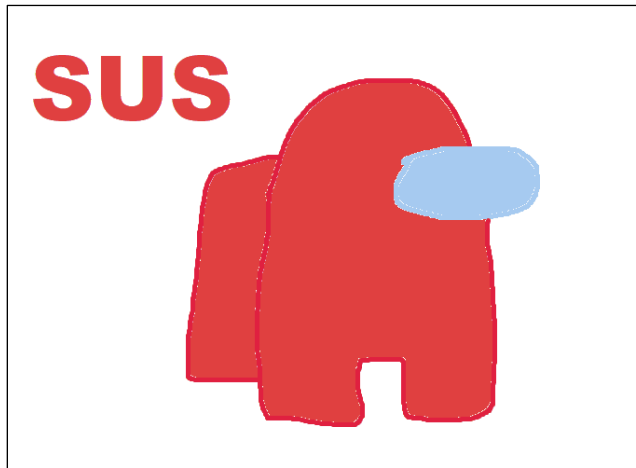


Introduces **block dependency**

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

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

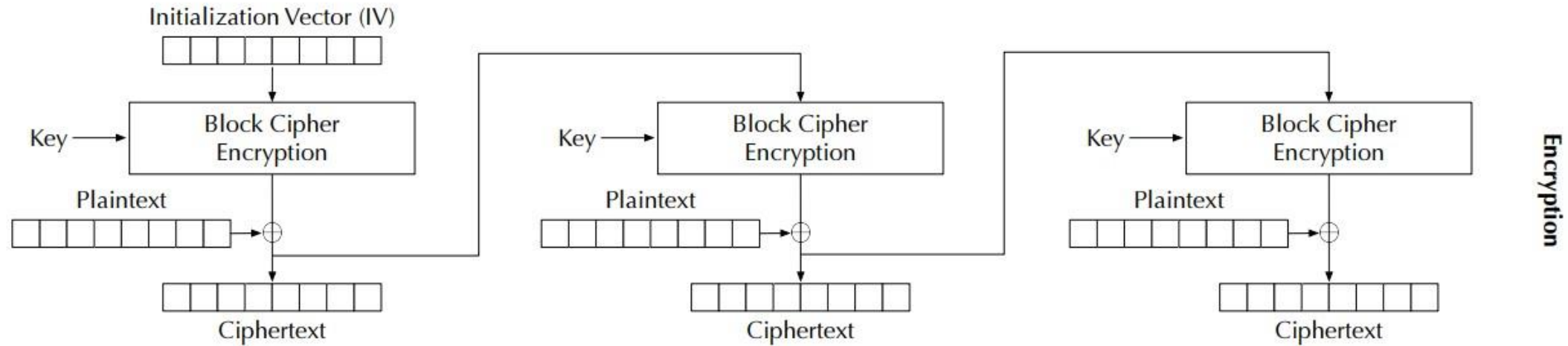
```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt \  
-K 00112233445566778899AABBCCDDEEFF \  
-iv 000102030405060708090A0B0C0D0E0F
```

```
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
```

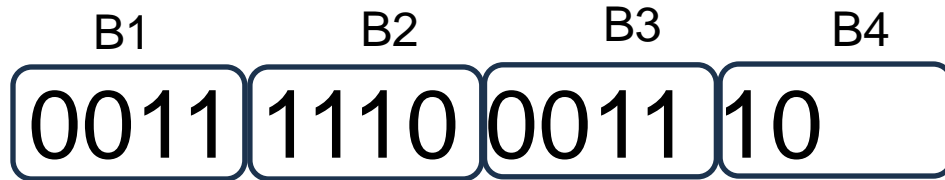
```
[11/10/22]seed@VM:~$ ls -al | grep "cipher"  
-rw-rw-r-- 1 seed seed 576 Nov 10 00:36 cipher2.txt  
-rw-rw-r-- 1 seed seed 592 Nov 10 00:36 cipher.txt
```

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?

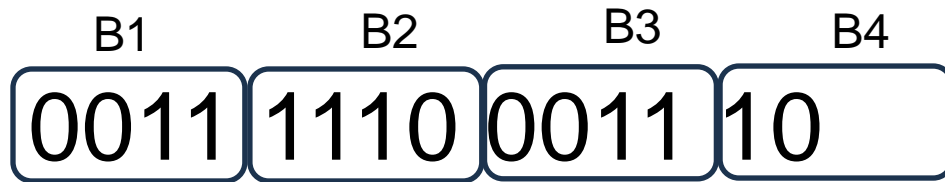


This block is not 4 digits... we need to add more so that our encryption method works!

# 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
```

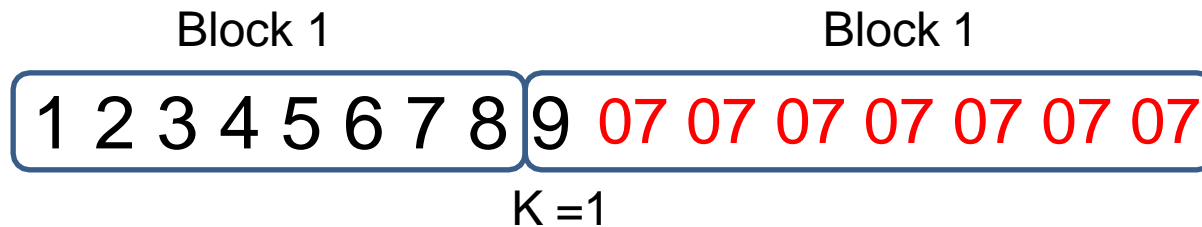
7 bytes of 0x07 are added as padding data

```
[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

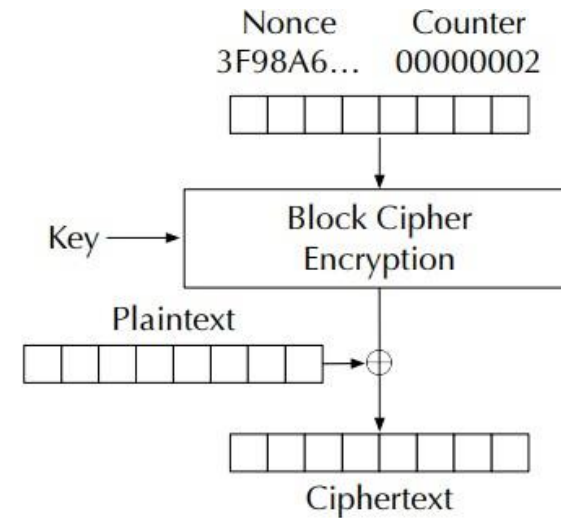
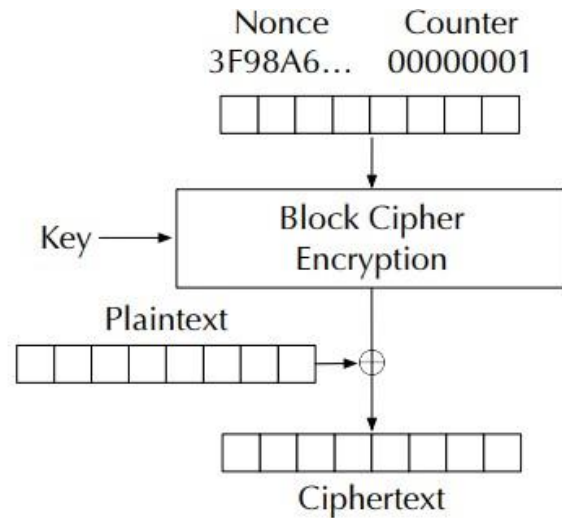
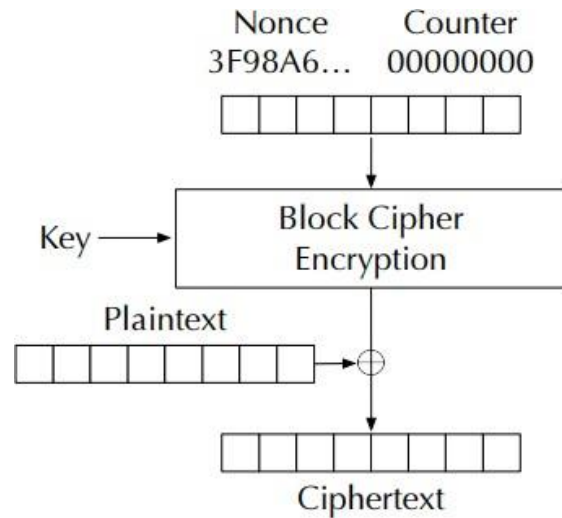
$ ls -ld cipher3.bin plain3_new.txt
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 cipher3.bin
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 plain3_new.txt

$ 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
```

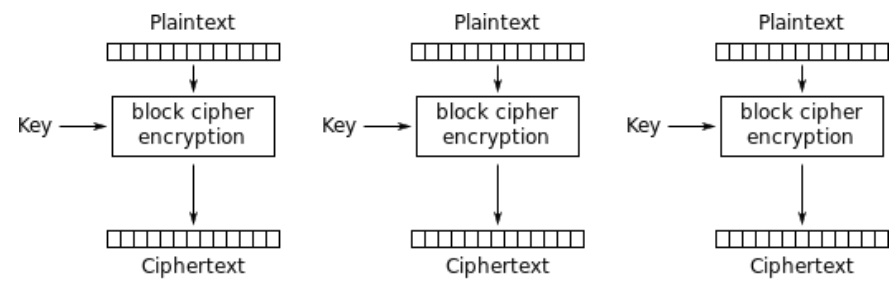
- Size of plaintext (plain3.txt) is **16 bytes**
- Size of decryption output (plain3\_new.txt) is **32 bytes** → a new, full block is added as the padding
- In PKCS#5, if the input length is already an exact multiple of the block size  $B$ , then  $B$  bytes of value  $B$  are added as the padding.

# 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)

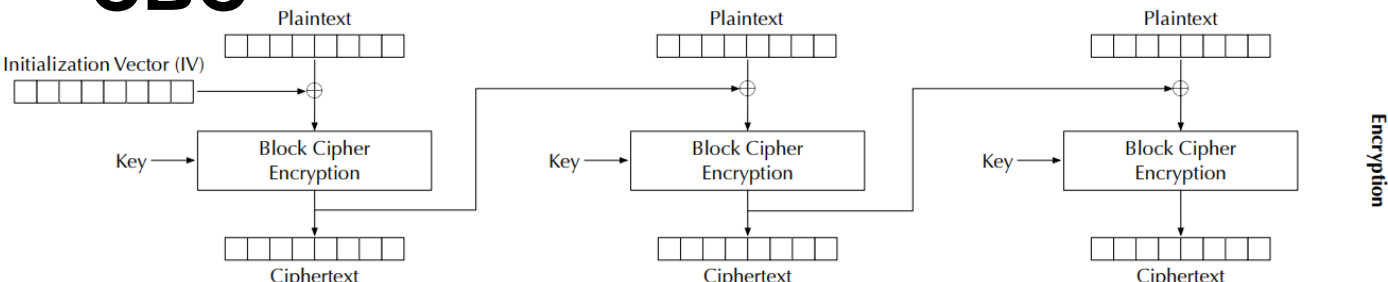


# Modes of Encryption

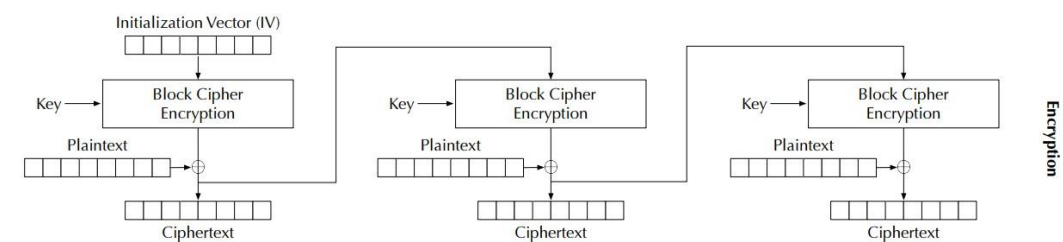


Electronic Codebook (ECB) mode encryption

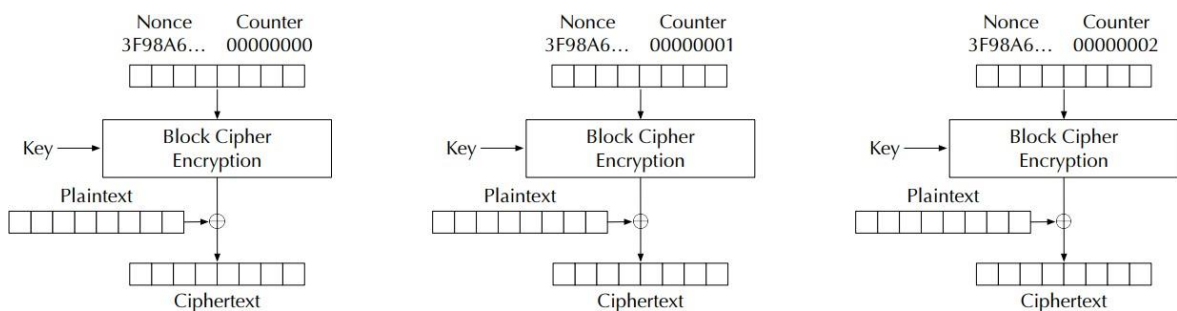
## CBC



## 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
```

File Edit View Search Tools Help

test.txt\*

00000000	F2	2D	53	80	16	B4	27	69	2B	68	43	31	12	2C	A3	D3	D3	AE	.-S... 'i+hCl,....
00000012	2E	F2	8A	A6	61	85	A4	79	A8	0F	20	54	AC	F2	27	21	B5	AE	....a..y.. T..!..
00000024	22	C5	DD	37	5E	E0	70	77	E8	F3	60	CC	EA	43	57	DB	EF	DB	"..7^.pw...CW...
00000036	4D	67	2F	94	44	80	12	00	16	EF	20	58	0A	03	E7	63	05	C5	Mg/.D... X...c..
00000048	3B	CA	86	0D	BB	E7	F0	E9	3B	C1	13	72	EC	6F	7B	0D	7C	12	;.....;..r.o{. .

Signed 8 bit:	0	Signed 32 bit:	1503008	Hexadecimal:	00 16 EF 20
Unsigned 8 bit:	0	Unsigned 32 bit:	1503008	Decimal:	000 022 239 032
Signed 16 bit:	22	Float 32 bit:	2.106163E-39	Octal:	000 026 357 040
Unsigned 16 bit:	22	Float 64 bit:	3.18937737954747E-308	Binary:	00000000 00010110 111

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

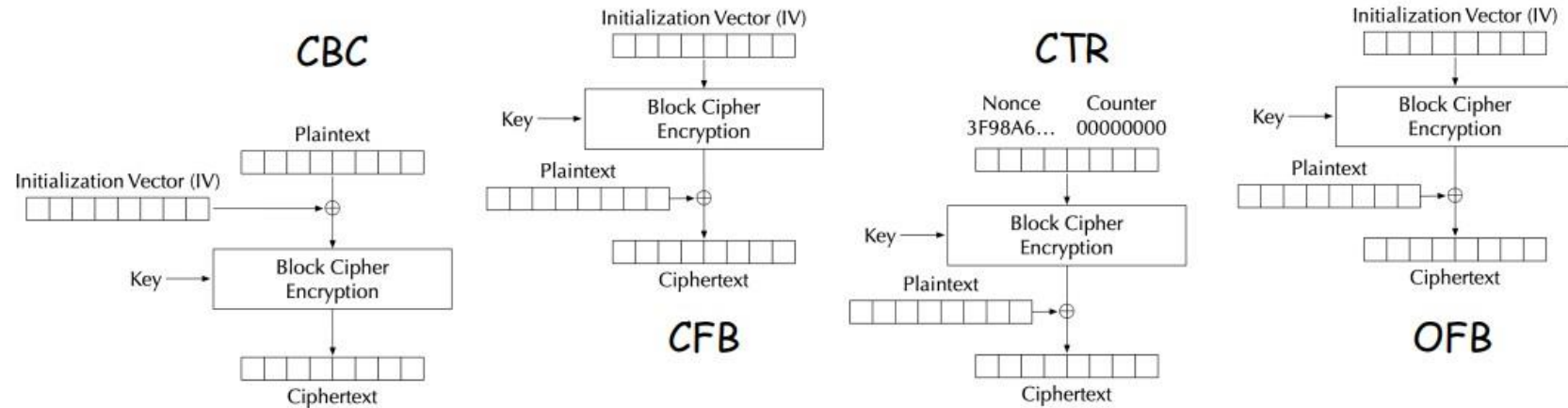
When decrypting the ciphertext, we can see

```
[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:



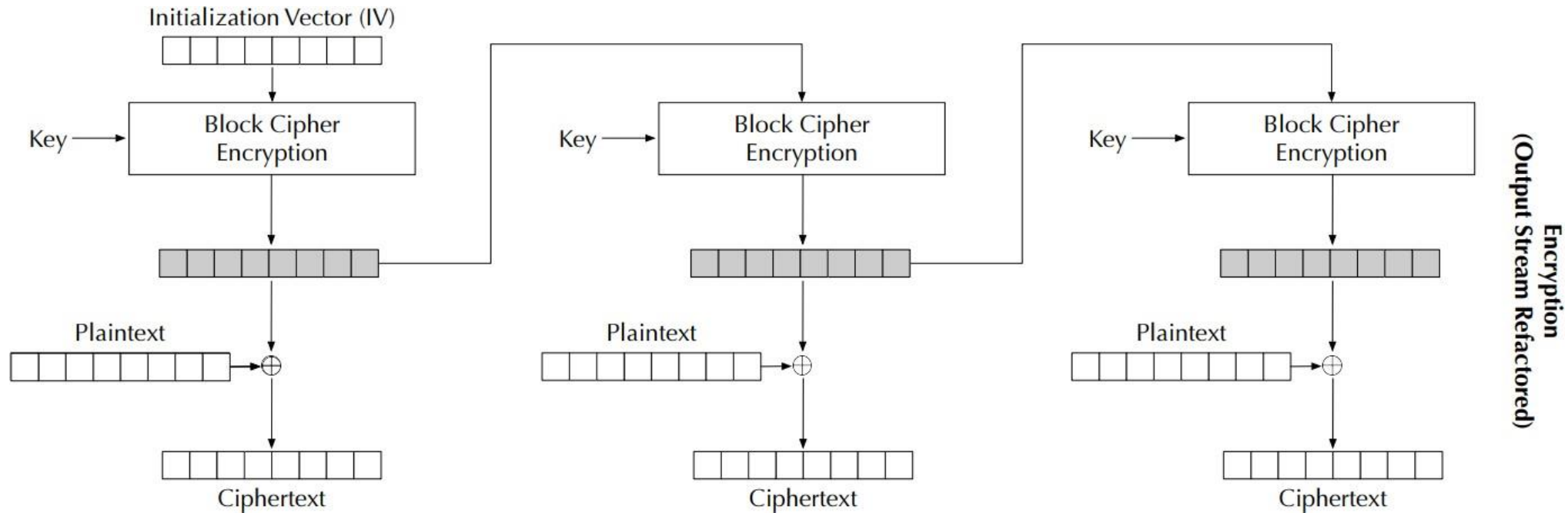


# IV should not be reused...

## 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

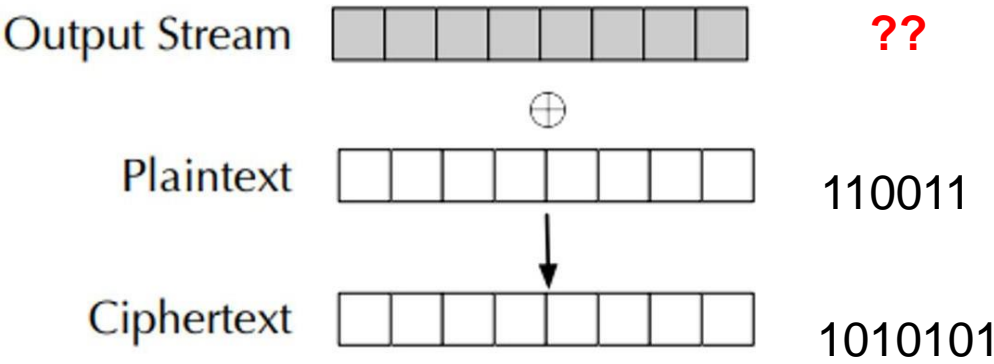


# Chosen Plaintext Attack:

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?

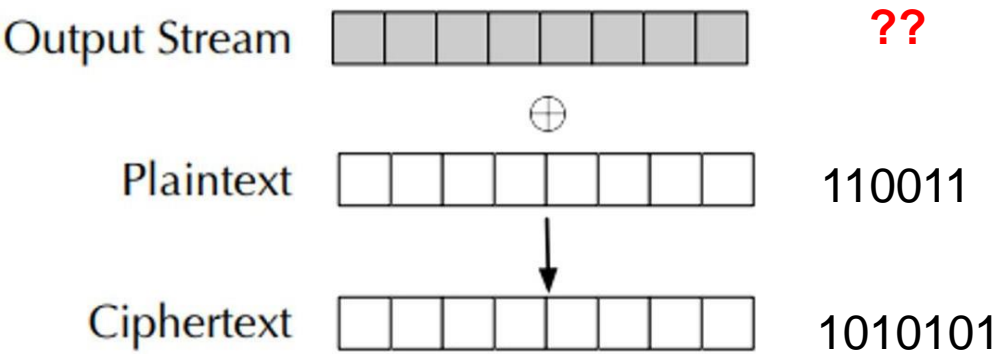


# Chosen Plaintext Attack:

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:

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

The result, 011001, is written in green.

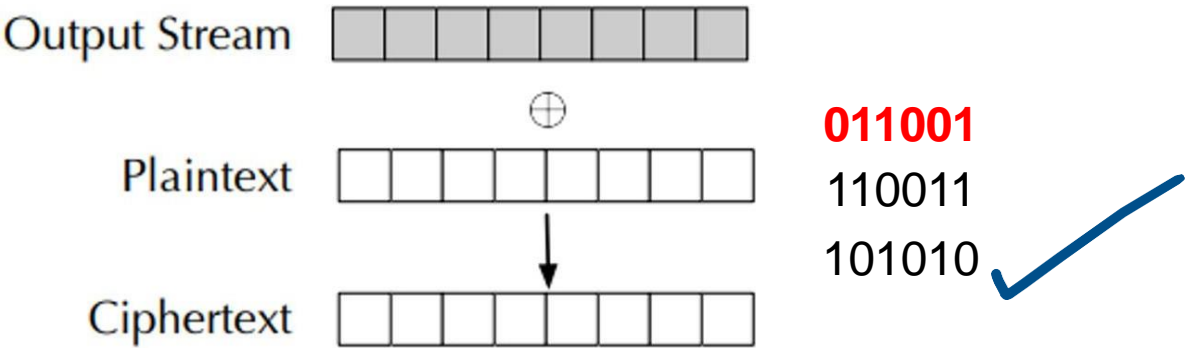


# Chosen Plaintext Attack:

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!

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

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