CSCI 476: Computer Security

Secret Key Encryption/Symmetric Cryptography (Part 2)

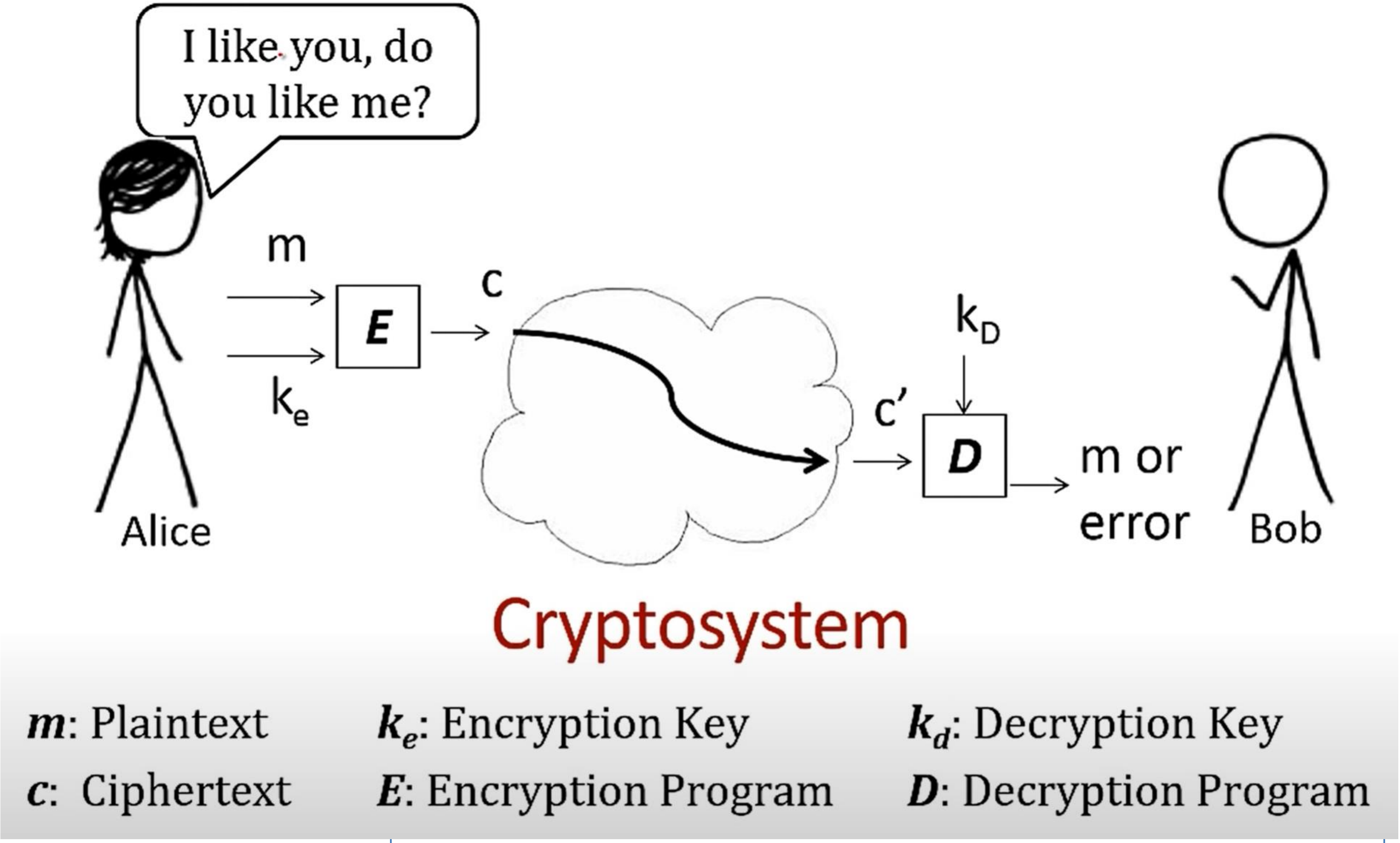
Reese Pearsall

Spring 2023

Announcement

Lab 8 due date changed

4/16 → 4/19



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

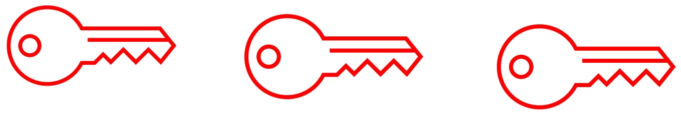
Split in messages into fixed sized blocks, encrypt eac
block separately

Hello there world

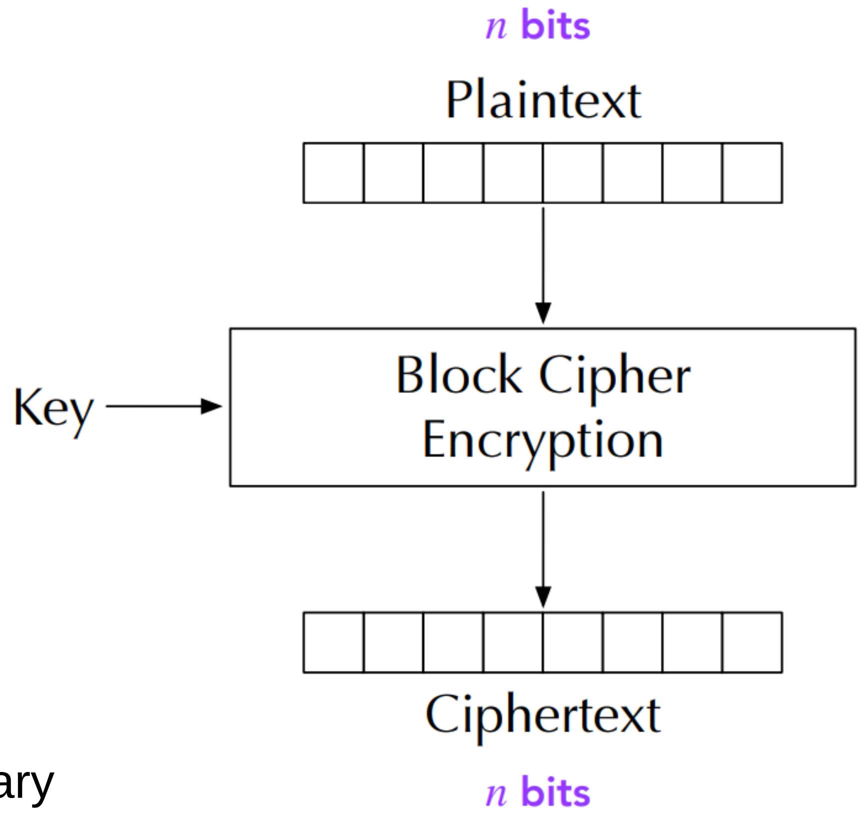
01101000	01100101	01101100
01101100	01101111	00100000
01110100	01101000	01100101
01110010	01100101	00100000
01110111	01101111	01110010
01101100	01100100	00001010

Block 1 Block 2 Block 3

$\oplus$ $\oplus$ $\oplus$



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



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

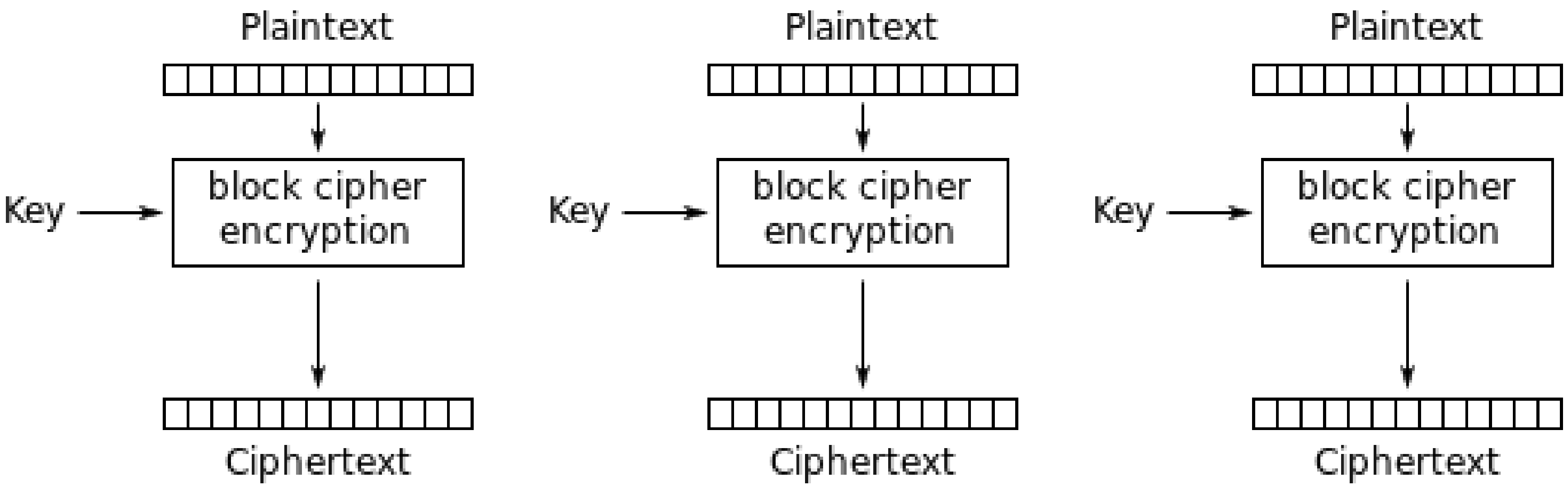
Modes of Encryption

- 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. **E**ncrypt
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

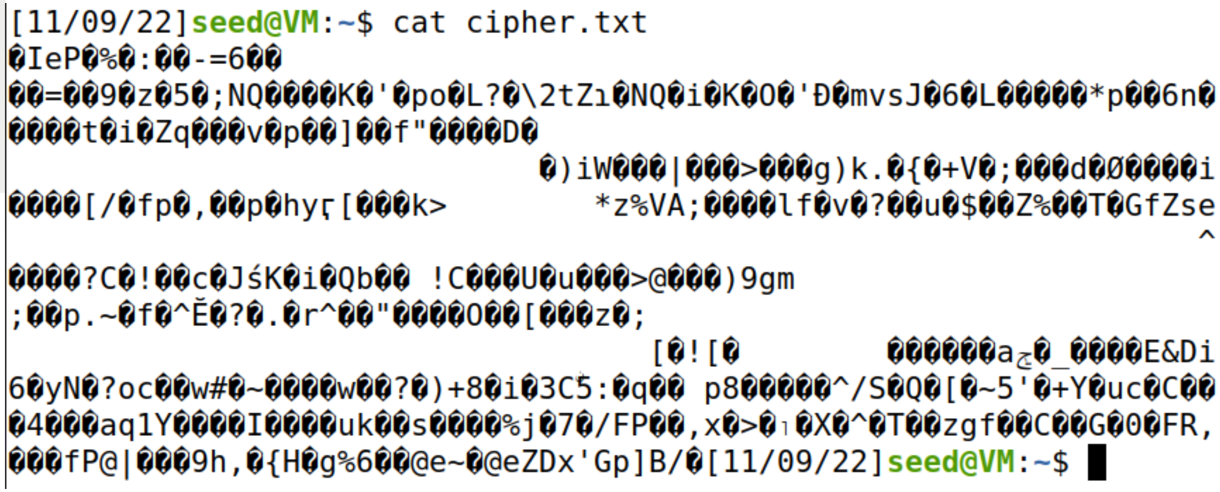
Using OpenSSL to encrypt w/ ECB

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.
```



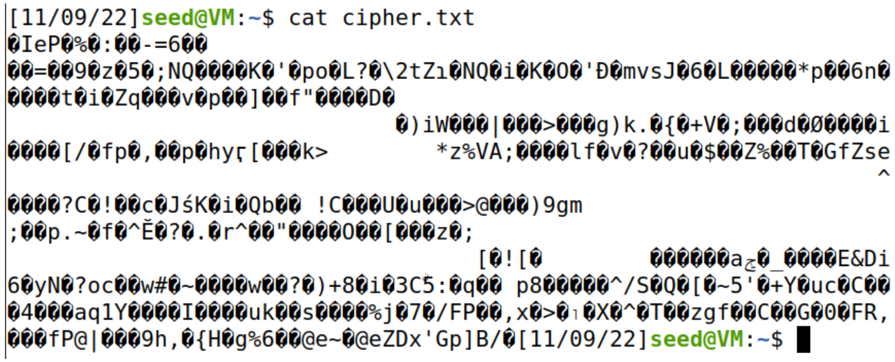
Using OpenSSL to encrypt w/ ECB

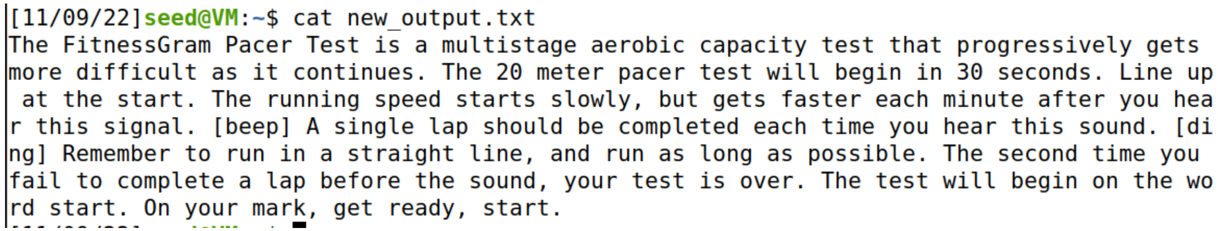
Encrypt a .txt file

```
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
```





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
```

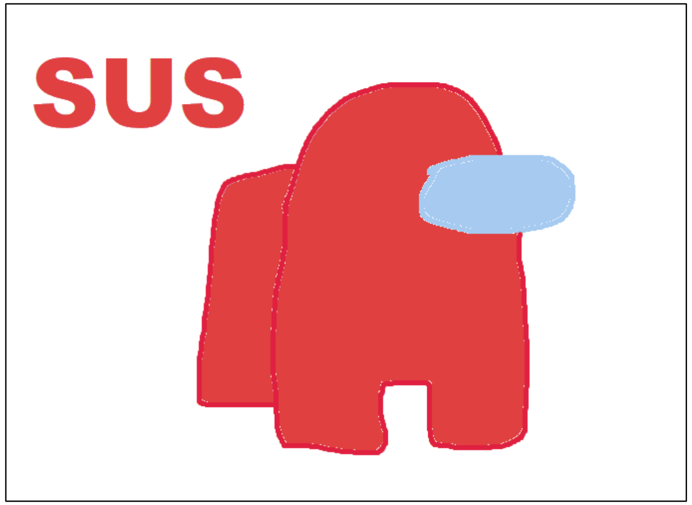
Changing the key used for decryption wont decrypt correctly!

```
[11/09/22]seed@VM:~$ openssl enc -aes-128-ecb -d -in cipher.txt -out new_output.txt -K 00
112233445566778899AABBCCDDEEEE
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.��>Ӝ!�@.�~hy4c��A}�����(��tg{�M��q��u(��KU�h�%g�zmH����(���
g'���]���5n�����kD���'L���a���7�Vf�(���K�^2��J/3;2Y�q��b���&w�O-hQ���zY��R+���C�?���ǂ�����
?�'�o��qj?�~A5J/;F.L/D?V�Ů/��fO�m�����M��t��HoDr.#.�
                                                  ����i��s*����&F/���Bv�w=
                                                                       d�>��r�Q�3�i����r�z
}d��dA�����]F����3����]:�ZX�/�?h�Y�Md�2W��w�5栨 �<�z����r��|��2�|�U�bb�[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



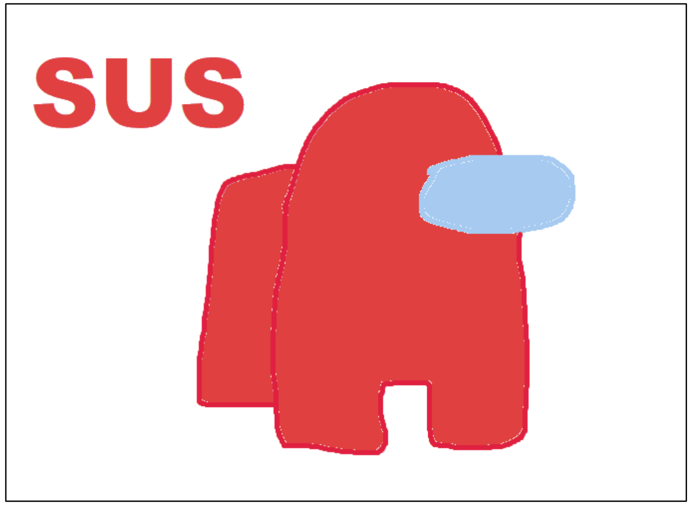
When encrypting images on the lab, make sure you use a **.bmp** image

(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

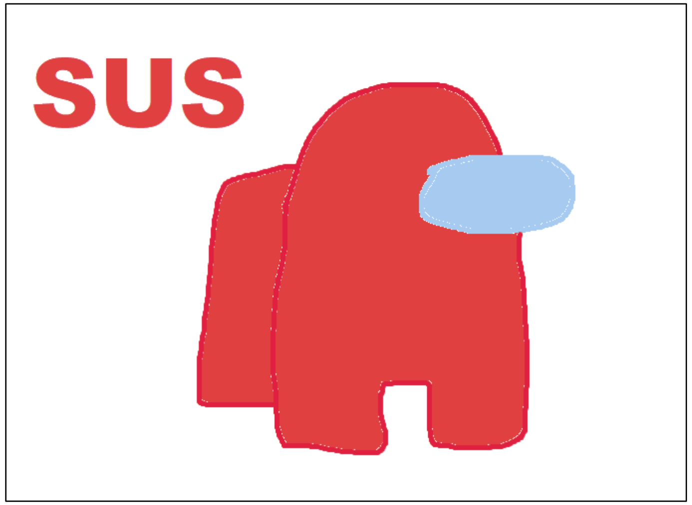
When we encrypt the image, the header will also get encrypted

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

Using OpenSSL to encrypt w/ ECB

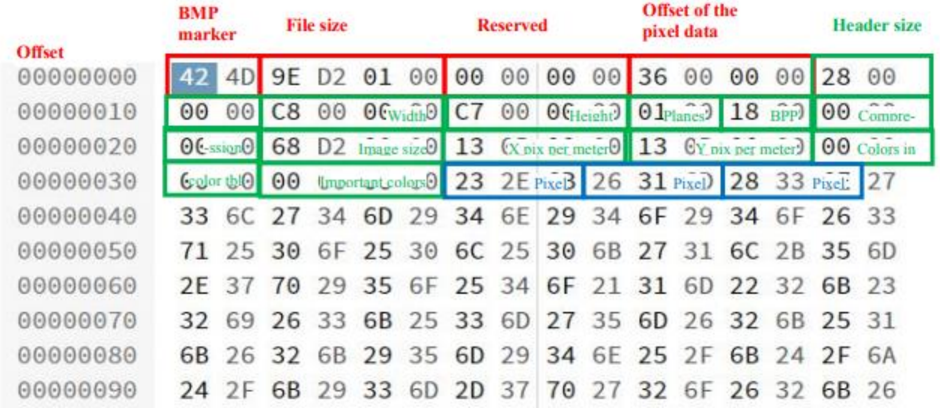
We can encrypt many things (everything on computers is just 0s and 1s). Let's try an image!

sus.bmp



010110101010101110101101101101010101010101010101100101

Header



1001010011010101111011100101100010011100000101101100001110011011010111011100000
0010111100100101011011101111011011110111111111001101011101100110010100100010010
0100000100000000101000110101000101001010000001100100011000000110111110110000010
1110001000001100011011110111100010110000000100000001110001001011001111100000001001
0000000010110110110010101101101110011111101111101111001000001011001000111001101
1000101010001001010100101010111100000011011011011100011101100100010111111011010
1110100000001001100100111000011001100010100010011111010001110100101000000100011
1001111111001100100101001000110010111101110111111000111000111101100011101101110
1101001111111101110011001011001001110010010010110001001000001010010100011010111
1000001011000100111000101000101001001010100101111011001110100111010010100000111
1011000000101010100110010101111110111111101110001000011001010101000101110000101
11010111111111110111000011100000010001000001001111001110011011100000101000011011
1100100001100111101111011100001001001010111110011110010011101100100010011100100
1101011001111001011000000011011111111011100101001100001100110111001110101010100100
0101001001000011011100000110011000111001011111011001010101000010010101011000100
00101111110100001101110111101110111001011010010000001000011110110010100000110000
0110101101110011100100110010111011001000011000010111000101010010001011011000110
0010111000111110110101110010011101010110111110001001011010001010000110101010011
1101001001111111101101101001010011011001010101001111100010111011001011111111011
0110110000010101100111101100011100011111100010001010110111000011010100000011010
1011101010111101011101111110010111111110001011111001000011011110101110111100111
00110000000111110001011101110100000001010101010101011110000111001011000100110101
1010000100110000000101010110110110111010010000100001000100111111111101010001101
0111101001101101111101101111100001101111011101101000110111011001100111101000110
11111011001110010010101011100110111111110010001111010000001110001010110011000110
10110011100110111110110001010111011010100001110110001111110000001110110000101001
11001100010011111000100110001001100111011111111000101011100100011101100101110101
10001010110111011001010100001000000111000100101010111000000111000010000101011110
00110001110100001111111110001011111000110110110010111010110000111110010111001101
0001000110001011000010100011101111100101011010110100001100010110100100101110001
1110111011001100101110111101110111110001101001001110100001010110100101110100101
1000001100101110101000100001001011100010111111100111

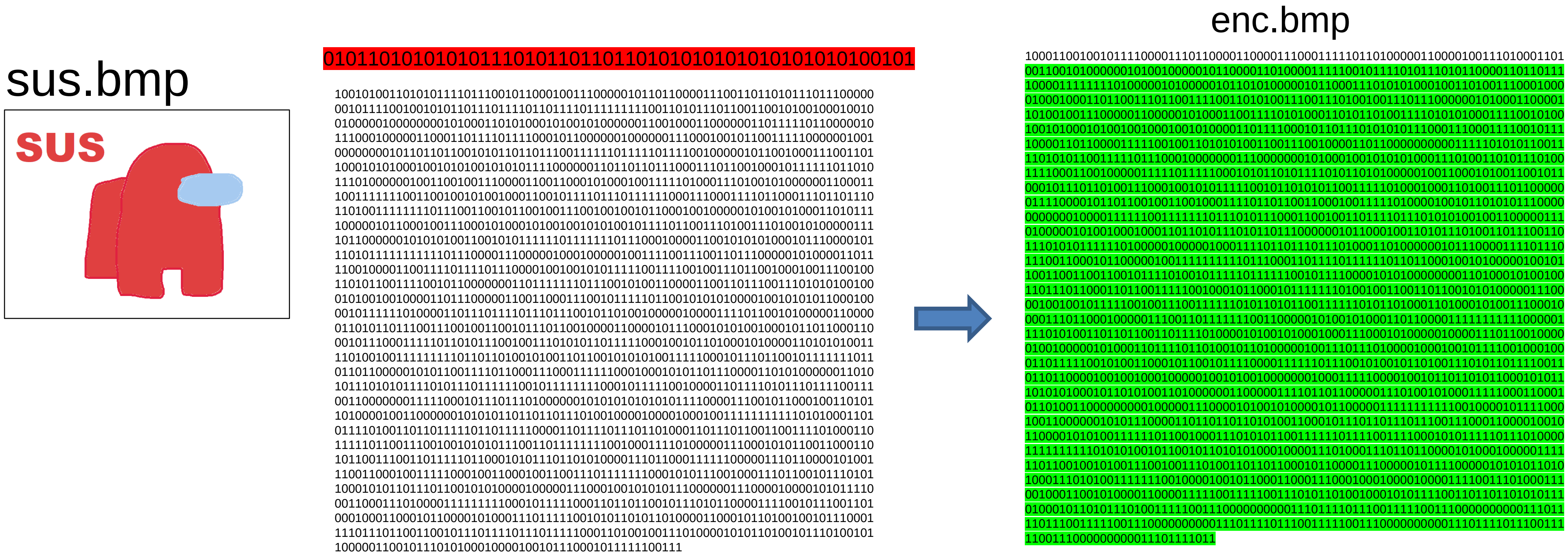
Body of the image

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

I arbitrarily selected this bit sequence for the header… this would not actually be a valid 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!



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
```

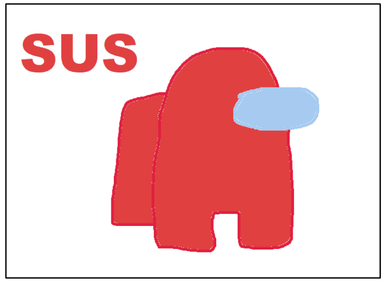
Take the first 54 bytes of the original image (header)

Take everything after the 54th byte of the encrypted image (image)

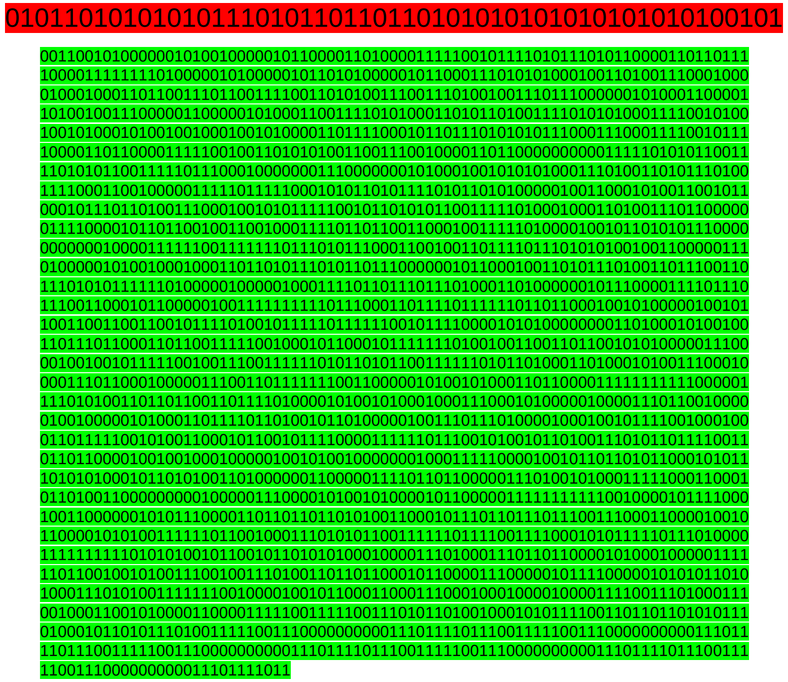
Using OpenSSL to encrypt w/ ECB

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

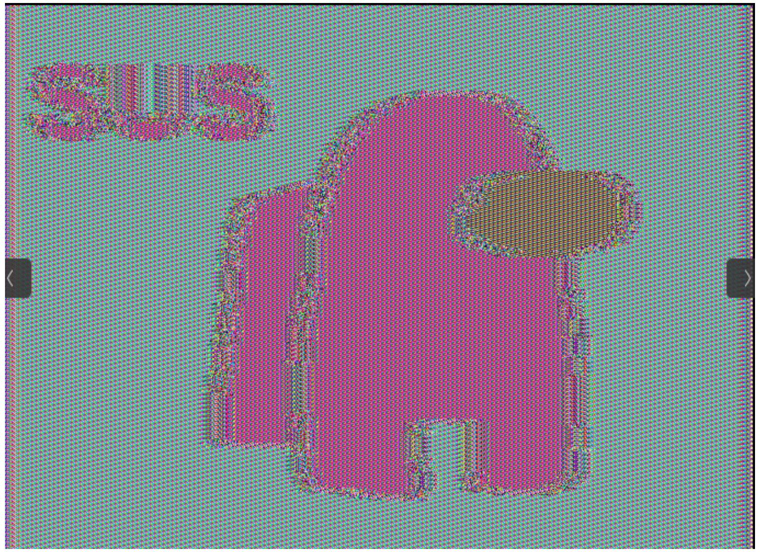
sus.bmp



final.bmp



```
[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
```

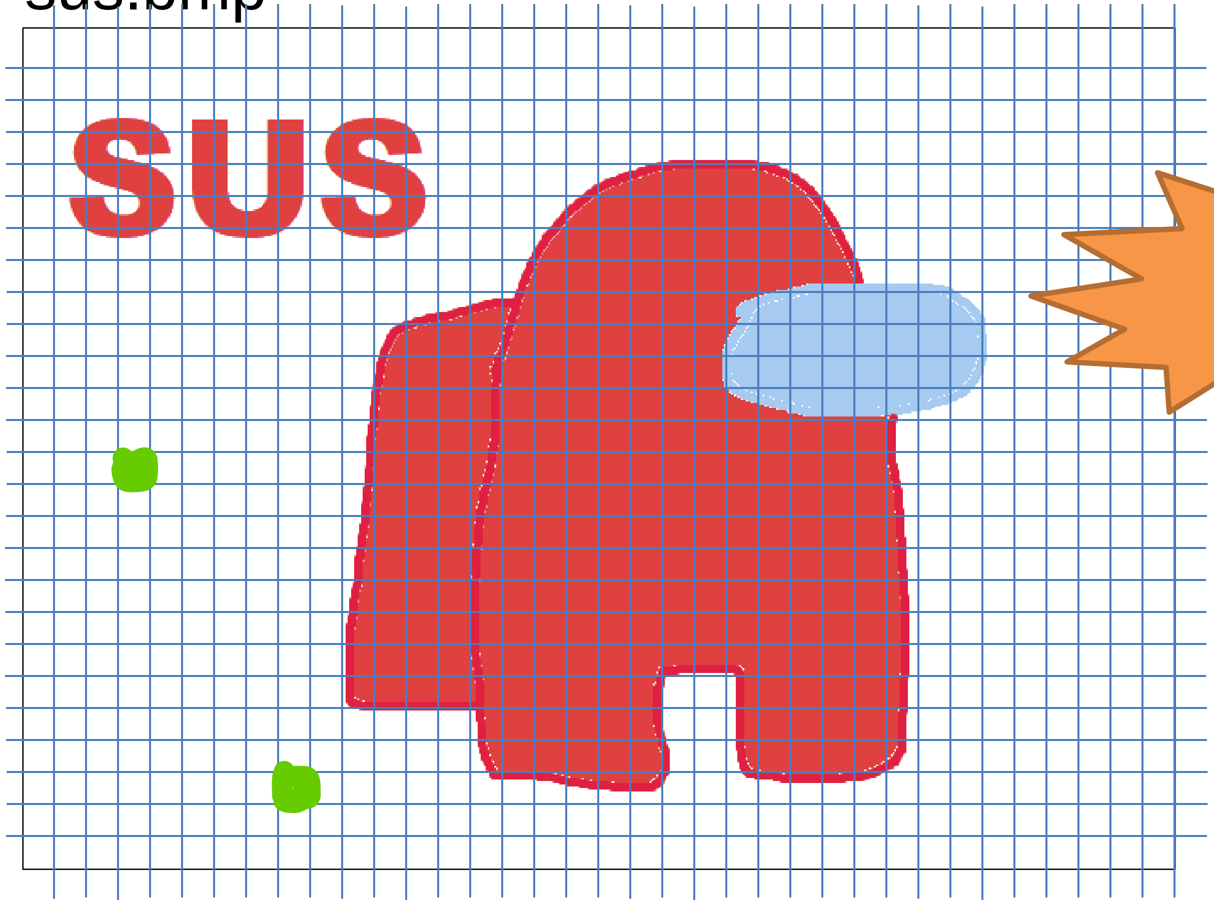
Take the first 54 bytes of the <u>original</u> image (header)

Take everything after the 54th byte of the <u>encrypted</u> image (image)

Using OpenSSL to encrypt w/ ECB

Why does this suck?

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**

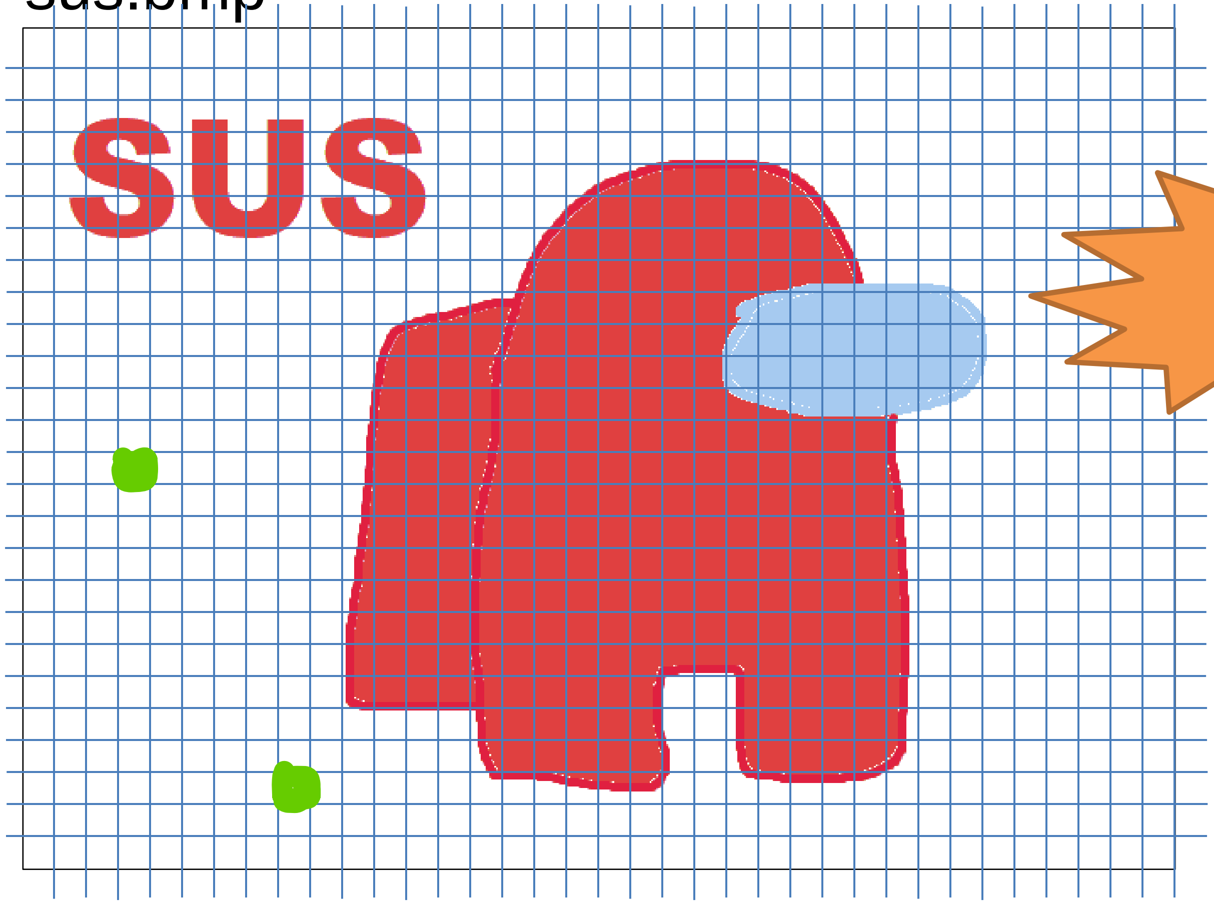
Dividing this image up, we can see that there are many blocks that are the exact same!

Using OpenSSL to encrypt w/ ECB

Why does this suck?

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

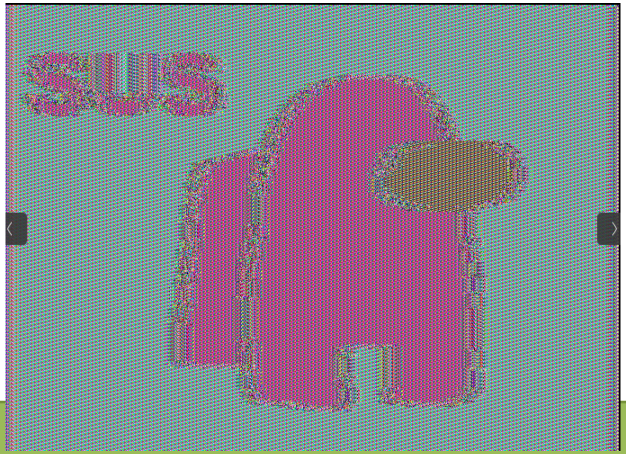
sus.bmp

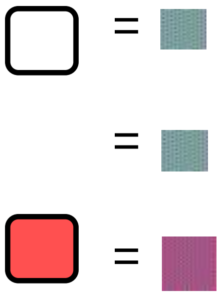


Important Properties

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

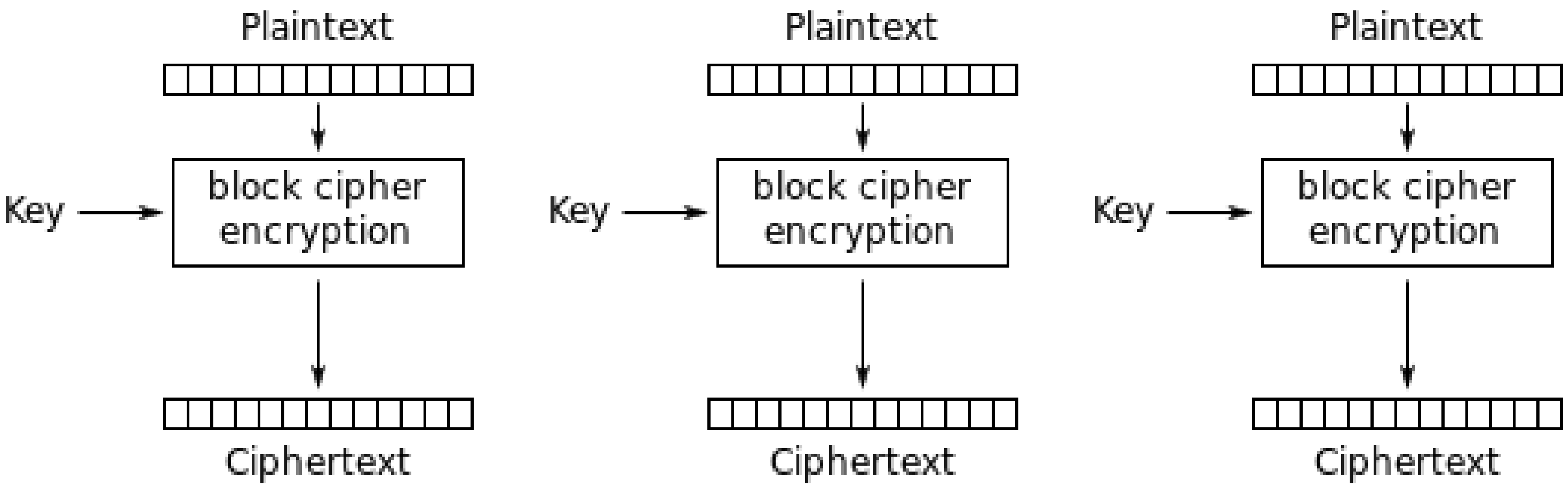
Let retry this experiment on a more **complex** image

```
[11/09/22]seed@VM:~$ openssl enc -aes-128-ecb -e -in capy.bmp -out enc_capy.bmp -K 001122
3344556677889AABBCCDDEEEE
[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
```

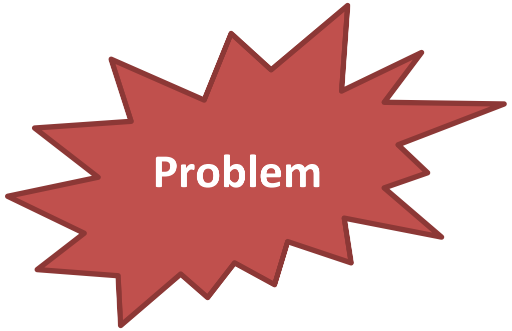
capy.bmp

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

Using OpenSSL to encrypt w/ ECB



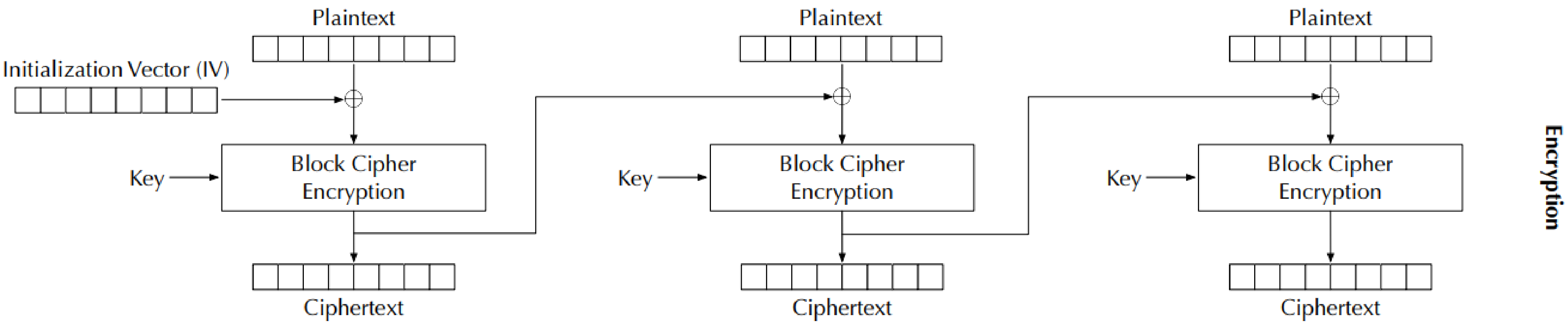
Electronic Codebook (ECB) mode encryption



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



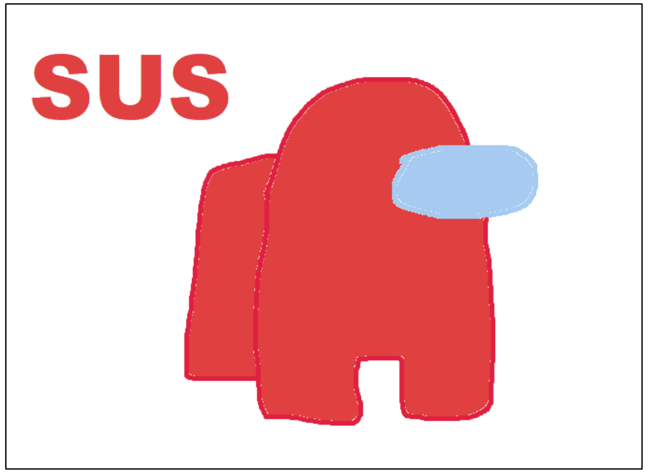
Introduces **block dependency** $\boxed{C_i = E_K(P_i \oplus C_{i-1})}$

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

Cipher Block Chaining (CBC) Mode

Reminder: $\oplus$ = XOR operator

Using CBC to encrypt images??



→ ???

You will do this on the lab.

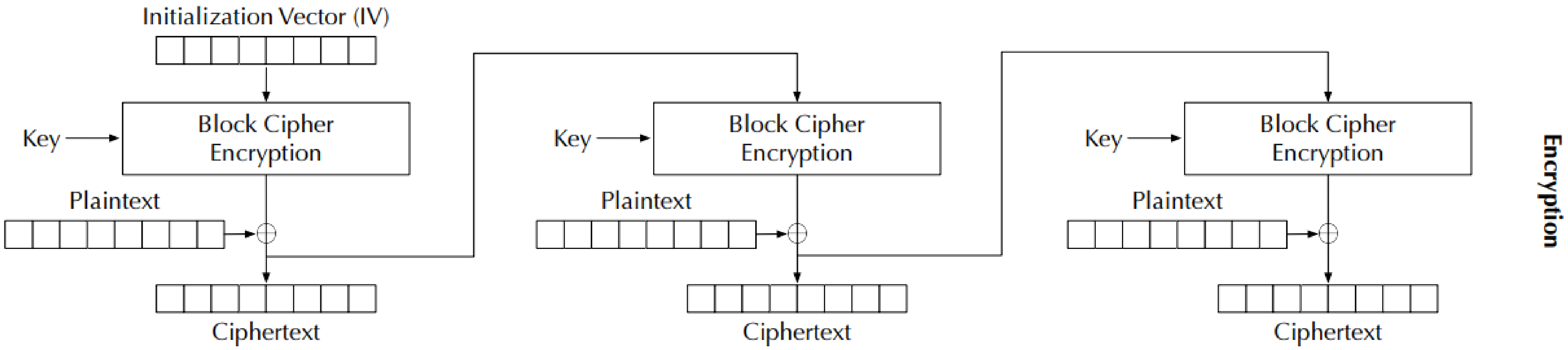
Using OpenSSL to encrypt w/ CBC

```
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?

B1	B2	B3	B4
0011	1110	0011	10

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?

B1	B2	B3	B4
0011	1110	0011	10XX

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.......
```

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

K =1

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

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

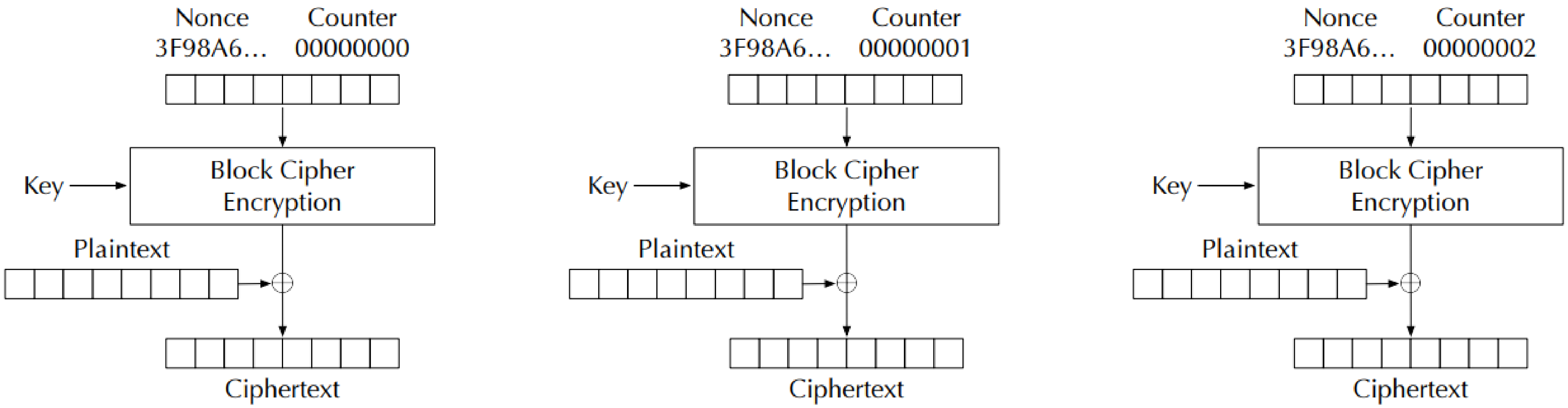
```
$ ls -ld cipher3.bin plain3_new.txt
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 cipher3.bin
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 plain3_new.txt
```

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

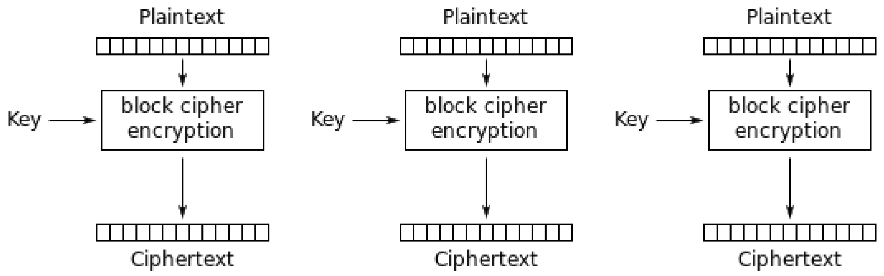
- Size of plaintext (plain3.txt) is **16 bytes**
- Size of decryption output (plaint3_new.txt) is **32 bytes** → *a new, full block is added as the padding*
- *In PKCS#5, if the input length is already an exact multiple of the block size B, then B bytes of value B are added as the padding.*

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)**

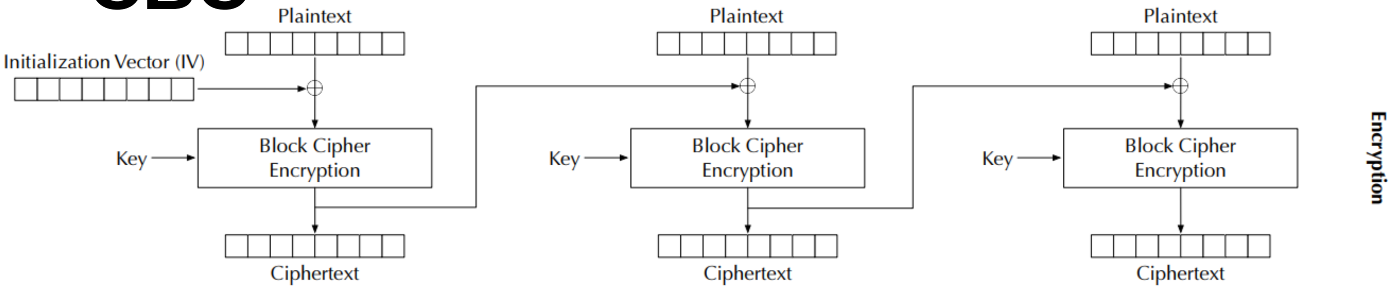


Modes of Encryption

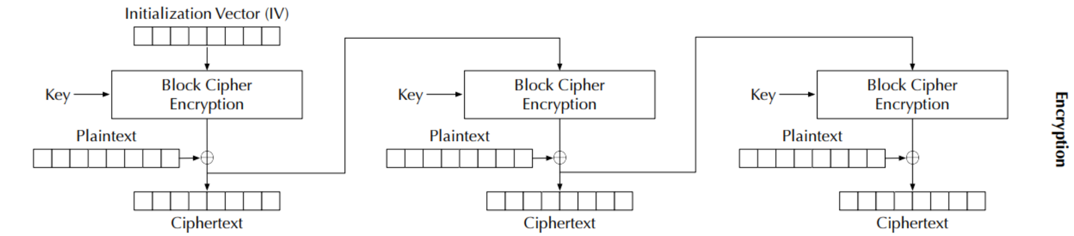


Electronic Codebook (ECB) mode encryption

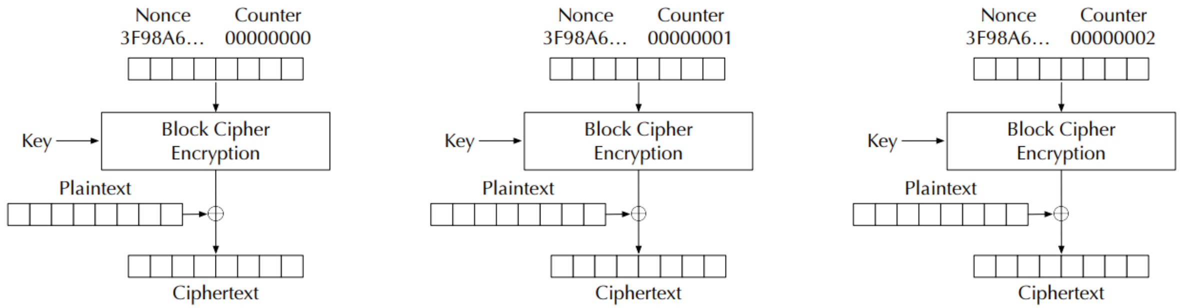
CBC



Cipher Feedback (CFB) Mode



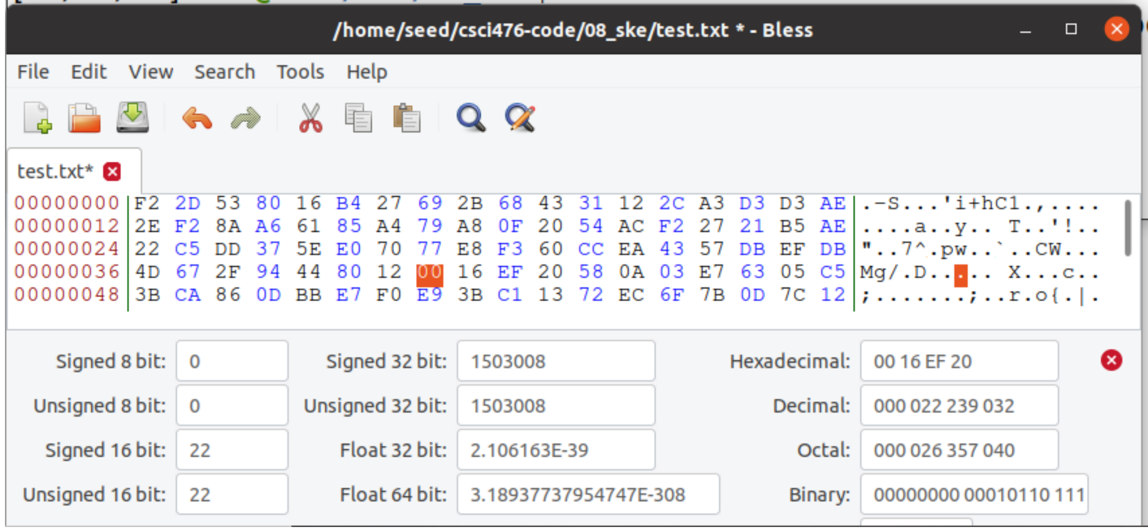
Counter(CTR) Mode



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
                                                      dule"
```



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

```
[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, dis@#�f0w�^�+��wDer 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.
```

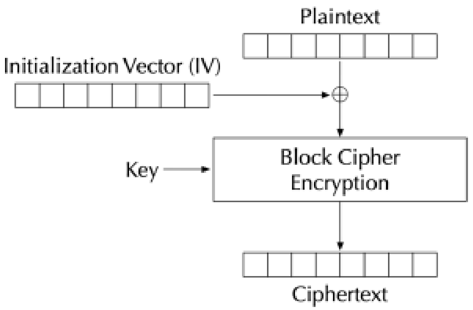
When decrypting the ciphertext, we can see

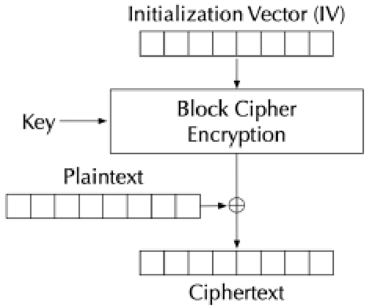
Initialization Vectors and Common Mistakes

- 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:

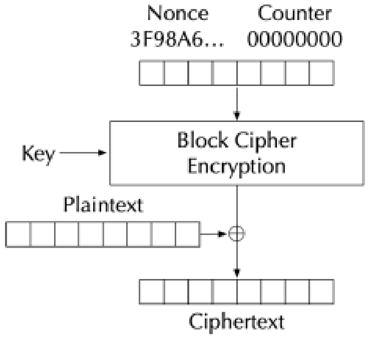
CBC

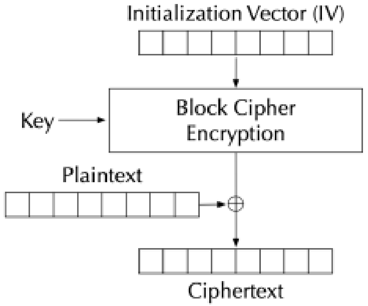




CFB

CTR





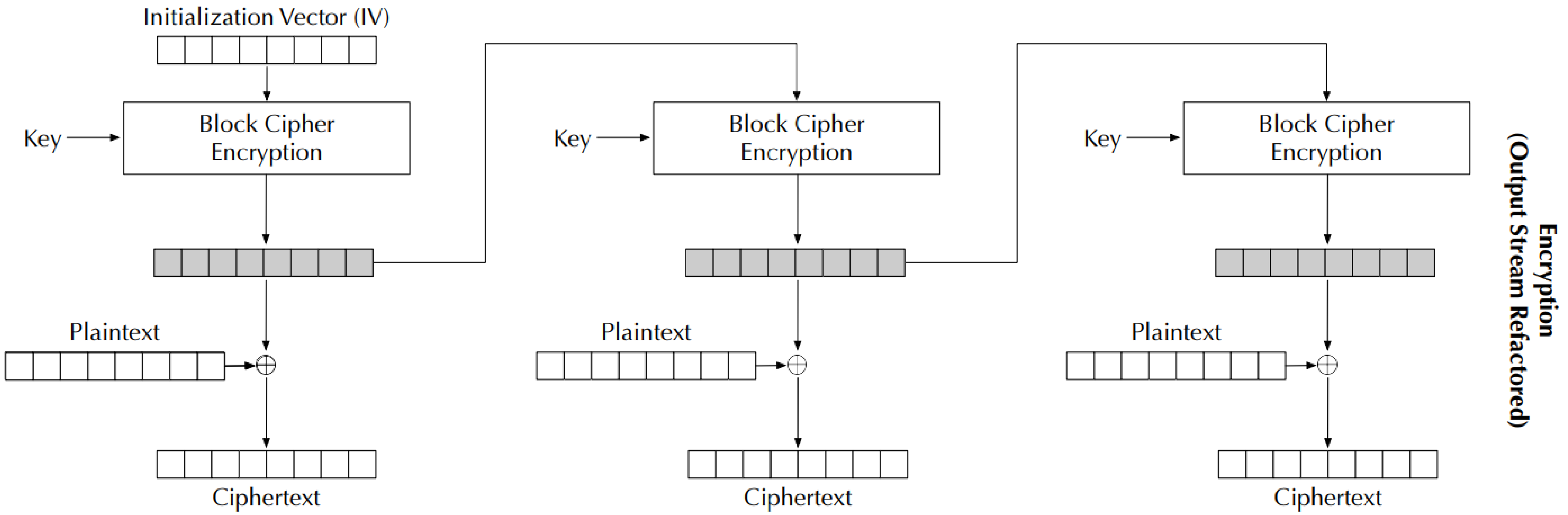
OFB

IV should not be <u>**reused**</u>...

Scenario:

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

Attacker Goal: Decrypt other plaintexts

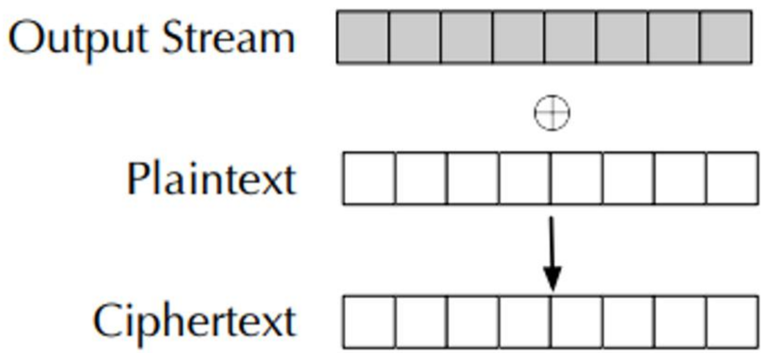


Chosen Plaintext Attack:

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?



??

110011

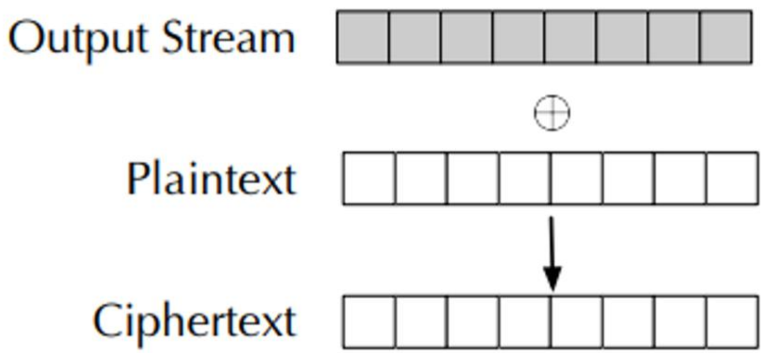
1010101

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?



??

110011

1010101

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

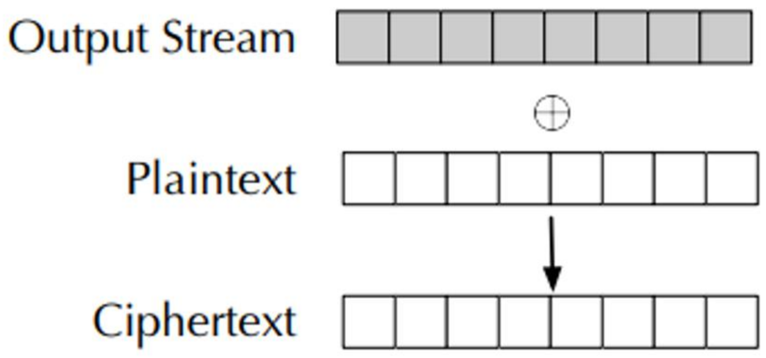
$$\begin{array}{r} 110011 \\ \oplus\ 101010 \\ \hline 011001 \end{array}$$

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?



011001
110011
101010

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)