CSCI 476: Computer Security

Secret Key Encryption/Symmetric Cryptography

Reese Pearsall

Fall 2022

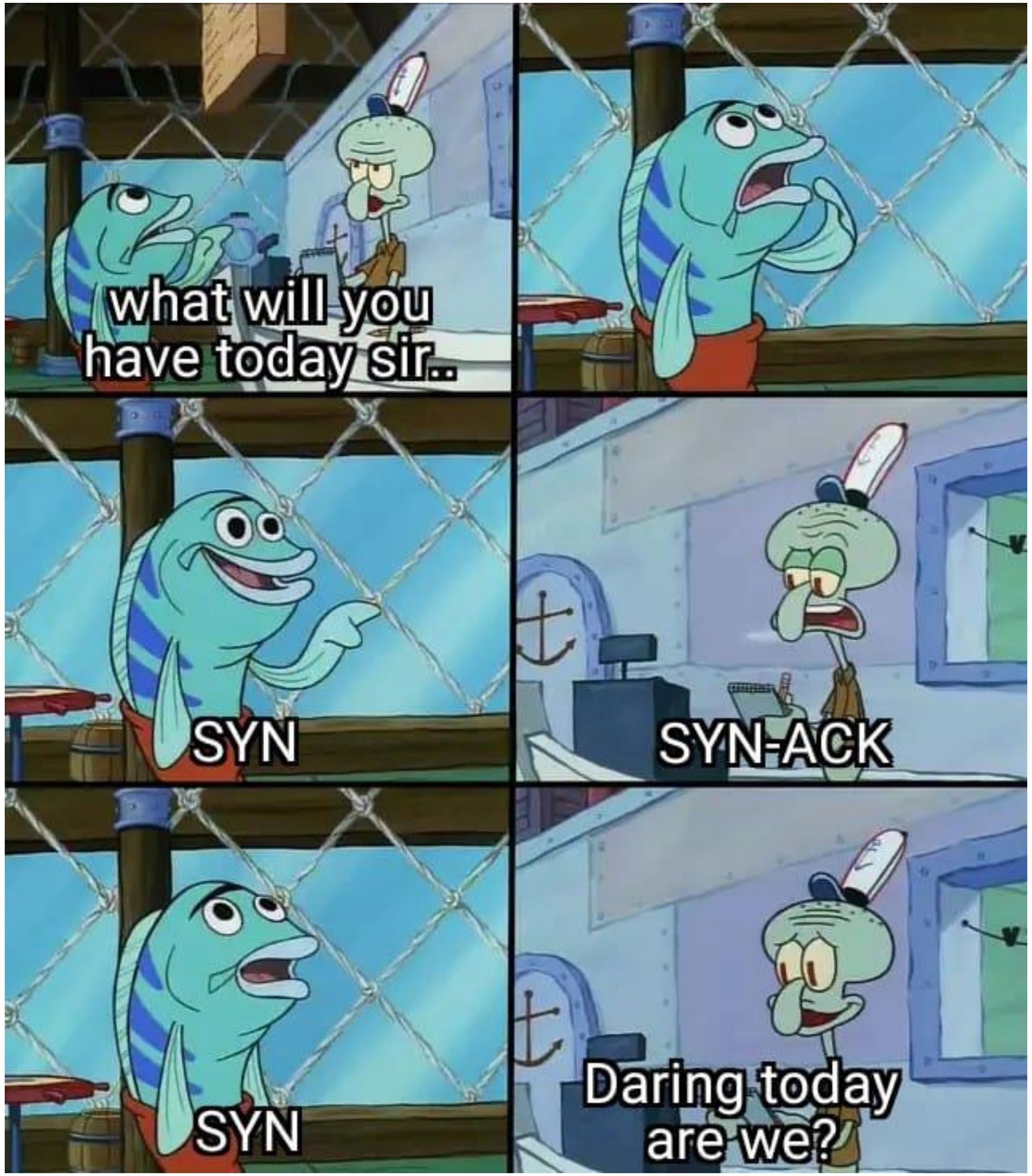
Announcement

Lab 7 (TCP attacks) Due **Thursday** November 10^{th}

→ Sounds like we have some issues with the C program

No class on Tuesday next week (11/8) (go vote)

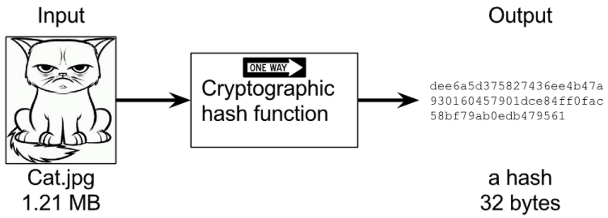
Grading rubric now on project instructions webpage



Crypto Roadmap

- Secret-Key Encryption (a.k.a Symmetric Key Encryption)

- Cryptographic Hash Functions



- Public-Key Encryption (a.k.a Asymmetric Key Encryption)

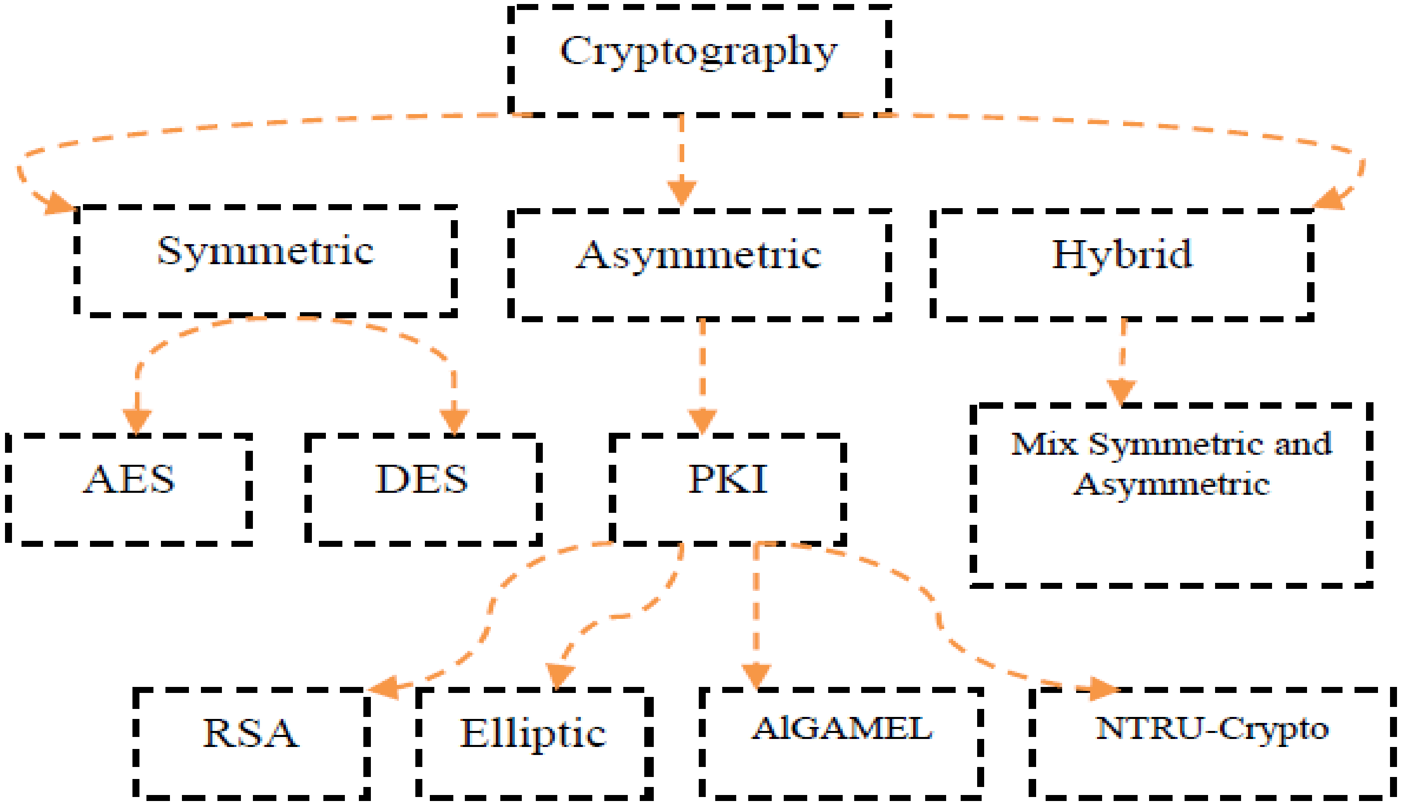
Sorry to the people that are in CSCI 476, CSCI 466 *and* CSCI 460

Information Security

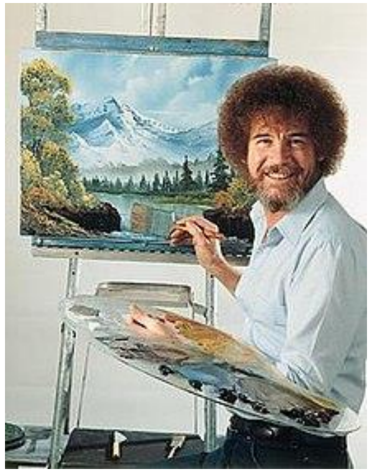
- The protection of information and information systems

Cryptography is the practice and study of techniques for securing communications and data in the presence of adversaries

There are many types of encryption



Bob

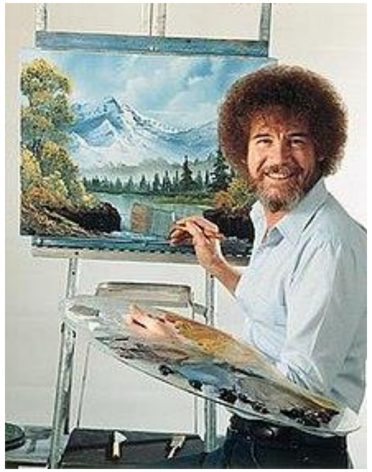


“Hi Alice, my address is
123 Painting Avenue.
Please stop by at 6:00”

Over a wire, wirelessly, via a Pidgeon etc

Alice

Bob



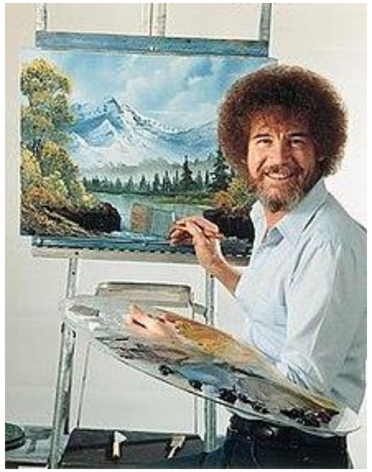
“Hi Alice, my address is 123 Painting Avenue. Please stop by at 6:00”

Alice

Because our transmission medium is **shared**, there is a possible someone else could be eavesdropping

Eve

Bob



"Hi Alice, my address is
123 Painting Avenue.
Please stop by at 6:00"

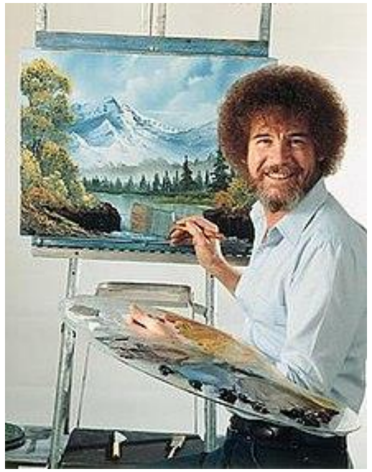
Alice

Because our transmission medium is **shared**, there is a possible someone else could be eavesdropping

Eve

Our goal is to make sure Alice can receive our message securely, and our original message cannot be intercepted

Bob



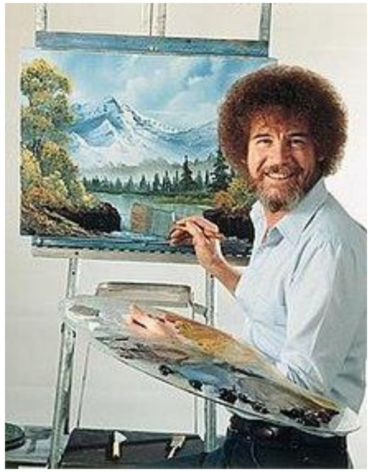
Alice

Cleartext/Plaintext

“Hi Alice, my address is 123 Painting Avenue. Please stop by at 6:00”

Eve

Bob

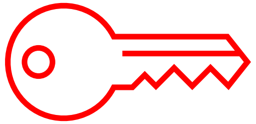


Alice

Eve

Cleartext/Plaintext

"Hi Alice, my address is
123 Painting Avenue.
Please stop by at 6:00"

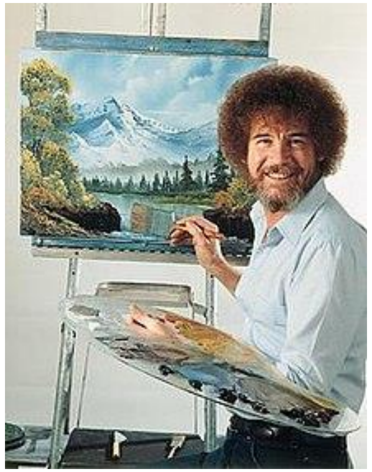


Bob **encrypts** his message with a **key**

MuYGoP5LiTTGPVX6U/r2VTpxPSqT
Fmy5nsoFWURThKMhHk/7tbjYsS2EJ
917q7megTAcV+V4ZMU4HjJjiW2DC
BroxvJ0V3ZYDgZ8B9lUvGUmdiRMH
25Xkf7QrhAGR3FF

The result is a **ciphertext**

Bob



MuYGoP5LiTTGPVX6U/r2VTpxPSqTFmy5nsoF
WURThKMhHk/7tbjYsS2EJ917q7megTAcV+V4Z
MU4HjJjiW2DCBroxvJ0V3ZYDgZ8B9lUvGUmdiR
MH25Xkf7QrhAGR3FF

Alice

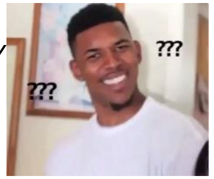
Cleartext/Plaintext

“Hi Alice, my address is 123 Painting Avenue. Please stop by at 6:00”

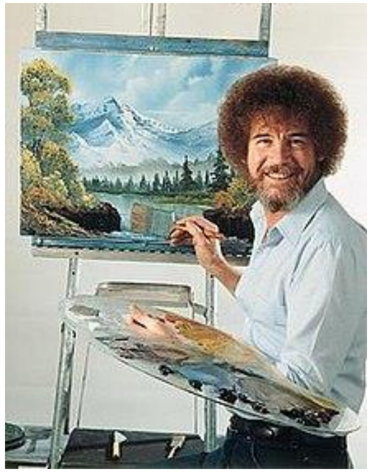
Eve

If Eve intercepts our ciphertext, she can't do very much with it

MuYGoP5LiTTGPVX6U/r2VTpx
PSqTFmy5nsoFWURThKMhHk
/7tbjYsS2EJ917q7megTAcV+V
4ZMU4HjJjiW2DCBroxvJ0V3ZY
DgZ8B9lUvGUmdiRMH25Xkf7
QrhAGR3FF



Bob



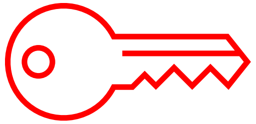
Cleartext/Plaintext

"Hi Alice, my address is 123 Painting Avenue. Please stop by at 6:00"

Eve

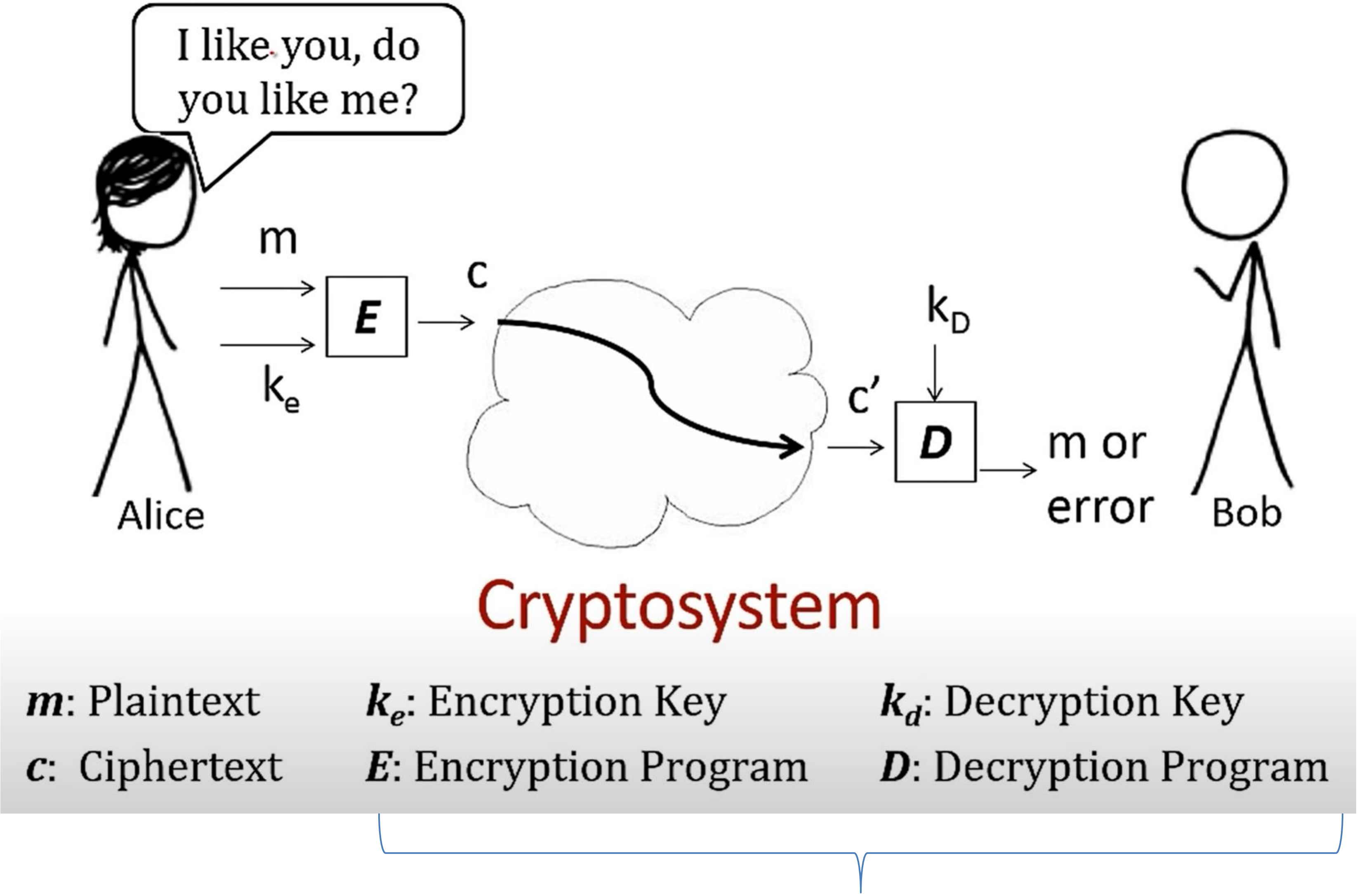
Alice

MuYGoP5LiTTGPVX6U/r2
VTpxPSqTFmy5nsoFWUR
ThKMhHk/7tbjYsS2EJ917
q7megTAcV+V4ZMU4HjJji
W2DCBroxvJ0V3ZYDgZ8
B9lUvGUmdiRMH25Xkf7
QrhAGR3FF



Alice receives the ciphertext, and then uses the **same key** that bob used, and then **decrypts** the ciphertext

"Hi Alice, my address is 123 Painting Avenue. Please stop by at 6:00"



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

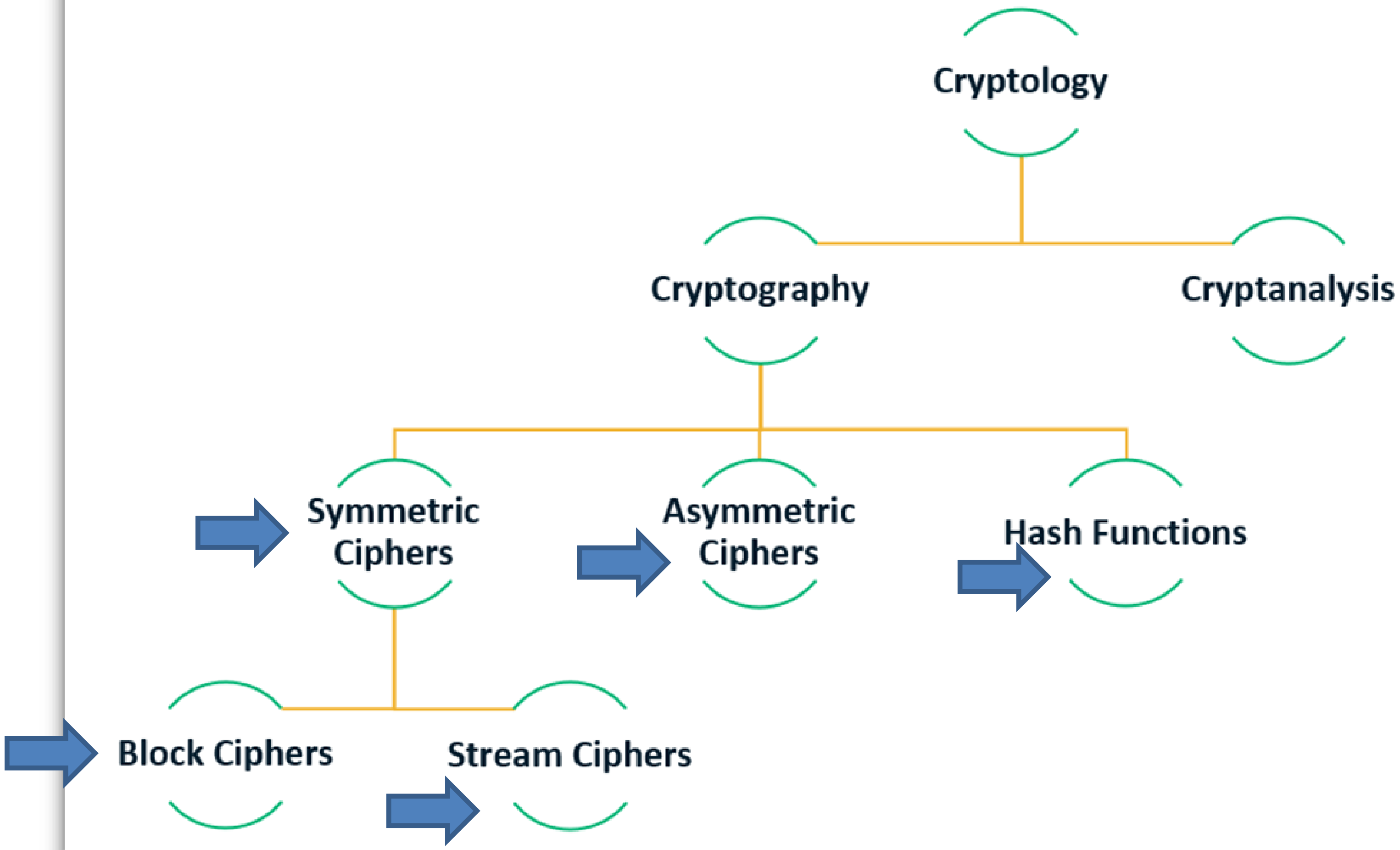
Secure cryptography is the foundation for our secure communications in the cyber world (HTTPS, SSH, etc)

The encryption algorithms are typically rooted in **very difficult problems** in computing (ie there does not exist a program that can efficiently break RSA **YET**)

There are very intense proofs and prove the secureness of the encryption procedures we use today

Never try to roll out your own cryptography scheme, and never use the built-in RNG for secure communications (import random)

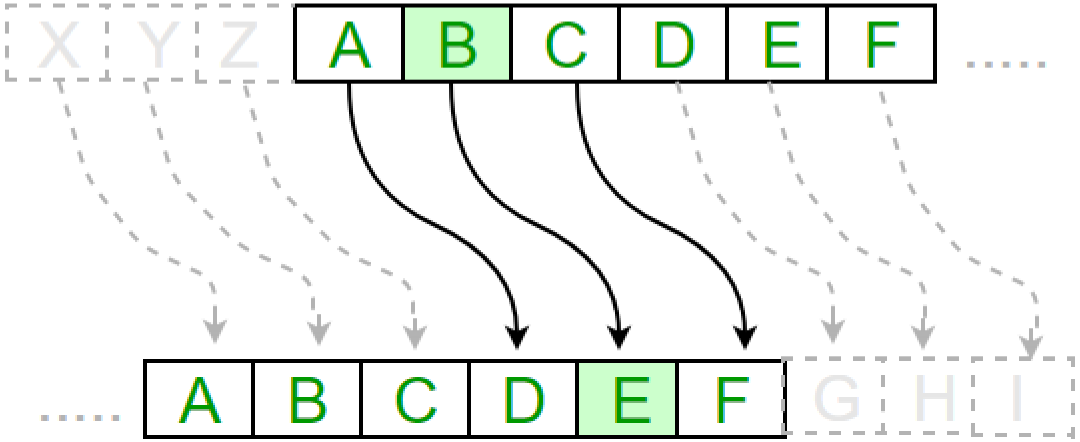
Cryptology vs Cryptography



Early cryptography techniques

Caesar Cipher- letters in the plaintext will be replaced by some fixed number of positions downs in the alphabet.

Shift 3



plaintext

hello there world my name is reese

ciphertext

khoor wkhuh zruog pb qdph lv uhhvh

Nifty, but we have the technology to brute force 26 possible shifts

Substitution Cipher

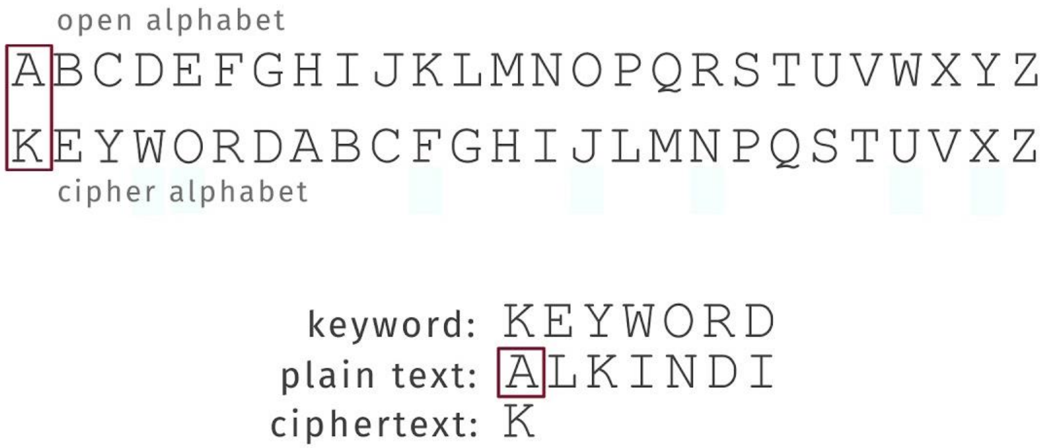
Letters in plaintext are substituted by another letter

E→X
R → Z

REESE = ZXXSX

Monolithic Substitution Cipher – Same “rules” are applied throughout the entire plaintext

Polyalphabetic Substitution Cipher – different “rules” are applied throughout the plaintext



Here is a ciphertext (cipher.txt)

```
ydq ufyiqoobxrk lrcqx yqoy fo r kwgyfoyrbq rqxepfc crlrcfyt yqoy ydry lxebxqoofvqgt bqyo kexq
mfuufcwgy ro fy ceiyfiwqo. ydq ysqiyt kqyqx lrcqx yqoy sfgg pqbfi fi ydfxyt oqceimo. gfiq wl ry ydq
oyrxy. ydq xwiifib olqqm oyrxyo ogesgt, pwy bqyo uroyqx qrcd kfiwyq ruyqx tew dqrx ydfo ofbirg pqql r
ofibgq grl odewgm pq ceklgqyqm qrcd yfkq tew dqrx ydfo oewim. [mfib] xqkqkpqx ye xwi fi r oyxrfbdy
gfiq, rim xwi ro geib ro leoofpgq. ydq oqceim yfkq tew urfg ye ceklgqyq r grl pquexq ydq oewim, tewx
yqoy fo evqx. ydq yqoy sfgg pqbfi ei ydq sexm oyrxy. ei tewx krxj, bqy xqrmt, oyrxy.
```

Suppose we know that that this message is an english message encrypted with a monolithic substitution cipher

Can we crack this?

Here is a ciphertext (cipher.txt)

```
ydq ufyiqoobxrk lrcqx yqoy fo r kwgyfoyrbq rqxepfc crlrcfyt yqoy ydry lxebxqoofvqgt bqyo kexq
mfuufcwgy ro fy ceiyfiwqo. ydq ysqiyt kqyqx lrcqx yqoy sfgg pqbfi fi ydfxyt oqceimo. gfiq wl ry ydq
oyrxy. ydq xwiifib olqqm oyrxyo ogesgt, pwy bqyo uroyqx qrcd kfiwyq ruyqx tew dqrx ydfo ofbirg pqql r
ofibgq grl odewgm pq ceklgqyqm qrcd yfkq tew dqrx ydfo oewim. [mfib] xqkqkpqx ye xwi fi r oyxrfbdy
gfiq, rim xwi ro geib ro leoofpgq. ydq oqceim yfkq tew urfg ye ceklgqyq r grl pquexq ydq oewim, tewx
yqoy fo evqx. ydq yqoy sfgg pqbfi ei ydq sexm oyrxy. ei tewx krxj, bqy xqrmt, oyrxy.
```

Frequency Analysis leverages the fact that in any given written language, certain letters and combinations occur more frequently than others

In English, T, A , I, and O are the most common letters, so it is likely the letters that appear the most frequently in our ciphertext are one of those

```
ydq ufyiqoobxrk lrcqx yqoy fo r kwgyfoyrbq rqxepfc crlrcfyt yqoy ydry lxebxqoofvqgt bqyo kexq
mfuufcwgy ro fy ceiyfiwqo. ydq ysqiyt kqyqx lrcqx yqoy sfgg pqbfi fi ydfxyt oqceimo. gfiq wl ry ydq
oyrxy. ydq xwiifib olqqm oyrxyo ogesgt, pwy bqyo uroyqx qrcd kfiwyq ruyqx tew dqrx ydfo ofbirg pqql r
ofibgq grl odewgm pq ceklgqyqm qrcd yfkq tew dqrx ydfo oewim. [mfib] xqkqkpqx ye xwi fi r oyxrfbdy
gfiq, rim xwi ro geib ro leoofpgq. ydq oqceim yfkq tew urfg ye ceklgqyq r grl pquexq ydq oewim, tewx
yqoy fo evqx. ydq yqoy sfgg pqbfi ei ydq sexm oyrxy. ei tewx krxj, bqy xqrmt, oyrxy.
```

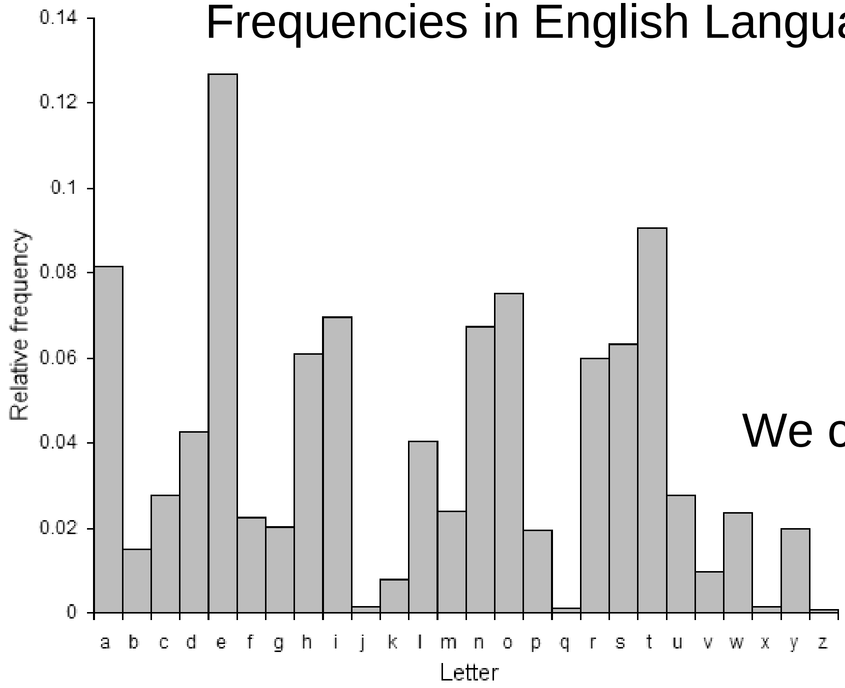
Here is a ciphertext (cipher.txt)

We can write a program that counts the frequency of characters (**1-gram**) and frequency of character pairs (**2-gram**)

```
[11/03/22]seed@VM:~/encyption_lecture$ ./freq.py < ciphertext.txt
-------------------------------------
1-gram (top 20):
q: 61
y: 58
o: 39
r: 34
f: 32
x: 30
i: 27
e: 26
g: 21
d: 18
w: 17
b: 14
c: 13
k: 12
l: 12
m: 12
t: 11
p: 9
.: 8
u: 7
```

```
2-gram (top 20):
yd: 12
oy: 12
yq: 11
fi: 11
dq: 10
qo: 8
qx: 8
ew: 8
rx: 7
qy: 6
ei: 6
pq: 6
rc: 5
fo: 5
yr: 5
xq: 5
ce: 5
xy: 5
im: 5
wi: 5
```

Frequencies in English Language



We can start making guesses!

Most common bigrams (in order)

th, he, in, en, nt, re, er, an, ti, es, on, at, se, nd, or, ar, al, te, co, de, to, ra, et, ed, it, sa, em, ro.

```
ydq ufyiqoobxrk lrcqx yqoy fo r kwgyfoyrbq rqxepfc crlrcfyt yqoy ydry lxebxqoofvqgt bqyo kexq
mfuufcwgy ro fy ceiyfiwqo. ydq ysqiyt kqyqx lrcqx yqoy sfgg pqbfi fi ydfxyt oqceimo. gfiq wl ry ydq
oyrxy. ydq xwiifib olqqm oyrxyo ogesgt, pwy bqyo uroyqx qrcd kfiwyq ruyqx tew dqrx ydfo ofbirg pqql r
ofibgq grl odewgm pq ceklgqyqm qrcd yfkq tew dqrx ydfo oewim. [mfib] xqkqkpqx ye xwi fi r oyxrfbdy
gfiq, rim xwi ro geib ro leoofpgq. ydq oqceim yfkq tew urfg ye ceklgqyq r grl pquexq ydq oewim, tewx
yqoy fo evqx. ydq yqoy sfgg pqbfi ei ydq sexm oyrxy. ei tewx krxj, bqy xqrmt, oyrxy.
```

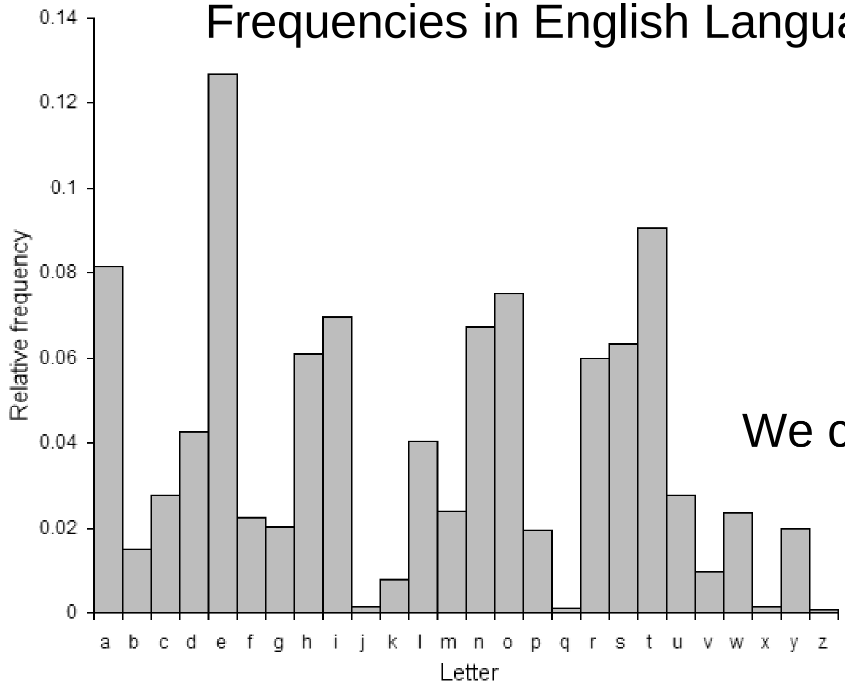
Here is a ciphertext (cipher.txt)

We can write a program that counts the frequency of characters (**1-gram**) and frequency of character pairs (**2-gram**)

```
[11/03/22]seed@VM:~/encyption_lecture$ ./freq.py < ciphertext.txt
-------------------------------------
1-gram (top 20):
q: 61
y: 58
o: 39
r: 34
f: 32
x: 30
i: 27
e: 26
g: 21
d: 18
w: 17
b: 14
c: 13
k: 12
l: 12
m: 12
t: 11
p: 9
.: 8
u: 7
```

```
2-gram (top 20):
yd: 12
oy: 12
yq: 11
fi: 11
dq: 10
qo: 8
qx: 8
ew: 8
rx: 7
qy: 6
ei: 6
pq: 6
rc: 5
fo: 5
yr: 5
xq: 5
ce: 5
xy: 5
im: 5
wi: 5
```

Frequencies in English Language



We can start making guesses!

Most common bigrams (in order)

th, he, in, en, nt, re, er, an, ti, es, on, at, se, nd, or, ar, al, te, co, de, to, ra, et, ed, it, sa, em, ro.

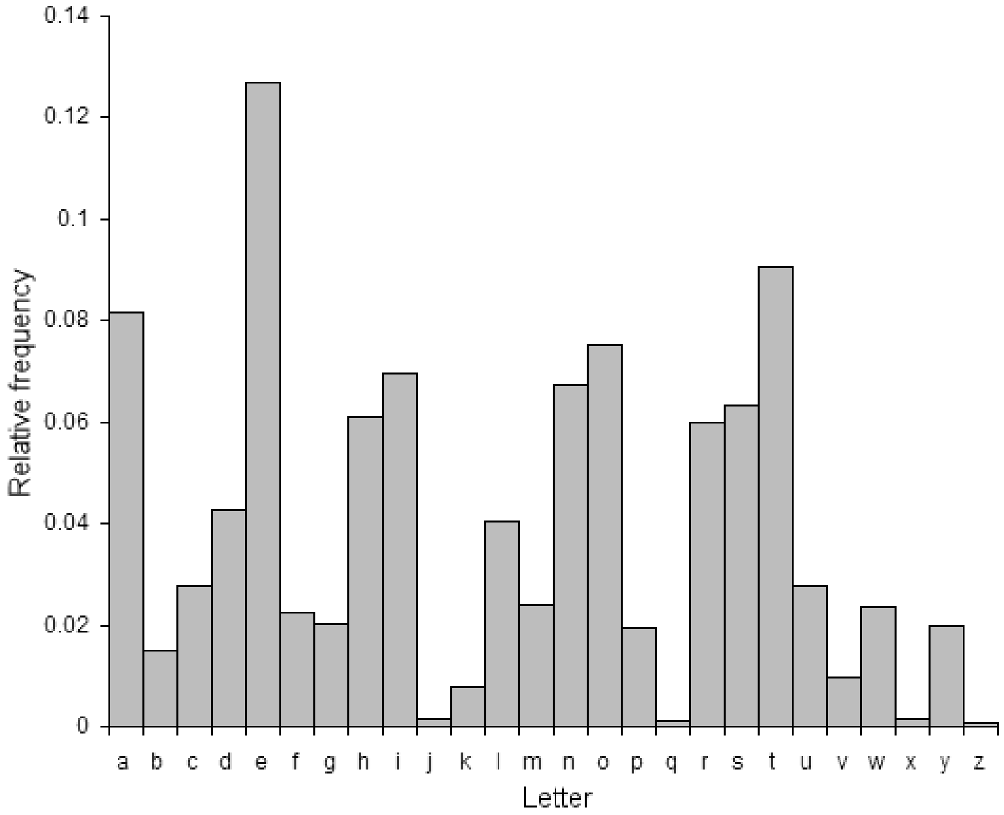
Listing 24.2: Bigram and trigram frequencies

Bigram frequency in English

TH : 2.71	EN : 1.13	NG : 0.89
HE : 2.33	AT : 1.12	AL : 0.88
IN : 2.03	ED : 1.08	IT : 0.88
ER : 1.78	ND : 1.07	AS : 0.87
AN : 1.61	TO : 1.07	IS : 0.86
RE : 1.41	OR : 1.06	HA : 0.83
ES : 1.32	EA : 1.00	ET : 0.76
ON : 1.32	TI : 0.99	SE : 0.73
ST : 1.25	AR : 0.98	OU : 0.72
NT : 1.17	TE : 0.98	OF : 0.71

Trigram frequency in English

THE : 1.81	ERE : 0.31	HES : 0.24
AND : 0.73	TIO : 0.31	VER : 0.24
ING : 0.72	TER : 0.30	HIS : 0.24
ENT : 0.42	EST : 0.28	OFT : 0.22
ION : 0.42	ERS : 0.28	ITH : 0.21
HER : 0.36	ATI : 0.26	FTH : 0.21
FOR : 0.34	HAT : 0.26	STH : 0.21
THA : 0.33	ATE : 0.25	OTH : 0.21
NTH : 0.33	ALL : 0.25	RES : 0.21
INT : 0.32	ETH : 0.24	ONT : 0.20



```
[11/03/22]seed@VM:~/encyption_lecture$ tr 'y' 't' < ciphertext.txt > output.txt
```

Translate ciphertext.txt, and replace all **y** with **t**

```
[11/03/22]seed@VM:~/encyption_lecture$ cat output.txt
tdq uftiqoobxrk lrcqx tqot fo r kwgtfotrbq rqxepfc crlrcftt tqot tdrt lxebxqoofvqgt bqto kexq mfuufcwgt ro ft ceitfiwqo. tdq tsqitt kqtqx lrcqx tqot sfgg pqbfi fi tdfxtt
 oqceimo. gfiq wl rt tdq otrxt. tdq xwiifib olqqm otrxto ogesgt, pwt bqto urotqx qrcd kfiwtq rutqx tew dqrx tdfo ofbirg pqql r ofibgq grl odewgm pq ceklgqtqm qrcd tfkq t
ew dqrx tdfo oewim. [mfib] xqkqkpqx te xwi fi r otxrfbdt gfiq, rim xwi ro geib ro leoofpgq. tdq oqceim tfkq tew urfg te ceklgqtq r grl pquexq tdq oewim, tewx tqot fo evq
x. tdq tqot sfgg pqbfi ei tdq sexm otrxt. ei tewx krxj, bqt xqrmt, otrxt.
```

```
[11/03/22]seed@VM:~/encyption_lecture$ tr 'yd' 'th' < ciphertext.txt > output.txt
```

Translate ciphertext.txt, and replace all **y** with **t,** and replace all **d** with **h**

```
thq uftiqoobxrk lrcqx tqot fo r kwgtfotrbq rqxepfc crlrcftt tqot thrt lxebxqoofvqgt bqto kexq mfuufcwgt ro ft ceitfiwqo. thq tsqitt kqtqx lrcqx tqot sfgg pqbfi fi thfxtt
 oqceimo. gfiq wl rt thq otrxt. thq xwiifib olqqm otrxto ogesgt, pwt bqto urotqx qrch kfiwtq rutqx tew hqrx thfo ofbirg pqql r ofibgq grl ohewgm pq ceklgqtqm qrch tfkq t
ew hqrx thfo oewim. [mfib] xqkqkpqx te xwi fi r otxrfbht gfiq, rim xwi ro geib ro leoofpgq. thq oqceim tfkq tew urfg te ceklgqtq r grl pquexq thq oewim, tewx tqot fo evq
x. thq tqot sfgg pqbfi ei thq sexm otrxt. ei tewx krxj, bqt xqrmt, otrxt.
```

Keep adding more characters to your decryption scheme until you get the full answer ☺

Review the XOR operator:

Everything on a computer is **zeros** and **ones**

0101010101010100101111101010100
1000010111001000101010101100
10101010101111110100100101010
10100101010101100101010111010
1001010101010101010101001010
1010101010101001010101010101
0101101001010101010100101…

Hello world

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

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

1 ⊕ 0 = 1
0 ⊕ 0 = 0
1 ⊕ 1 = 0
0 ⊕ 1 = 1

Message: 0001 1010 0011

Key: ⊕ 1100 1100 0101

Ciphertext: 1101 0110 0110

How to get original message?

Review the XOR operator:

Everything on a computer is **zeros** and **ones**

0101010101010010111101010100
1000010111001000101010101100
1010101010111110100100101010
1010010101010110010101011010
1001010101010101010101001010
1010101010101001010101010101
01011010010101010100101…

Hello world → 01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100 00001010

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

1 ⊕ 0 = 1
0 ⊕ 0 = 0
1 ⊕ 1 = 0
0 ⊕ 1 = 1

Message: 0001 1010 0011
Key: ⊕ 1100 1100 0101

Ciphertext: 1101 0110 0110
⊕ 1100 1100 0101

XOR with the key again! 0001 1010 0011

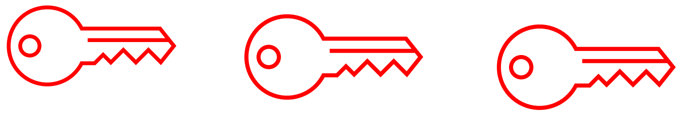
Block Cipher

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

Hello there world

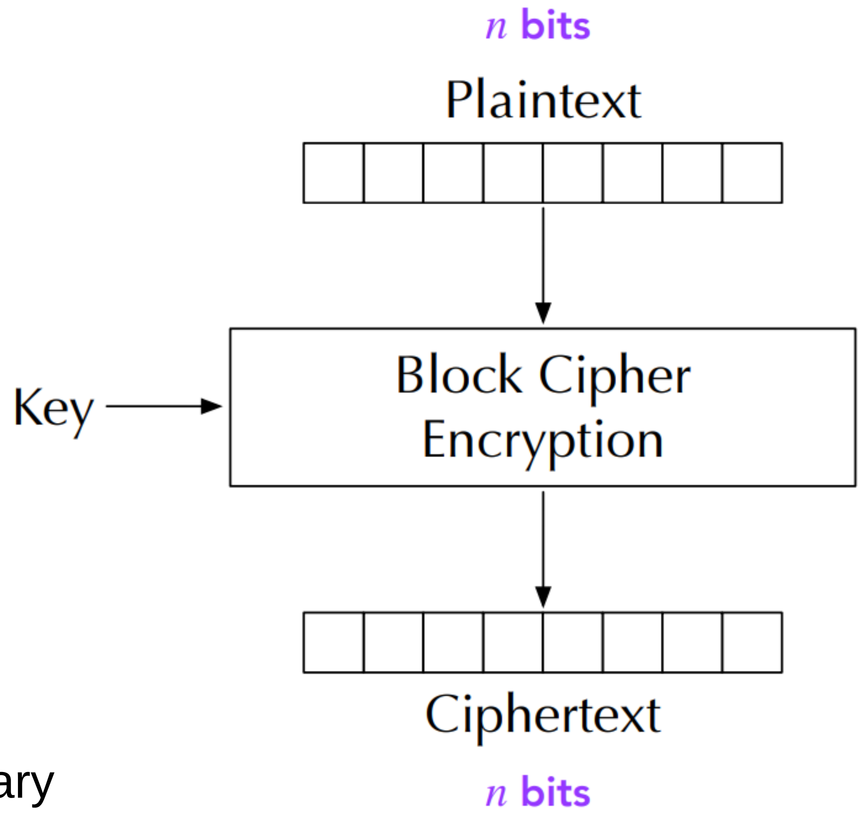
01101000	01100101	01101100
01101100	01101111	00100000
01110100	01101000	01100101
01110010	01100101	00100000
01110111	01101111	01110010
01101100	01100100	00001010

Block 1 $\oplus$ | Block 2 $\oplus$ | Block 3 $\oplus$



Ciphertext

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

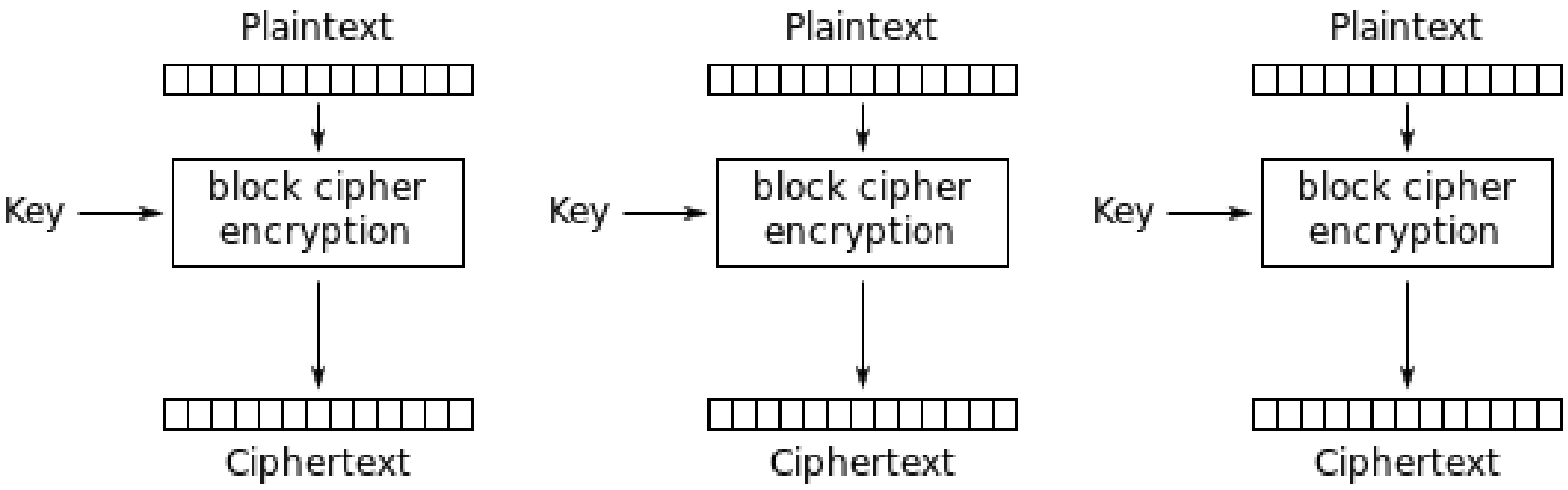
Modes of Encryption

- 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

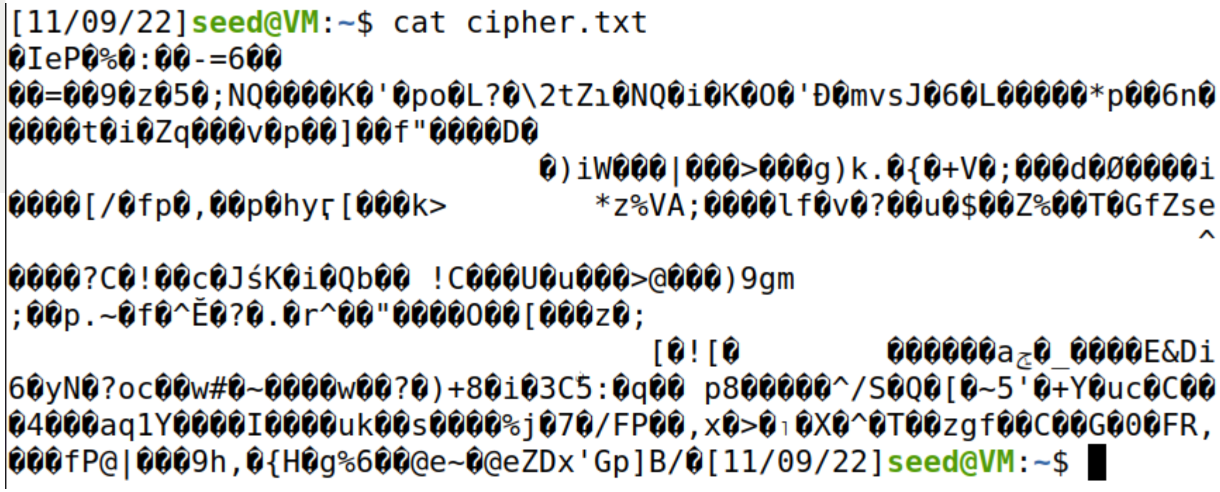
Using OpenSSL to encrypt w/ ECB

Encrypt a .txt file

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

plain.txt

```
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.
```



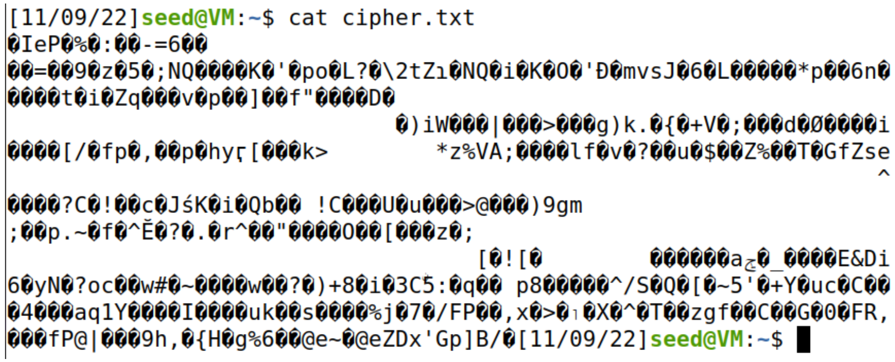
Using OpenSSL to encrypt w/ ECB

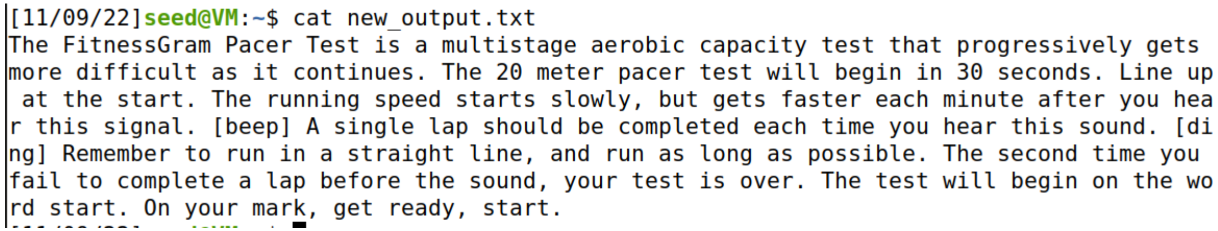
Encrypt a .txt file

```
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�?��ɟ�����
?�'�o��qj?�~A5J/;F.L/D?V�Ů/��fO�m�����M��t��HɑDr.#.�
                                                    ����i��s*����&F/���Bv�w=
                                                                          d�>��r�Q�3�i����r�z
}d��dA�����]F����3����]:�ZX�/�?h�Y�d�2W��w�5桢�<�z����r��|��2�|�U�bb�[11/09/22]seed@VM:
~$
```

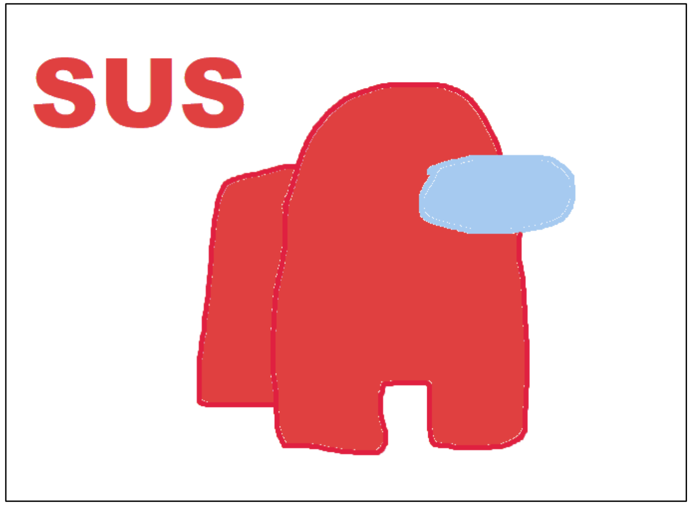
Using OpenSSL to encrypt w/ ECB

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

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



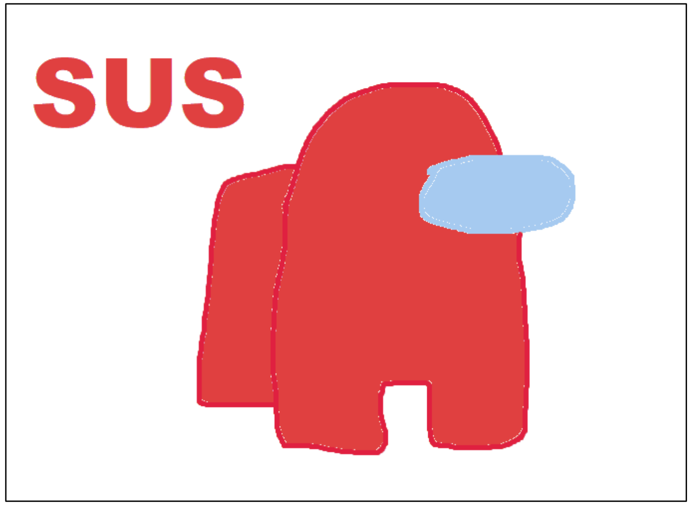
When encrypting images on the lab, make sure you use a **.bmp** image

(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

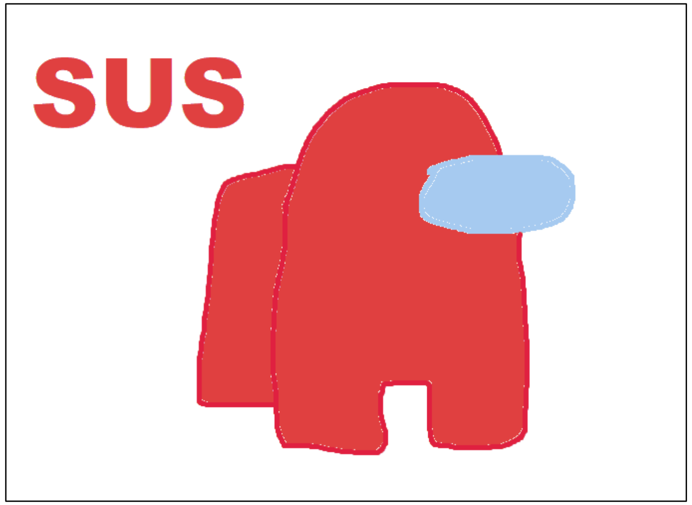
When we encrypt the image, the header will also get encrypted

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

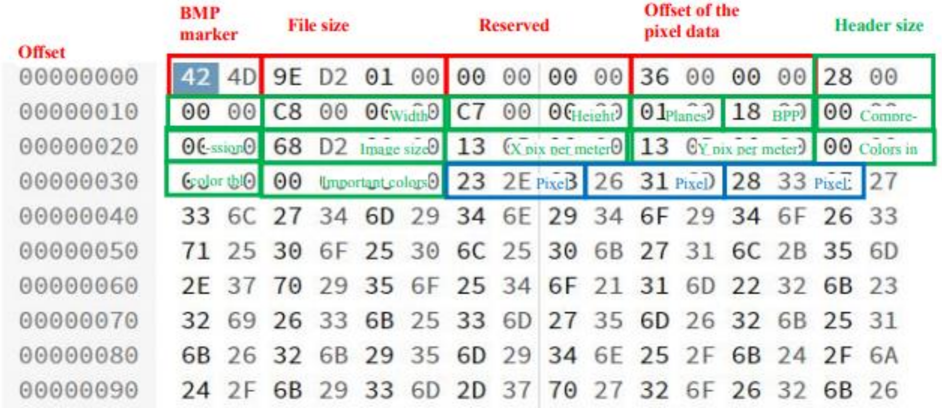


0101101010101011101011011011010101010101010101010100101

Header

1001010011010101111011100101100010011100000101101100001110011011010111011100000
0010111100100101011011101111011011110111111111001101011101100110010100100010010
0100000100000000101000110101000101001010000000110010001100000011011111011000010
1110001000001100011011110111100010110000000100000011100010010110011111000000100 1
0000000010110110110010101101101110011111101111101111001000001011001000111001101
1000101010001001010100101010111100000011011011011100011101100100010111111011010
1110100000001001100100111000011001100010100010011111010001110100101000000110001 1
1001111111001100100101001000110010111101110111111000111000111101100011101101110
1101001111111101110011001011001001110010010010110001001000001010010100011010111
1000001011000100111000101000101001001010100101111011001110100111010010100000111
1011000000101010100110010101111110111111101110001000011001010101000101110000101
1101011111111111011100001110000010001000001001111001110011011100000101000011011
1100100001100111101111011100001001001010111110011110010011101100100010011100100
1101011001111001011000000011011111110111001010011000011001101110011101010100100
0101001001000011011100000110011000111001011111011001010101000010010101011000100
0010111111010000110111011110111011100101101001000001000011110110010100000110000
0110101101110011100100110010111011001000011000010111000101010010001011011000110
0010111000111110110101110010011101010110111110001001011010001010000110101010011
1101001001111111110110110100101001101100101010100111110001011101100101111111011
0110110000010101100111101100011100011111100010001010110111000011010100000011010
1011101010111101011101111110010111111110001011111001000011011110101110111100111
0011000000011111000101110111010000001010101010101011110000111001011000100110101
1010000100110000000101010110110110111010010000100001000100111111111110101000110 1
0111101001101101111110110111110000110111101110110100011011101100110011110100011 0
1111101100111001001010101110011011111111001000111101000001110001010110011000110
1011001110011011111011000101011101101010000111011000111111000001110110000101001
1100110001001111100010011000100110011101111111100010101110010001110110010111010 1
1000101011011101100101010000100000111000100101010111000000111000010000101011110
0011000111010000111111111000101111100011011011001011101011000011110010111001101
0001000110001011000010100011101111100101011010110100001100010110100100101110001
1110111011001100101110111101110111110001101001001110100001010110100101110100101
10000011001011101010001000010010111000101111110011 1

Body of the image

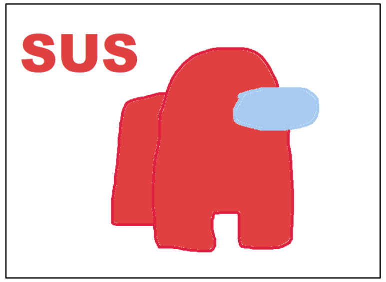


Fact: The first 54 bytes of a BMP file will be the header

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



01011010101010111010110110110101010101010101010100101

1001010011010101111011100101100010011100000101101100001110011011010111011100000
0010111100100101011011101111011011110111111111001101011101100110010100100010010
0100000100000000101000110101000101001010000001100100011000000110111110110000010
1110001000001100011011110111100010110000001000000111000100101100111110000001001
0000000010110110110010101101101110011111101111101111001000001011001000111001101
1000101010001001010100101010111100000011011011011100011101100100010111111011010
1110100000010011001001110000110011000101000100111110100011101001010000001100011
1001111111001100100101001000110010111101110111111000111000111101100011101101110
1101001111111101110011001011001001110010010010110001001000001010010100011010111
1000001011000100111000101000101001001010100101111011001110100111010010100000111
1011000000101010100110010101111110111111101110001000011001010101000101110000101
1101011111111111011100001110000010001000001001111001110011011100000101000011011
1100100001100111101111011100001001001010111111001111001001110110010001001110010 0
11010110011110010110000000110111111110111001010011000011001101110011101010100100
01010010010000110111000000110011000111001011111011001010101000010010101011000100
00101111110100001101110111101110111001011010010000010000111101100101000000110000
01101011011100111001001100101110110010000110000101110001010100100010110110001 10
00101110001111101101011100100111010101101111110001001011010001010000110101010011
11010010011111111101101101001010011011001010101001111100010111011001011111111011
01101100000101011001111011000111000111111000100010101101110000110101000000011010
10111010101111010111011111100101111111100010111110010000110111101011101111100111
00110000000111110001011101110100000001010101010101011110000111001011000100110101
10100001001100000001010101101101101110100100001000010001001111111111010100011 01
01111010011011011111011011111000011011110111011010001101110110011001111010001 10
1111101100111001001010101011100110111111111001000111101000000111000101011001100 0110
1011001110011011111101100010101110110101000011101100011111100000011101100001010 01
11001100010011111000100110001001100111011111111000101011100100011101100101110101
10001010110111011001010100001000001110001001010101110000000111000010000101011110
00110001110100001111111110001011111000110110110010111010110000111100101110011 01
00010001100010110000101000111011111001010110101101000011000101101001001011 10001
11101110110011001011101111011101111100011010010011101000010101101001011101001 01
10000011001011101010001000010010111000101111111100111

enc.bmp

1000110010010111100001110110000110000111000111110110100000110000100111010001101
0011001010000001010010000010110000110100001111100101111010111010110000110110111
1000011111111010000001010000010110101000001011000111010101000100110100111000100 0
0100010001101100111011001111001101010011100111010010011101110000001010001100001
1010010011100000110000001010001100111101010001101011010011110101010001110010100
1001010001010010010001001010000110111100010110111010101011100011100011110010111
1000011011000011111001001101010100110011100100001101100000000000111110101011001 1
1101010110011111011100010000000011100000000101000100101010100011101001101011101 00
1111000110010000001111101111100010101101011110101101010000010011000101001100101 1
0001011101101001110001001010111110010110101011001111101000100011010011101100000
0111100001011011001001100100011110110110011000100111110100001001011010101110000
0000000100001111110011111110111010111000110010011011110111010101001001100000111
0100000010100100010001101101011101011011100000010110001001101011101001101110011 0
1110101011111101000000100000100011110110111011101000110100000001011100001111011 10
11100110001011000000100111111111101110001101111011111101101100010010100000100101
10011001100110010111101001011111011111100101111000010101000000000110100010100100
11011101100011011001111100100010110001011111110100100110011011001010100000111 00
00100100101111100100111001111101011010110011111101011010001101000101001110001 0
00011101100010000011100110111111100110000010100101000110110000111111111100000 1
11101010011011011001101111010000101001010001000111000101000000100001110110010000
01001000000101000110111101101001011010000001001110111010000100010010111100100010 0
0110111110010100110001011001011110000111111011100101001011010011101011011110011
0110110000100100100010000010010100100000000100011111000010010110110101100101011
1010101000101101010011010000001100000111101101100000111010010100011111100011000 1
0110100110000000000100000111000010100101000010110000011111111111001000010111100 0
1001100000010101110000110110110110101001100010111011011101110011100011000010010
1100001010100111111011001000111010101100111111011110011110001010111110111010000
1111111111010101001011001011010101000100001110100011101101100001010001000001111
1101100100101001110010011101001101101100010110000111000001011110000010101011010
1000111010100111111100100001001011000110001110001000100001000011110011101000111
0010001100101000011000011111001111100111010110100100010101111001101101101010111
0100010110101110100111110011100000000000111011110111001111100111000000000011101 1
1101110011111001110000000000011101111011100111110011100000000000111011110111001 11
1100111000000000001110111101 1

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
```

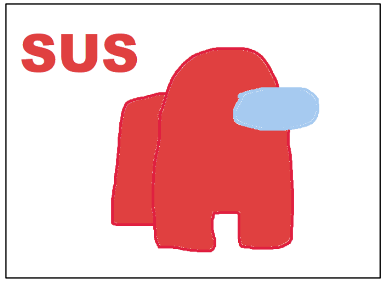
Take the first 54 bytes of the original image (header)

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

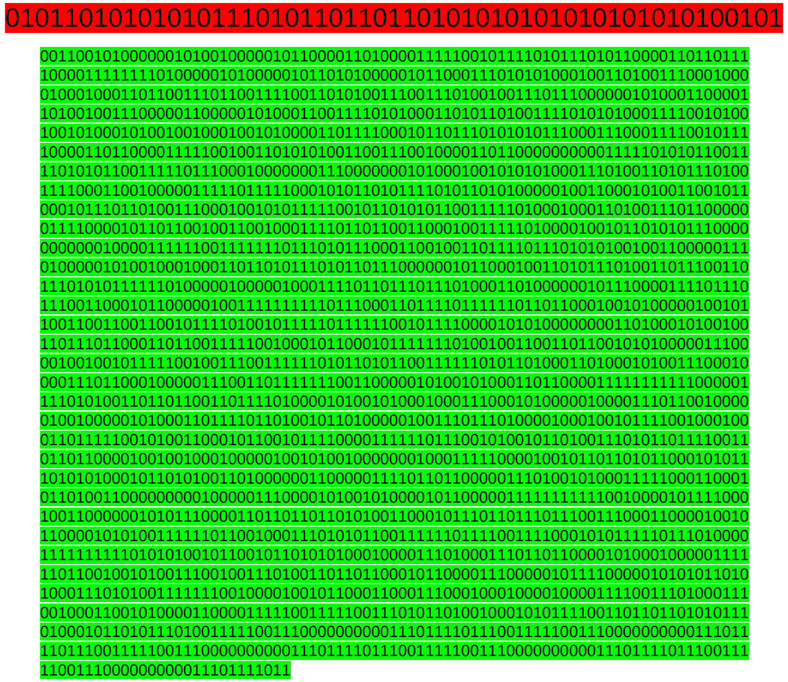
Using OpenSSL to encrypt w/ ECB

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

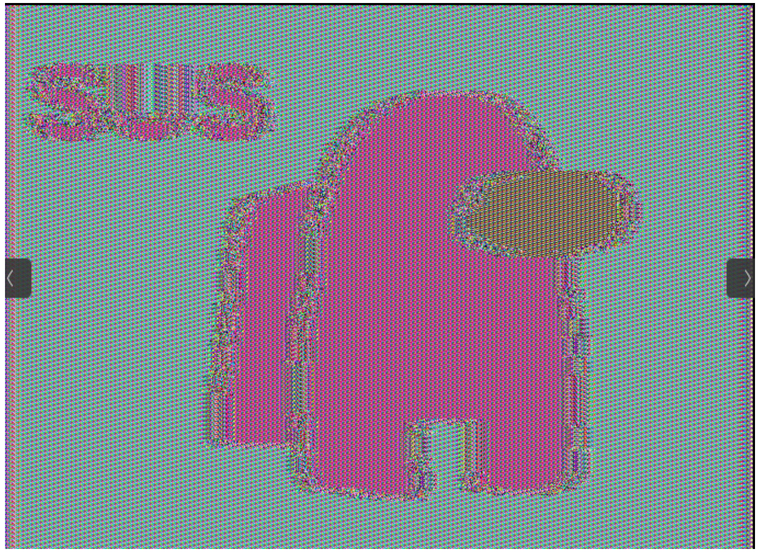
sus.bmp



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
```

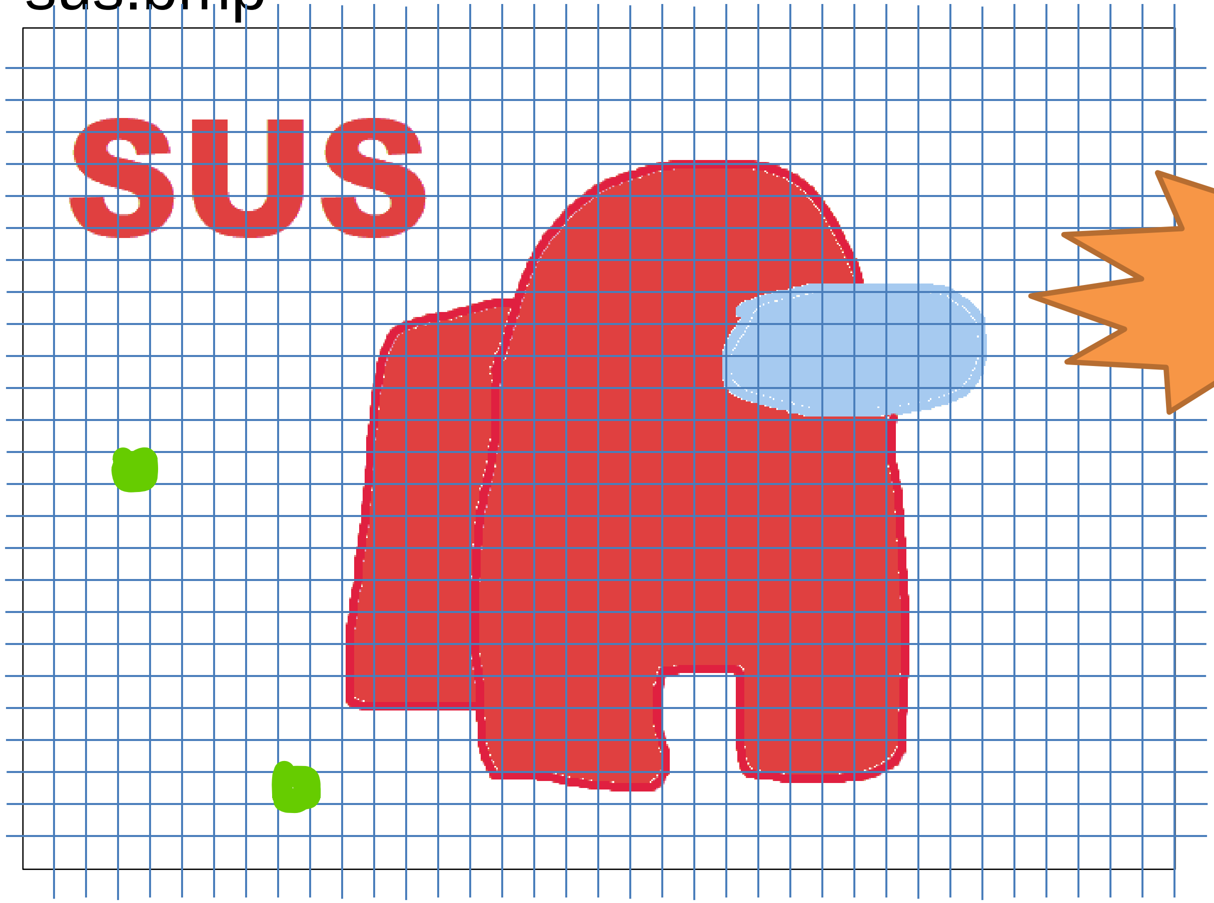
Take the first 54 bytes of the <u>original</u> image (header)

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



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**

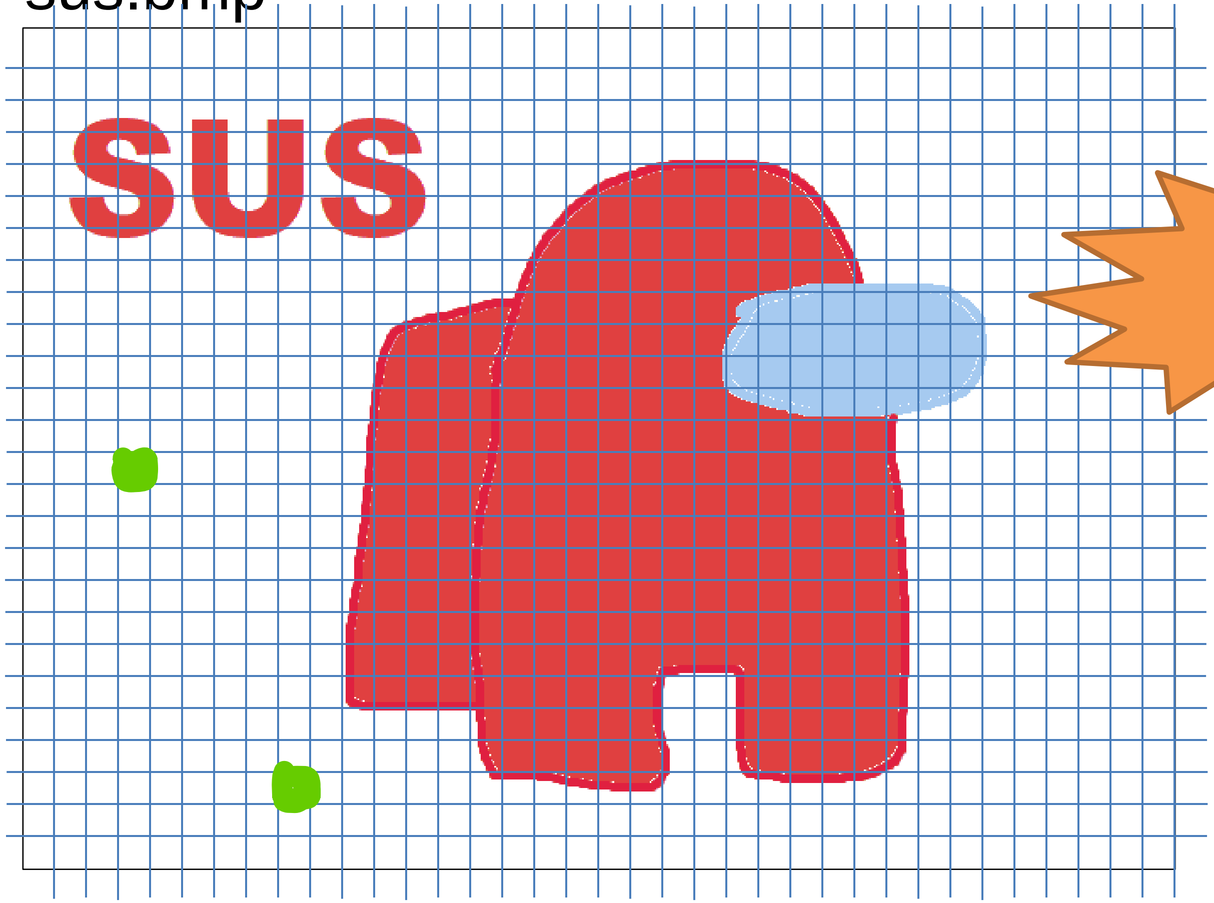
Dividing this image up, we can see that there are many blocks that are the exact same!

Using OpenSSL to encrypt w/ ECB

Why does this suck?

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

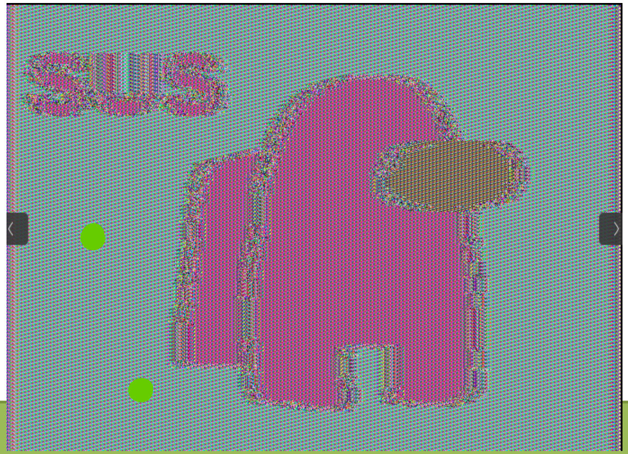
sus.bmp

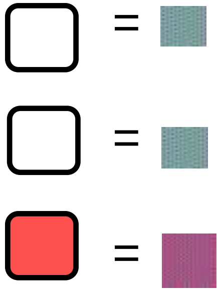


Important Properties

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

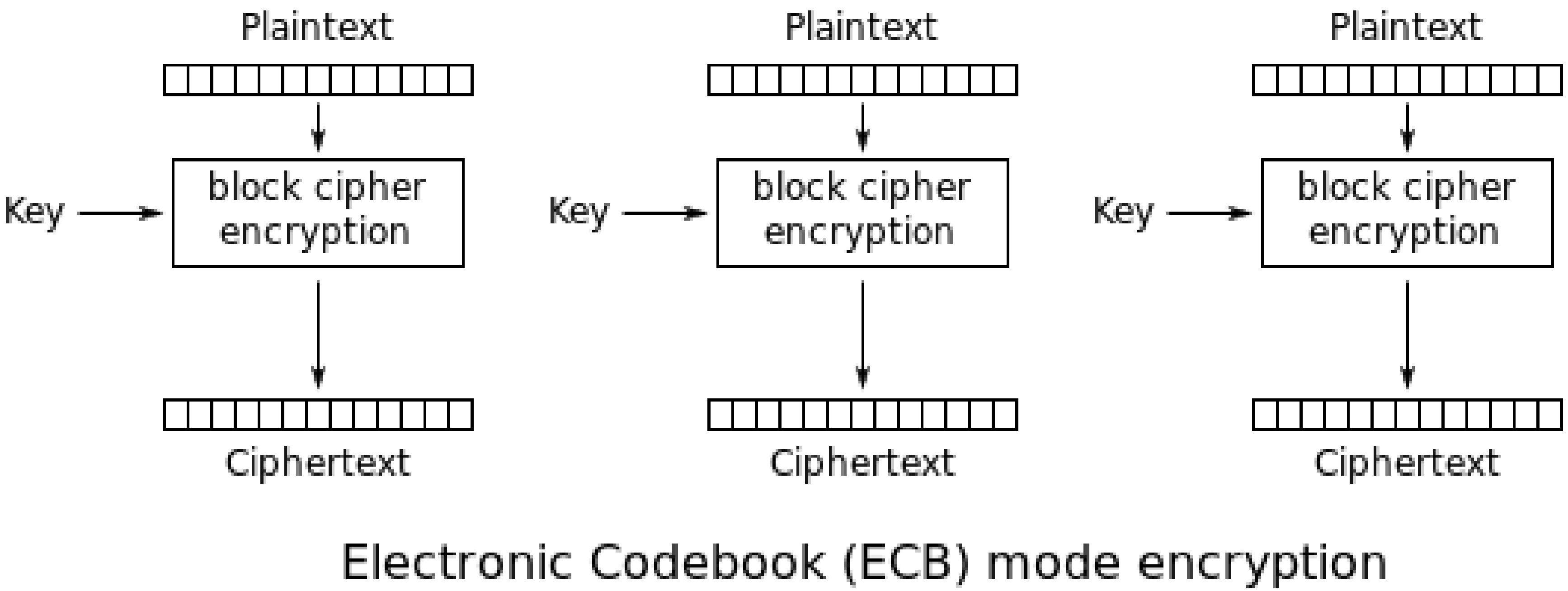
Let retry this experiment on a more **complex** image

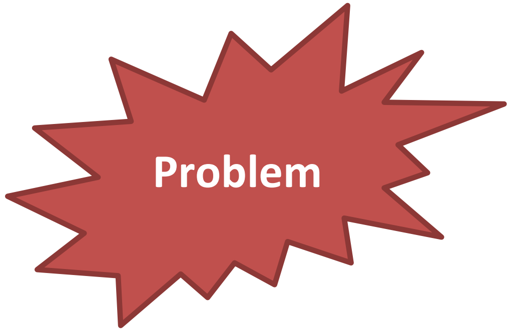
```
[11/09/22]seed@VM:~$ openssl enc -aes-128-ecb -e -in capy.bmp -out enc_capy.bmp -K 001122
3344556677889AABBCCDDEEEE
[11/09/22]seed@VM:~$ head -c 54 capy.bmp > header
[11/09/22]seed@VM:~$ tail -c +55 enc_capy.bmp > body
[11/09/22]seed@VM:~$ cat header body > final_capy.bmp
[11/09/22]seed@VM:~$ eog 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



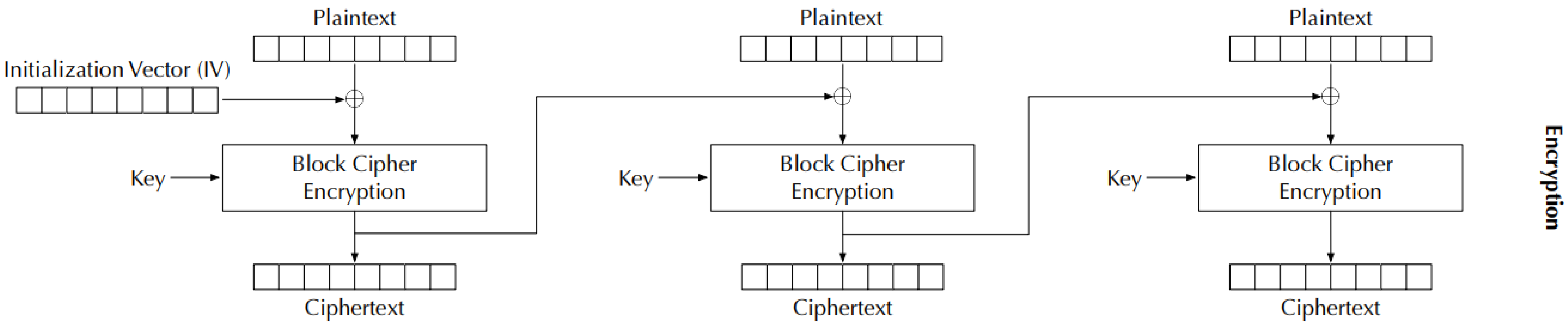


ECB can reveal information about our plaintext if our blocks are similar!

Solution: Add some randomness to each block during encryption

Cipher Block Chaining (CBC) Mode

Reminder: $\oplus$ = XOR operator



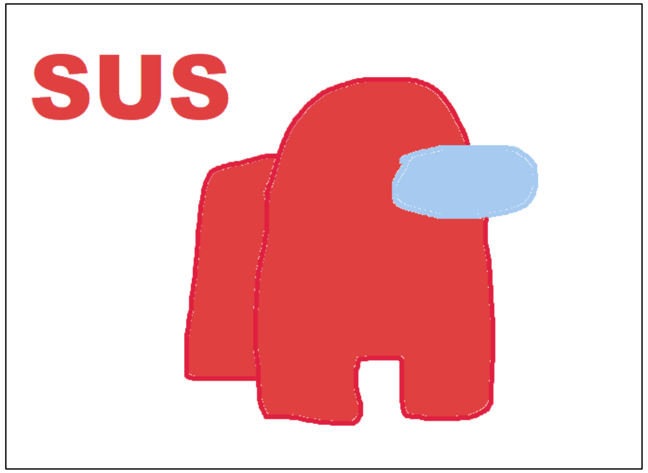
Introduces **block dependency** $C_i = E_K(P_i \oplus C_{i-1})$

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

Cipher Block Chaining (CBC) Mode

Reminder: ⊕ = XOR operator

Using CBC to encrypt images??



→ ???

You will do this on the lab.

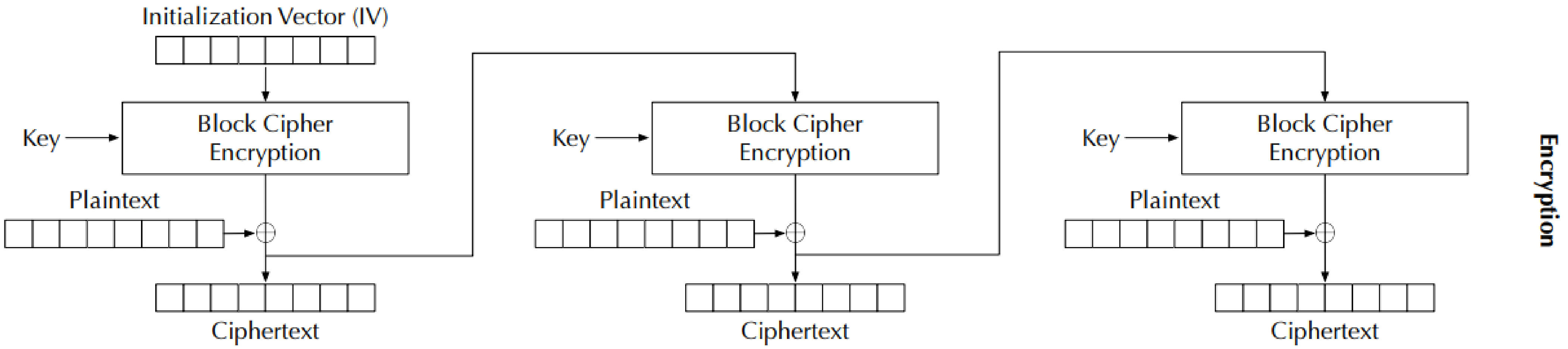
Using OpenSSL to encrypt w/ CBC

```
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
```

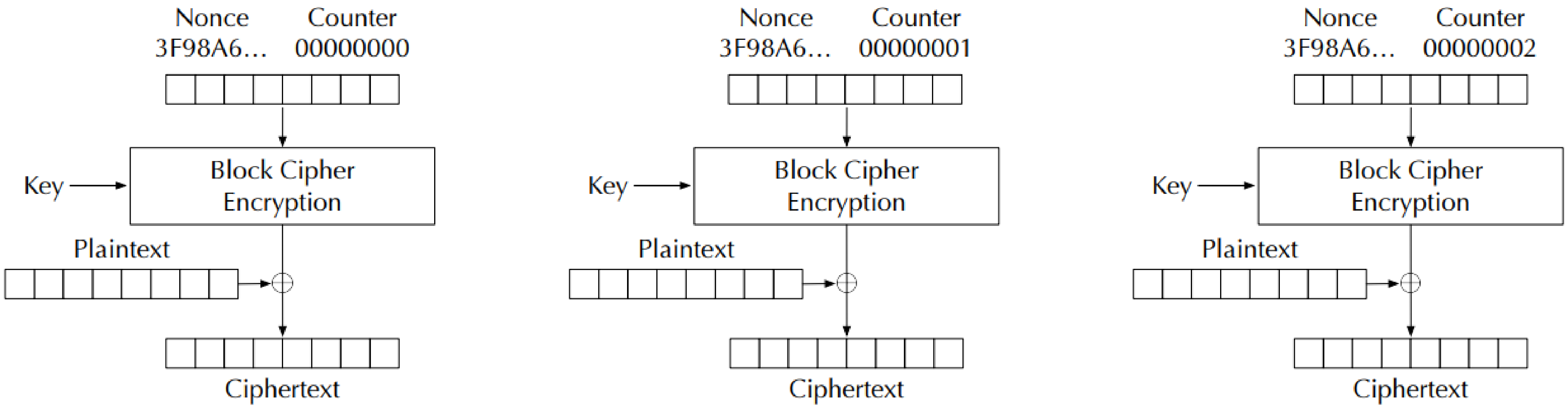
```
$ ls -ld cipher3.bin plain3_new.txt
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 cipher3.bin
-rw-rw-r-- 1 seed seed 32 Mar 18 21:07 plain3_new.txt
```

```
$ 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
```

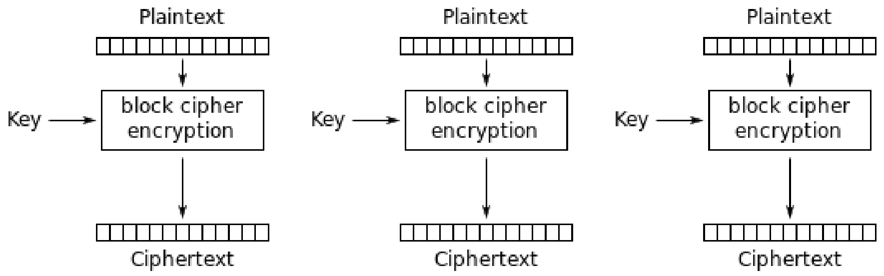
- Size of plaintext (plain3.txt) is **16 bytes**
- Size of decryption output (plaint3_new.txt) is **32 bytes** → *a new, full block is added as the padding*
- *In PKCS#5, if the input length is already an exact multiple of the block size B, then B bytes of value B are added as the padding.*

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)**

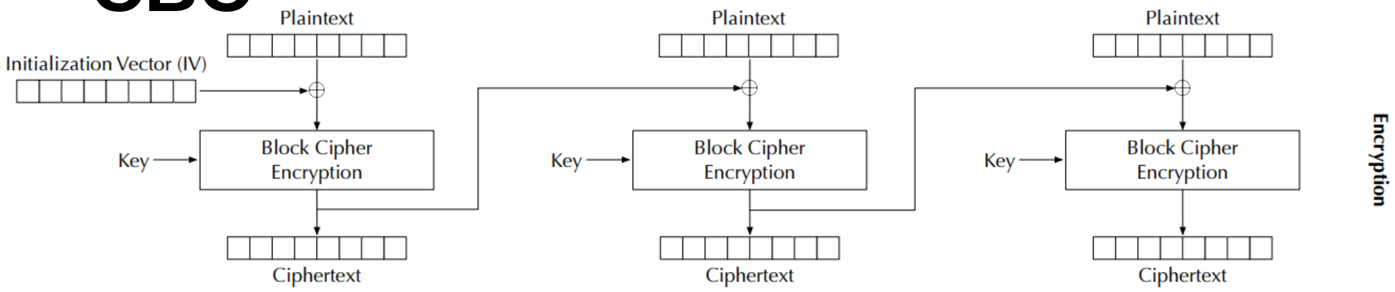


Modes of Encryption

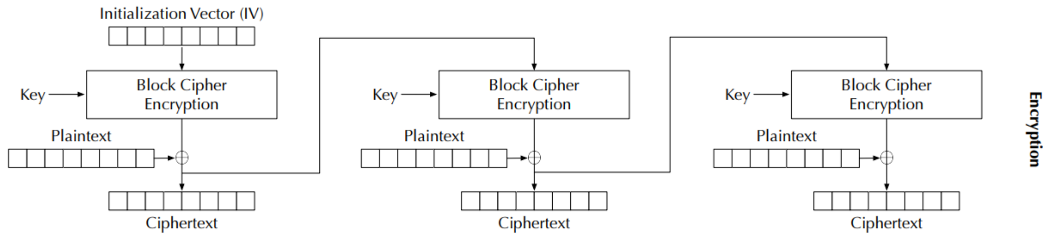


Electronic Codebook (ECB) mode encryption

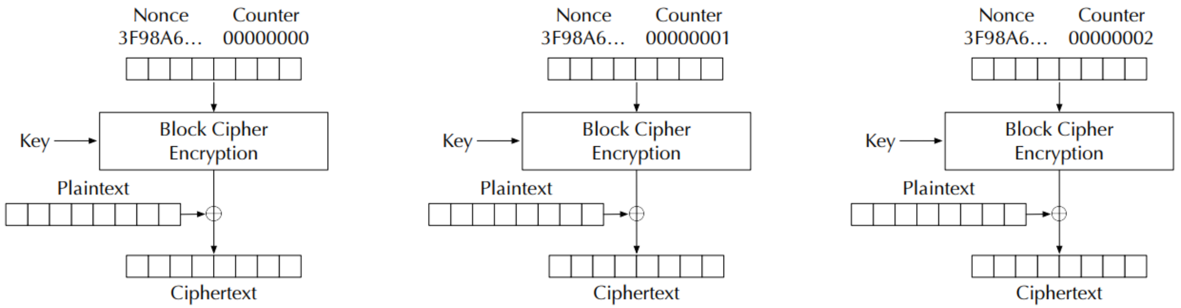
CBC



Cipher Feedback (CFB) Mode



Counter(CTR) Mode



You will explore these in the lab

Encryption Modes

- EBC
- CBC
- CFB
- CTR

None of the encryption modes discussed so far can be used to achieve **message authentication**
(we’ve only done message confidentiality)

How are keys actually exchanged?? We will talk about that when we get to asymmetric crypto

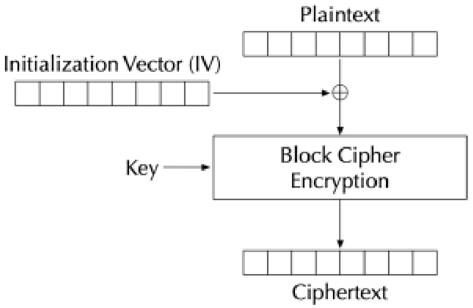
Initialization Vectors

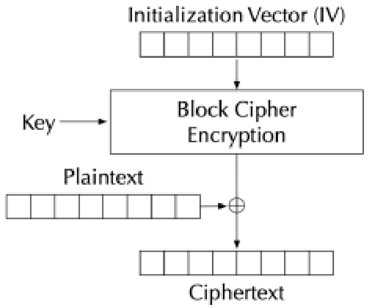
Initialization Vectors and Common Mistakes

- 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:

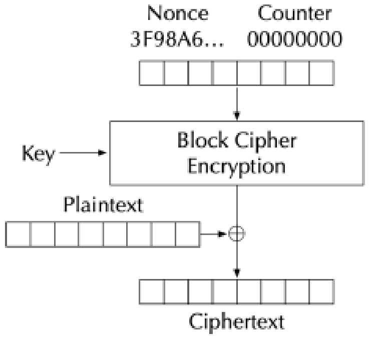
CBC

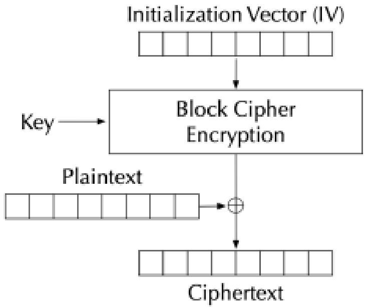




CFB

CTR





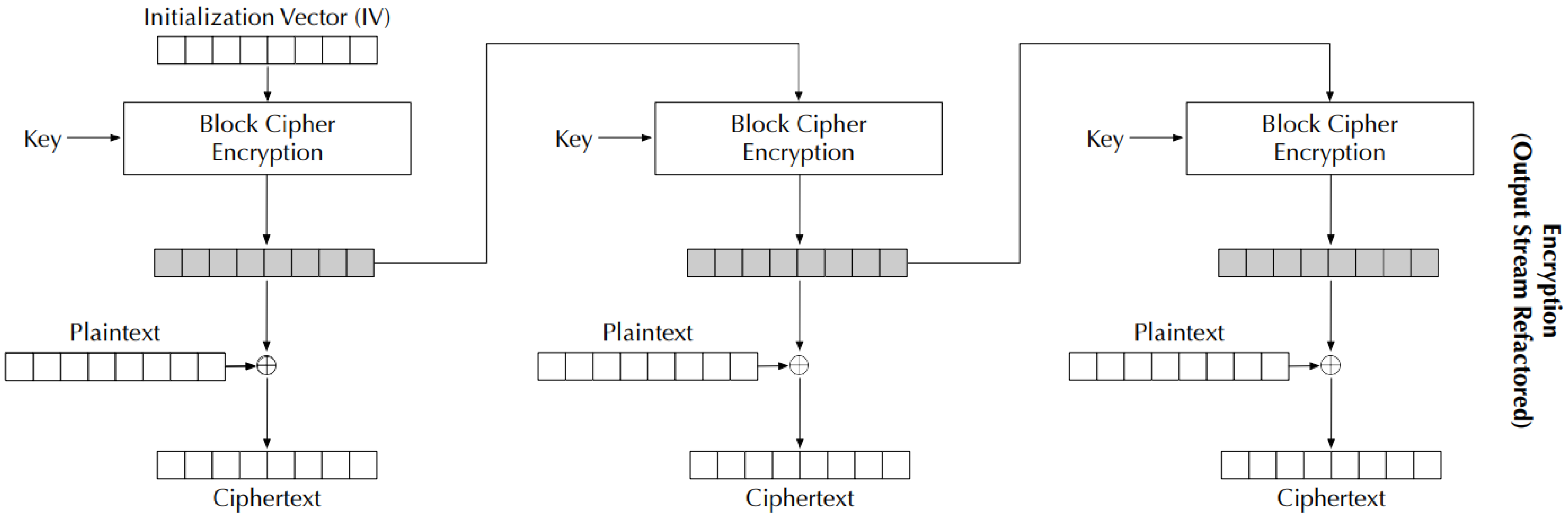
OFB

IV should not be **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

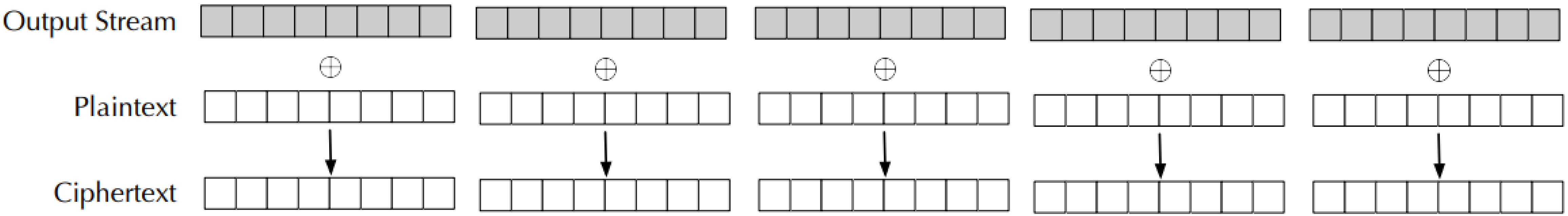


IV should not be **reused**...

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

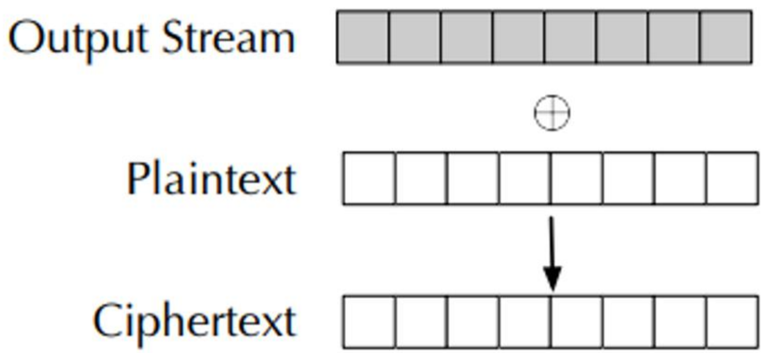


Chosen Plaintext Attack:

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?



??

110011

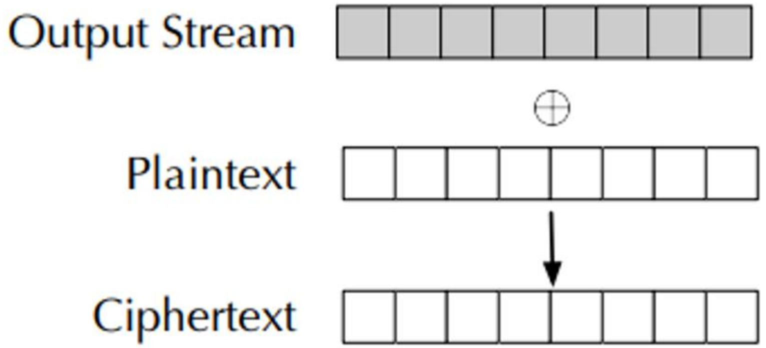
1010101

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?

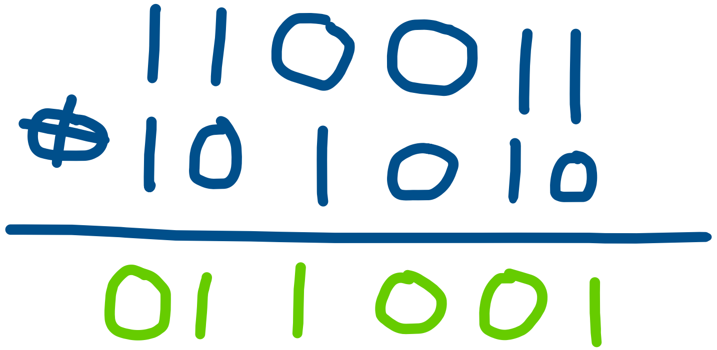


??

110011

1010101

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

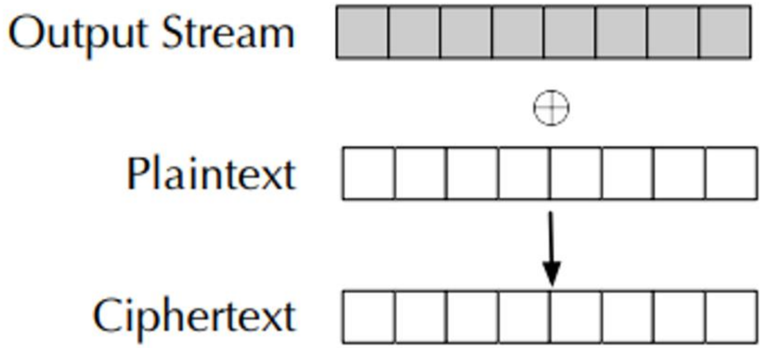


Chosen Plaintext Attack:

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?

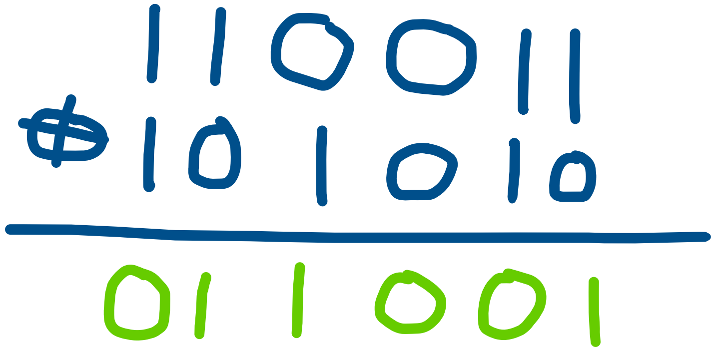


011001

110011

101010

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



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

Lab