

# CS 55: Security and Privacy

Symmetric encryption

## A CRYPTO NERD'S IMAGINATION:

HIS LAPTOP'S ENCRYPTED.  
LET'S BUILD A MILLION-DOLLAR  
CLUSTER TO CRACK IT.

NO GOOD! IT'S  
4096-BIT RSA!

BLAST! OUR  
EVIL PLAN  
IS FOILED!



## WHAT WOULD ACTUALLY HAPPEN:

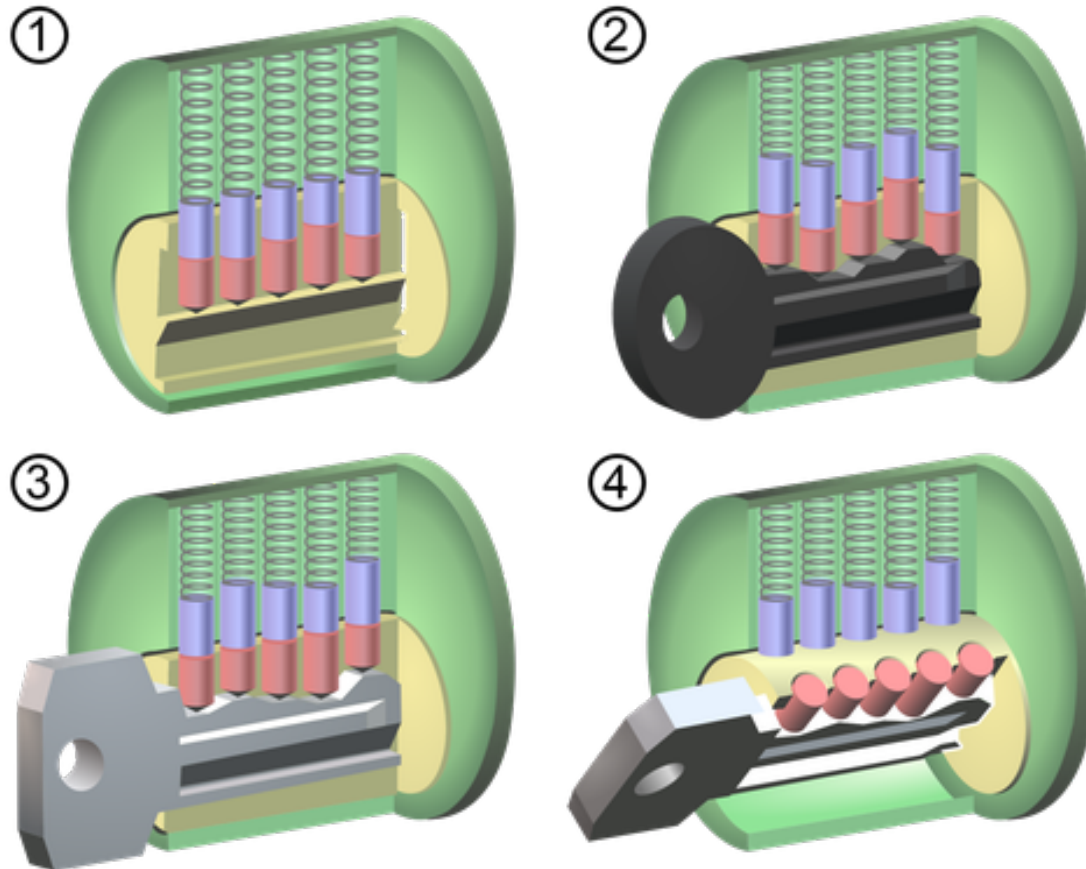
HIS LAPTOP'S ENCRYPTED.  
DRUG HIM AND HIT HIM WITH  
THIS \$5 WRENCH UNTIL  
HE TELLS US THE PASSWORD.

GOT IT.



# How to make a key to open a lock without a photo or physical access to the key

If you know the biting sequence, you can make a key as good as the original



Pin tumbler locks have pins set to various depths

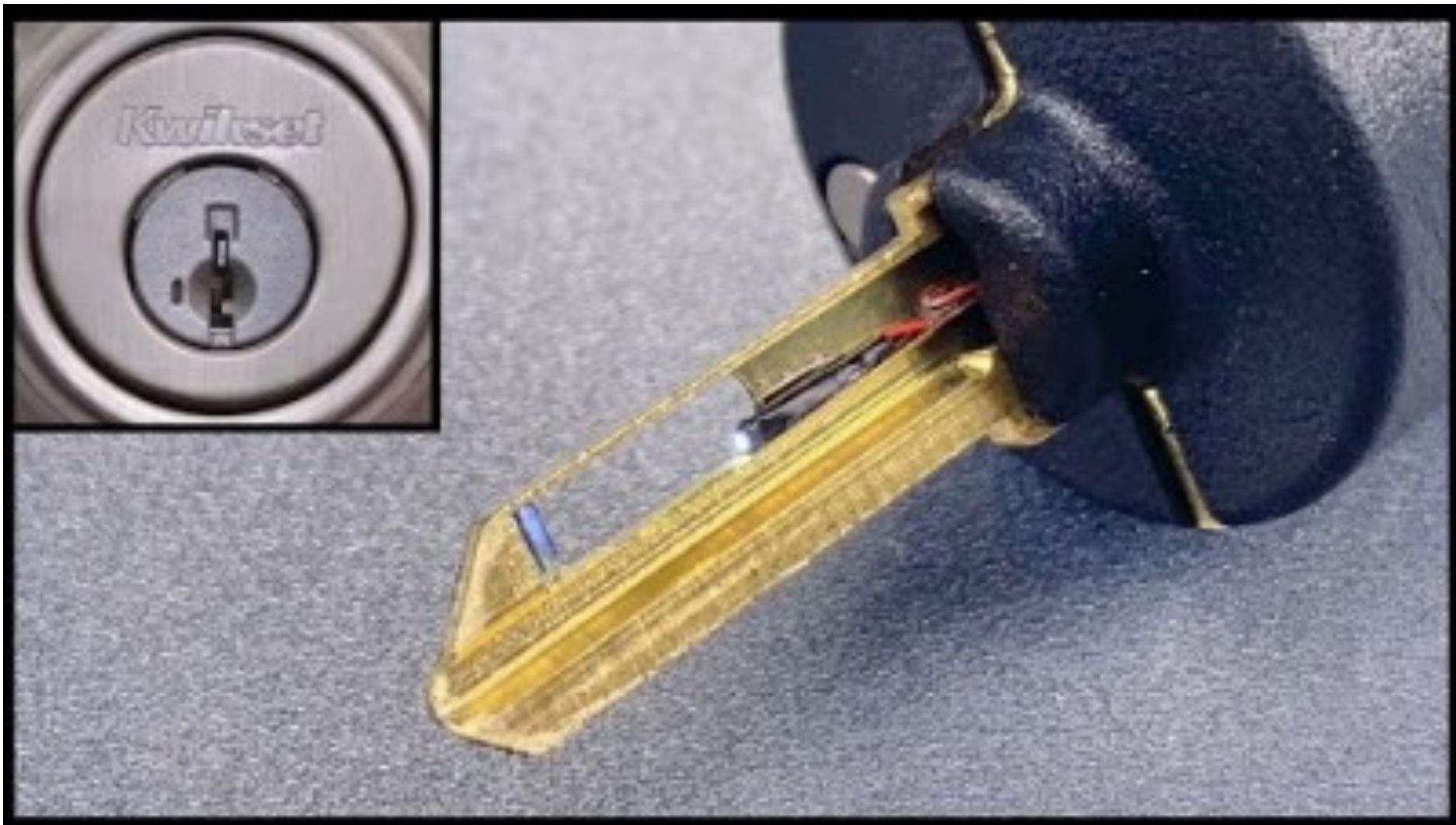
An inserted key displaces the pins

The right key aligns the pins with the shear line, allowing key to turn

The shape of the key is called the biting sequence

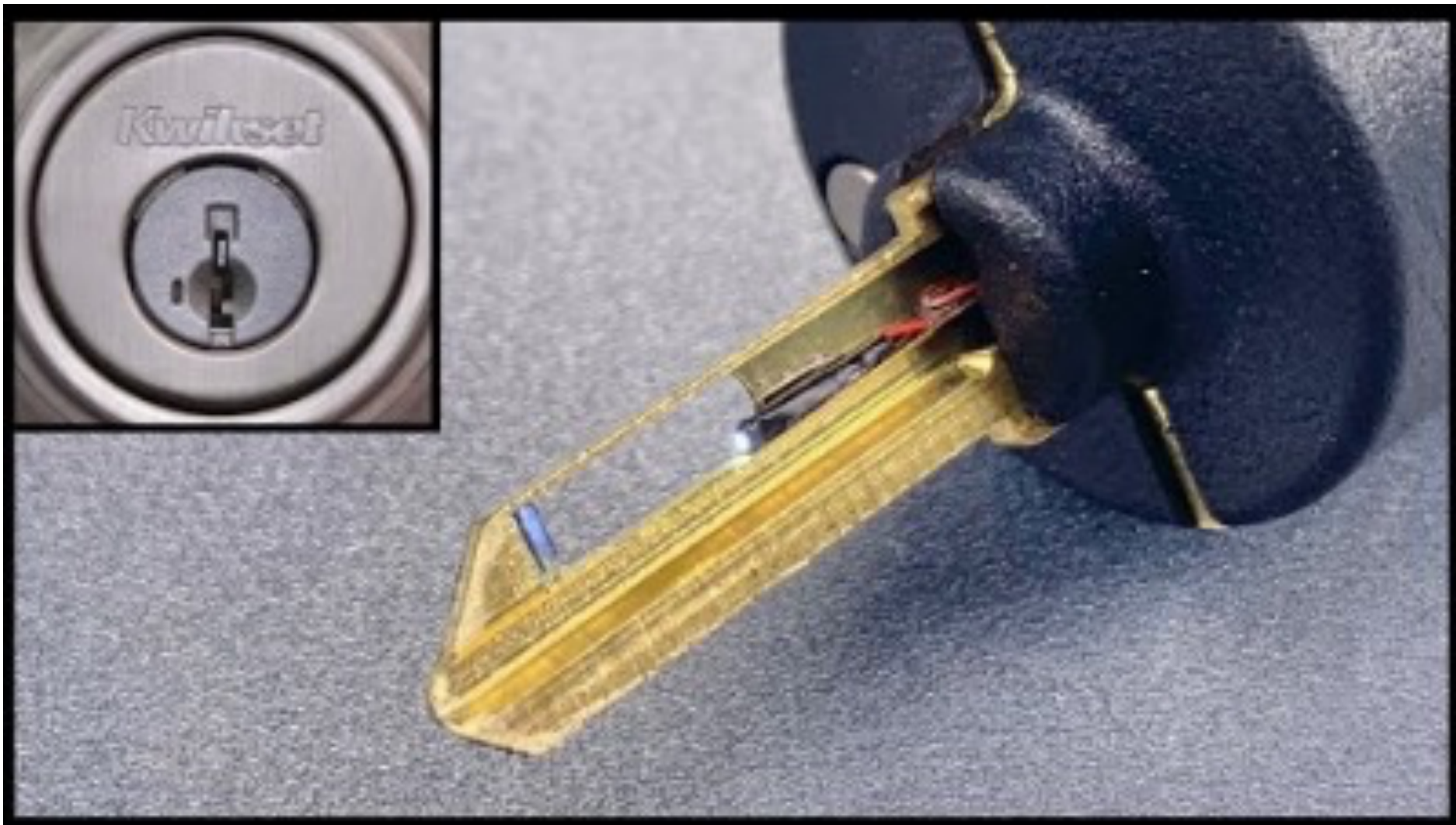
Normally around 5 positions, each position has around 6 possible depths

# How to make a key to open a lock without a photo or physical access to the key





# How to make a key to open a lock without a photo or physical access to the key



**Countermeasures?**

# Another approach uses a smartphone's microphone to listen to key insertion

## Listen to Your Key: Towards Acoustics-based Physical Key Inference

Soundarya Ramesh

sramesh@comp.nus.edu.sg

Department of Computer Science  
National University of Singapore

Harini Ramprasad

harinir@comp.nus.edu.sg

Department of Computer Science  
National University of Singapore

Jun Han

junhan@comp.nus.edu.sg

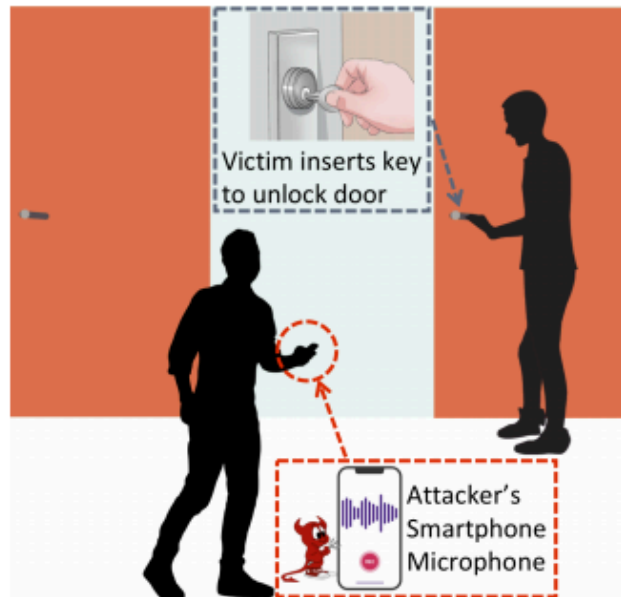
Department of Computer Science  
National University of Singapore

### ABSTRACT

Physical locks are one of the most prevalent mechanisms for securing objects such as doors. While many of these locks are vulnerable to lock-picking, they are still widely used as lock-picking requires specific training with tailored instruments, and easily raises suspicion. In this paper, we propose *SpiKey*, a novel attack that significantly lowers the bar for an attacker as opposed to the lock-picking attack, by requiring only the use of a smartphone microphone to infer the shape of victim's key, namely *bittings* (or cut depths) which form the secret of a key. When a victim inserts his/her key into the lock, the emitted sound is captured by the attacker's microphone. *SpiKey* leverages the time difference between audible clicks to ultimately infer the biting information, i.e., shape of the physical key. As a proof-of-concept, we provide a simulation, based on real-world recordings, and demonstrate a significant reduction in search space from a pool of more than 330 thousand keys to *three* candidate keys for the most frequent case.

### CCS CONCEPTS

• Security and privacy → Side-channel analysis and counter-measures; • Hardware → Sound-based input / output.




Bitting sequence recovered from listening to the sounds made by a key inserted into a lock

# Use defense in depth!

Employ multiple countermeasures

# Agenda

- 
1. Substitution and transposition ciphers
  2. Modern symmetric ciphers
  3. AES Modes
  4. Using crypto APIs

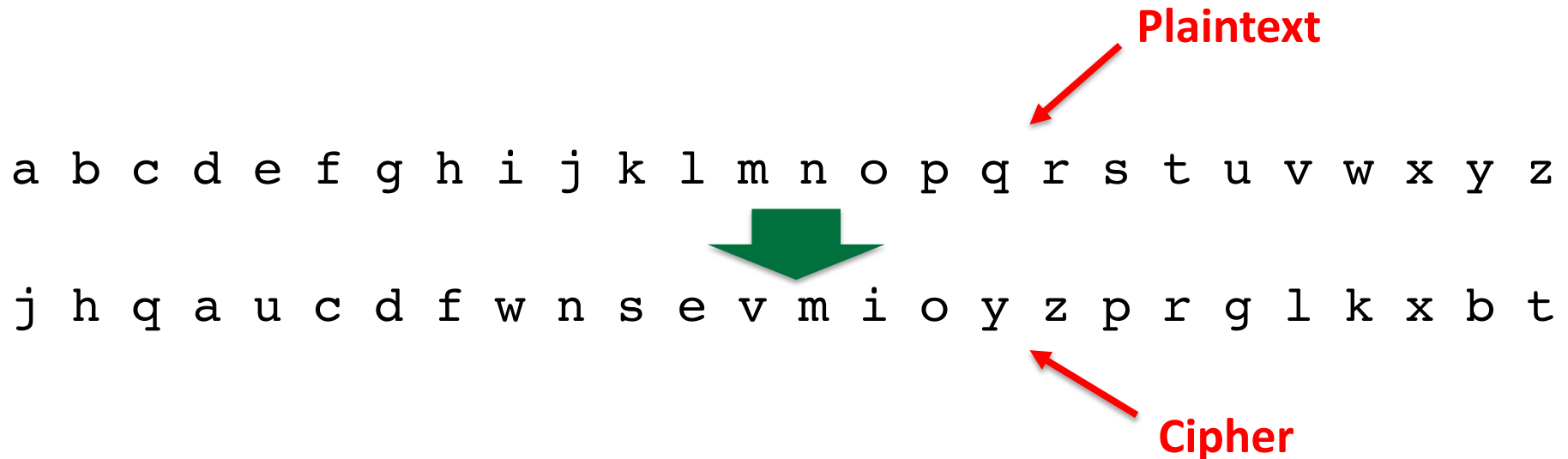


Do not create your own  
cryptography system!

Use vetted crypto instead

# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher

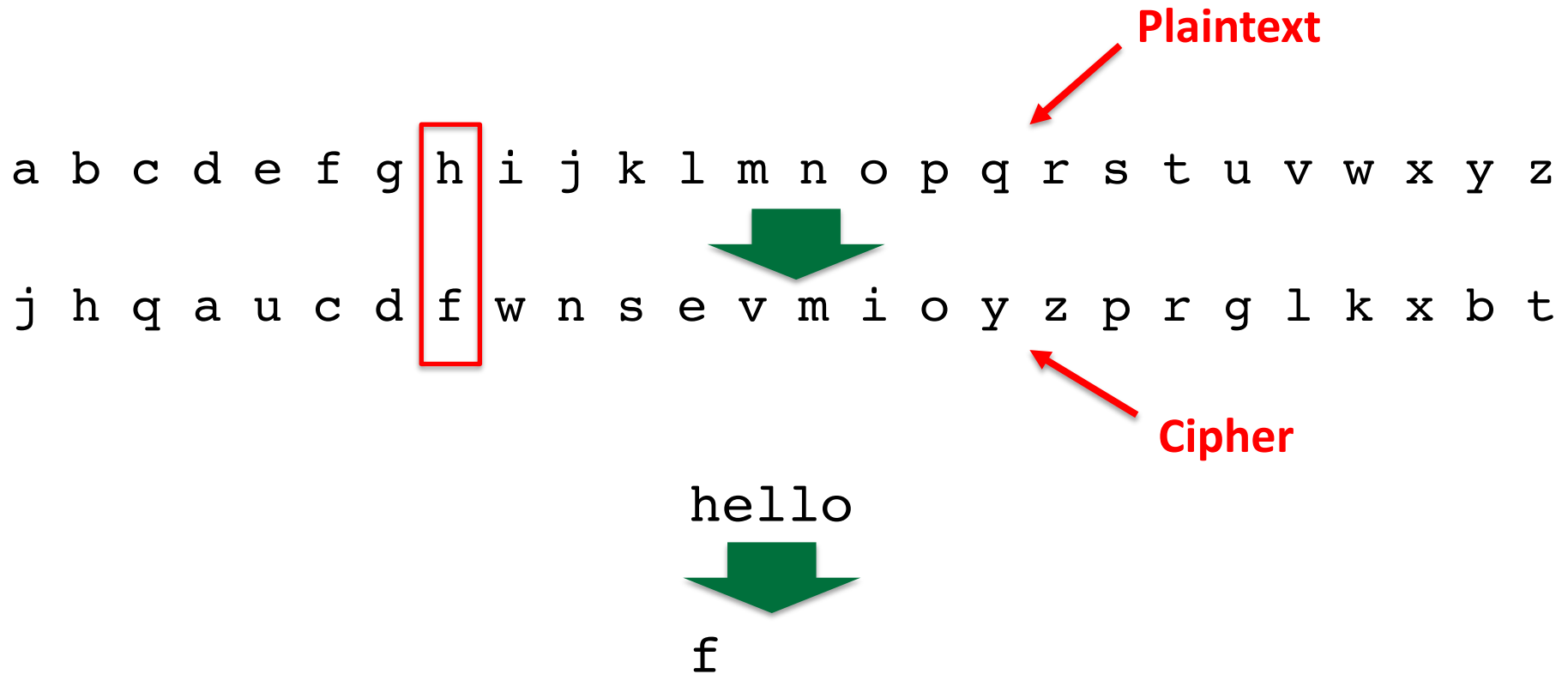


Each character in plaintext has a corresponding character in the cipher

Replace the plaintext character with its matching ciphertext character

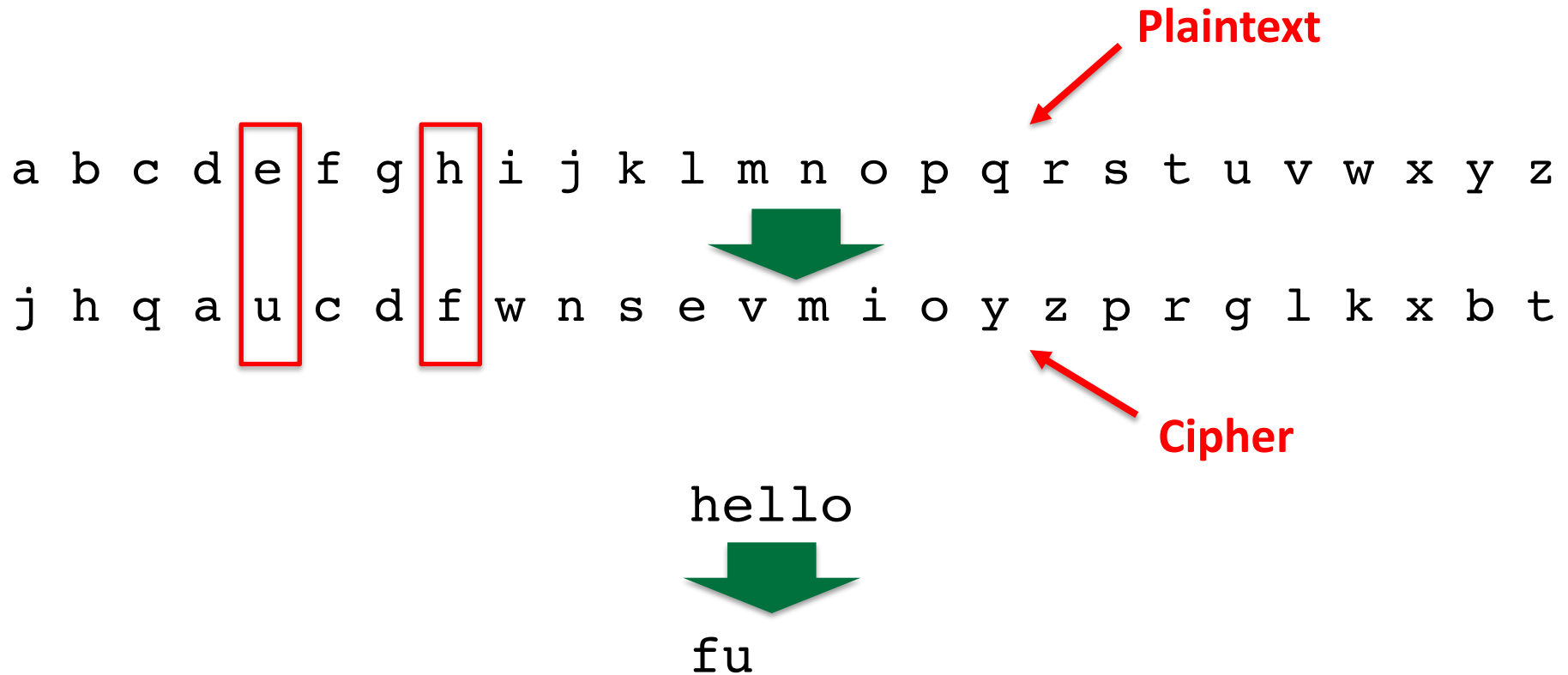
# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher



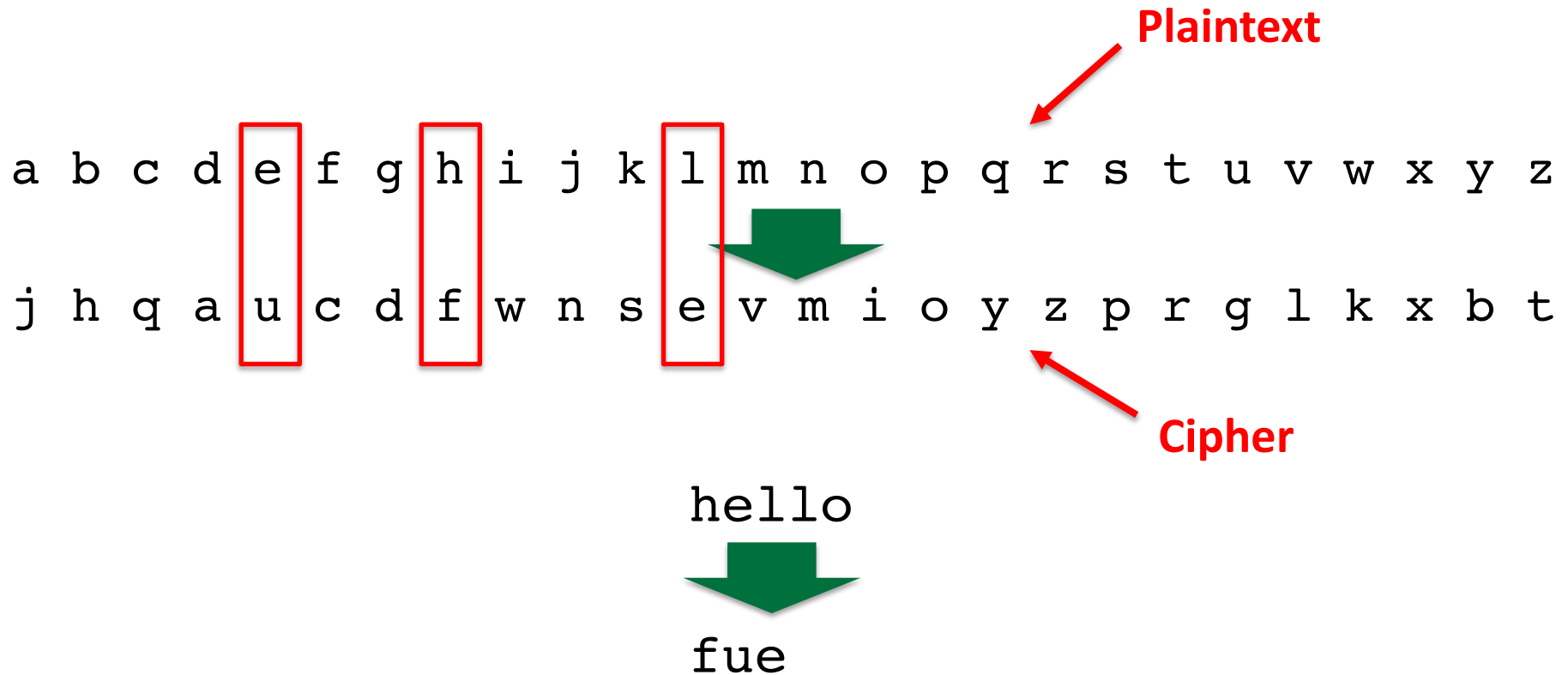
# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher



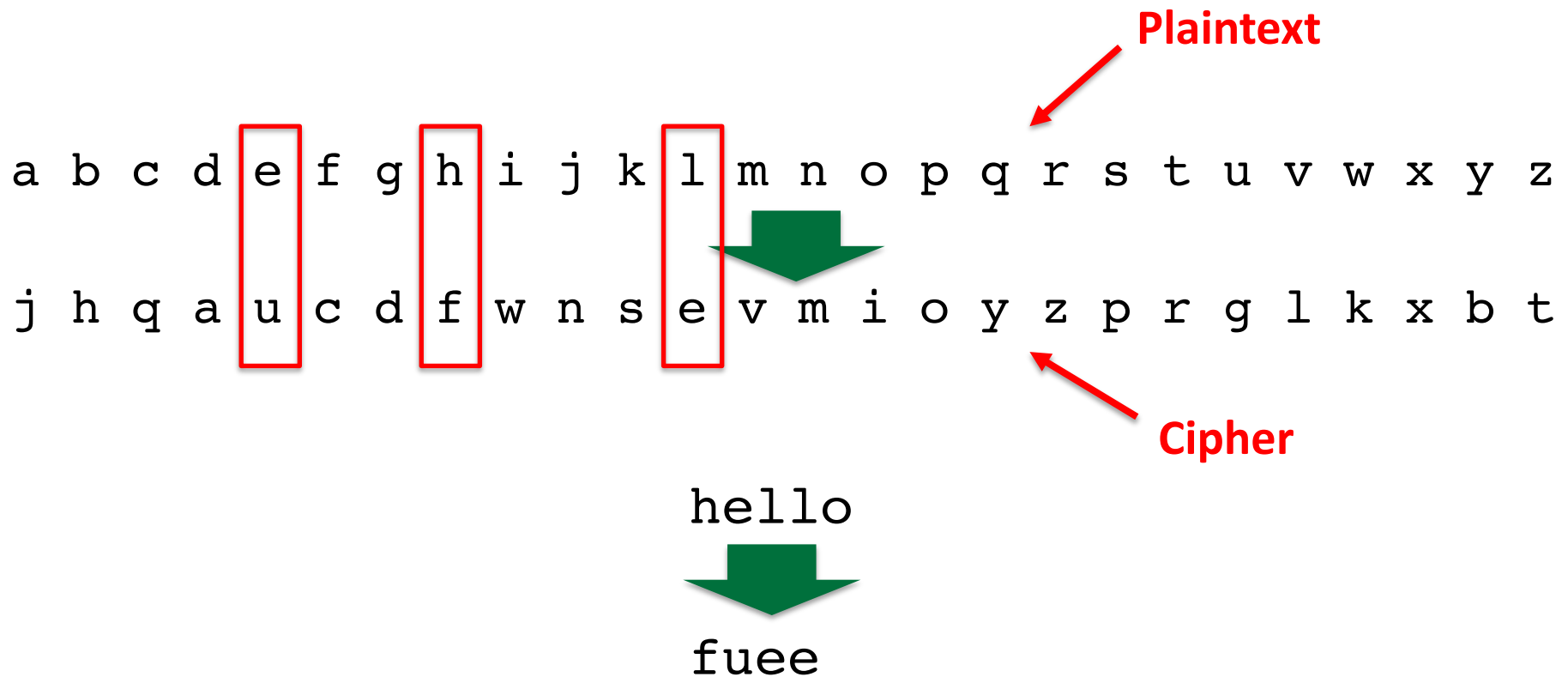
# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher



# With a substitution cipher, each character is replaced with another character

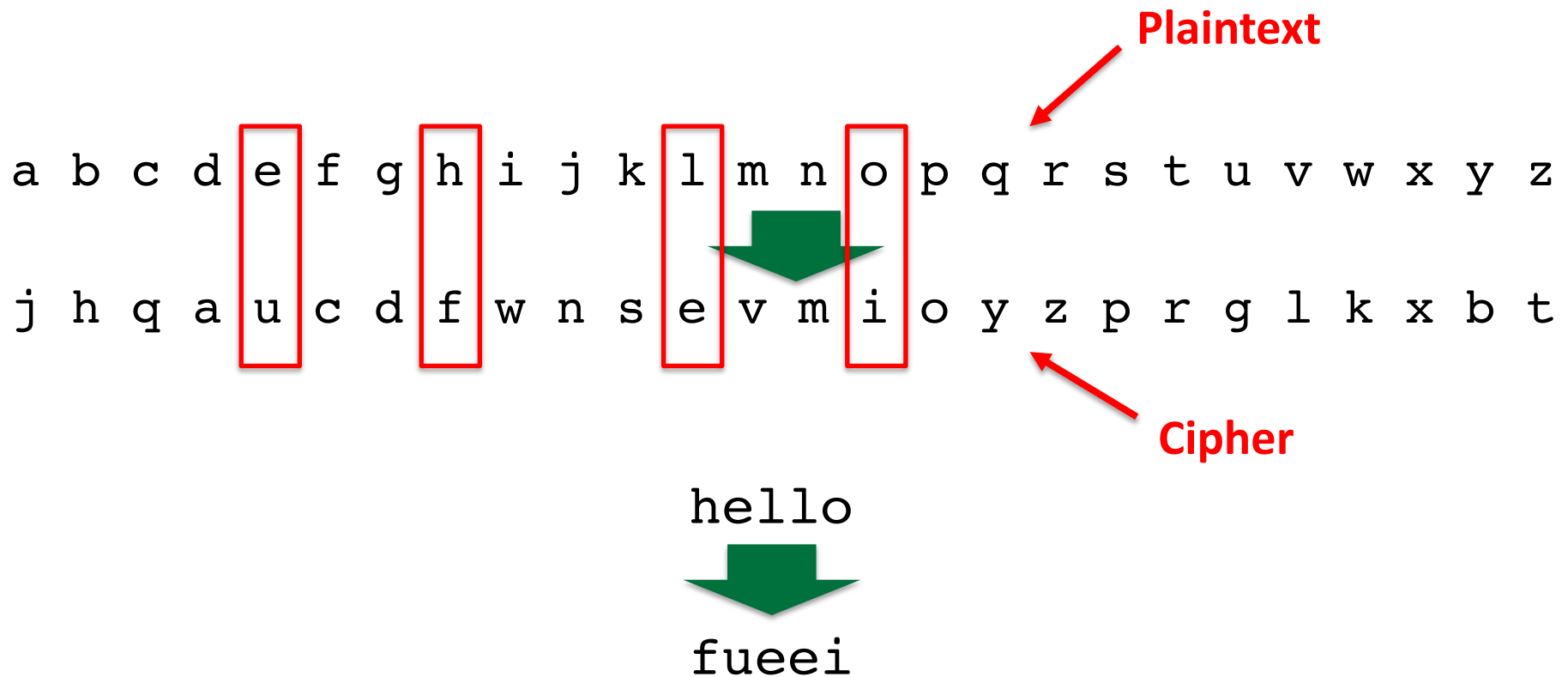
## Monoalphabetic substitution cypher





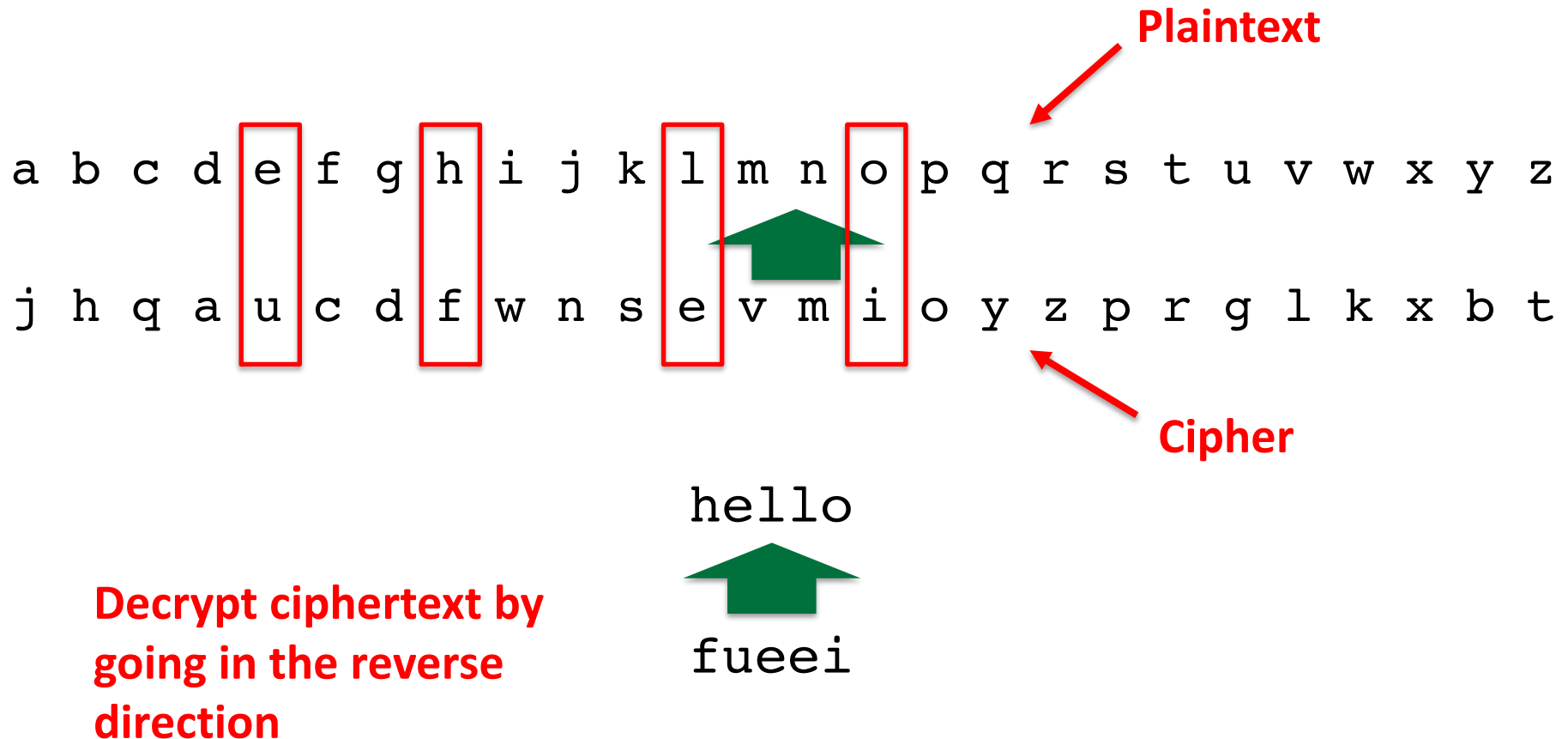
# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher



# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher



# With a substitution cipher, each character is replaced with another character

## Monoalphabetic substitution cypher

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
j	h	q	a	u	c	d	f	w	n	s	e	v	m	i	o	y	z	p	r	g	l	k	x	b	t

A plaintext character is always replaced with the same ciphertext character

hello  
↑  
fueei

Key point: both sender and receiver must know the mapping

26! Possibilities – seems strong, right?

# With a substitution cipher, encryption and decryption are done using the same key

## Develop key:

```
python3 monoalphabetic.py
```

- Outputs: jhqaucdfwnsevmioyzprglkxbt

```
import random

plain = "abcdefghijklmnopqrstuvwxyz"
cypher = random.sample(plain, len(plain))
print(''.join(cypher))
```

## Encrypt:

```
tr 'a-z' 'jhqaucdfwnsevmioyzprglkxbt' < plain.txt > cipher.txt
```

- Translates characters in plain.txt using key
- Saves output in cipher.txt

## Decrypt:

```
tr 'jhqaucdfwnsevmioyzprglkxbt' 'a-z' < cipher.txt > plain1.txt
```

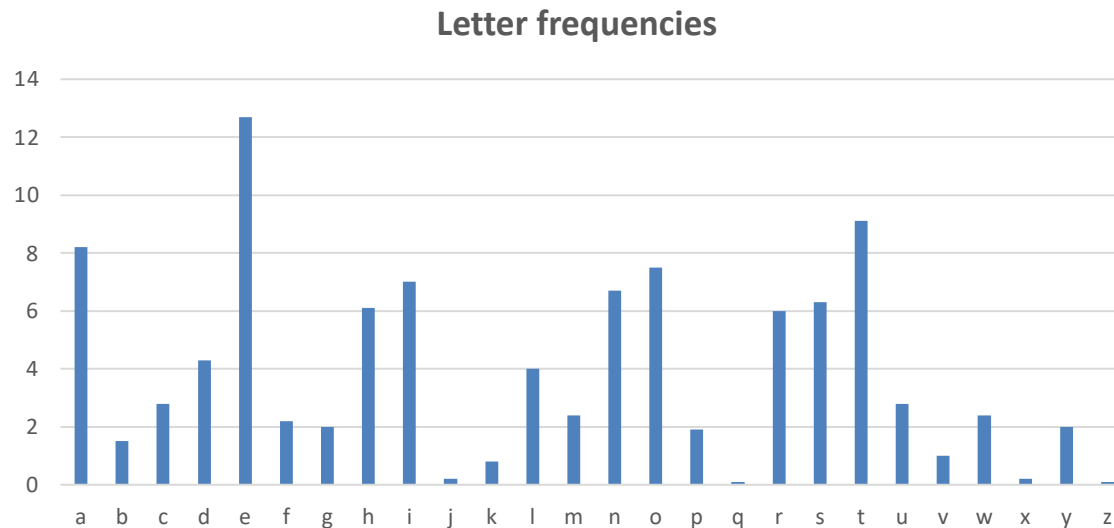
- Translates characters in cipher.txt replacing with a-z
- Saves output in plain1.txt

**Works assuming both sender and receiver both know the cipher**

# We can break substitution ciphers with frequency analysis

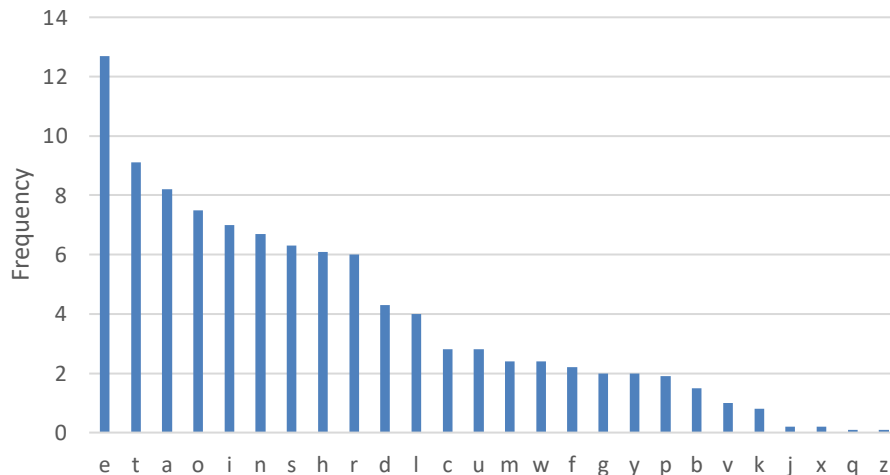
## Key ideas:

- Know or guess which language (e.g., English, German, ...) used in plaintext
- Look up how frequently different letters used in that language
  - e used in English most frequently
  - q and z not used frequently
- Letter that appears 100 times in plaintext will appear 100 times in ciphertext
- Letters are changed but letter frequencies are not (mapped 1:1)!
- Guess that frequent letters in ciphertext are frequent letters in English

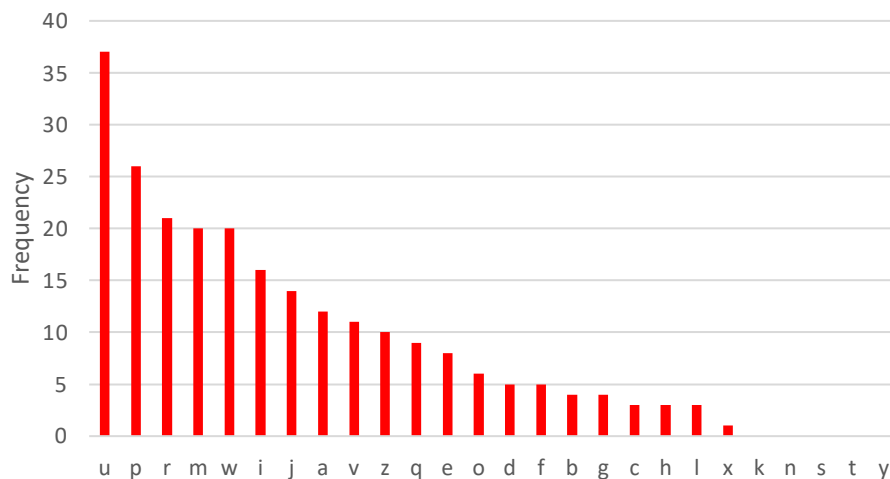


# Common letters in plaintext appear more frequently in ciphertext than others

English letter frequencies



Ciphertext letter frequencies



Letter frequencies in English language<sup>1</sup>

```
python3 counts.py cipher.txt
```

gives counts of characters in cipher.txt file

Good chance u=e

tr u E < cipher.txt

Look for THE candidates

Look for AND candidates

...

Assumes the key was: jhgaucdfwnsevmioyzprglkxibt as in the example on slide 19

[1] <http://www.richkni.co.uk/php/crypta/freq.php>



# Polyalphabetic ciphers are a better solution, but still not ideal

## Key ideas:

- Instead of one key, create N keys (say 1000)
- Encode character at position  $i$  with  $i^{\text{th}}$  dictionary (wrap around if  $i > N$ )
- Now same character repeated results in different characters

## Much more resistant to frequency analysis, but could be a problem if

- Use same set of N keys repeatedly
- Attacker sees large amount of ciphertext

See example in `polyalphabet.py`

**Key point: both sender and receiver must know the set of N keys**



German Enigma machine from WWII used polyalphabet principle

# A transposition cipher rearranges the order of the letters without substitution

## Transposition cypher

Message:

"nowrunalonganddontgetintomischiefiangoingoutxxxx"

Padding



Write message as matrix (here 4 rows with 12 letters each)

```
nowrunalonga
nddontgetint
omischiefiam
goingoutxxxx
```

There **\*many\* variations**  
around this transposition idea

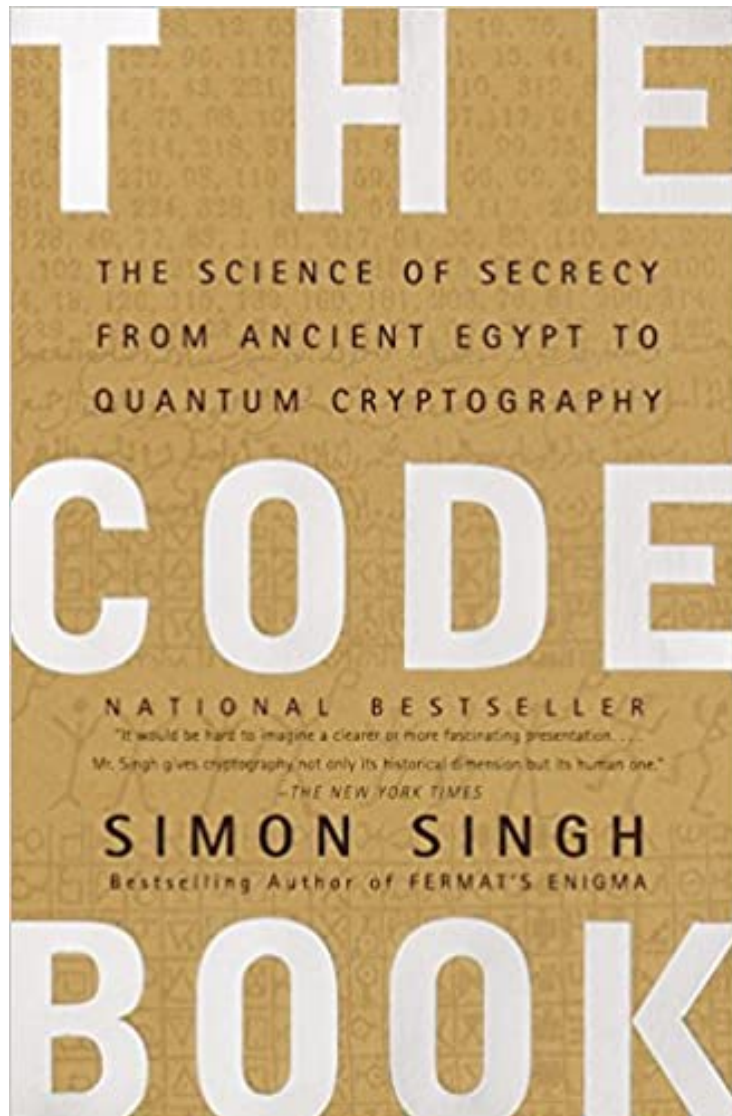
Key point: Characters change  
positions but are not substituted  
with other characters

Encrypt by reading down columns

```
nnog odmo wdii rosn uncg ntho agiu leet otfx
niix gnax atmx
```

Decrypt by writing ciphertext  
characters column by column

# Book recommendation



If you are interested in learning more about secret codes used throughout history

***The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Computing***

By Simon Singh

<https://www.amazon.com/Code-Book-Science-Secrecy-Cryptography/dp/0385495323>

Covers many other ciphers!

# Agenda

1. Substitution and transposition ciphers



2. Modern symmetric ciphers

3. AES Modes

4. Using crypto APIs

# A brief history of modern encryption: DES and AES

## Data Encryption Standard (DES)

- Developed by US government in 1970s from original IBM proposal
- Uses 56-bit keys (NSA reduced keys size by half....)
- Block cipher – works on blocks of 64 bits vs. stream ciphers which work bit-by-bit
- RSA Security sponsored series of DES challenges
  - Broken in 1997 by using idle cycles of machines on the Internet
  - In 1998 Electronic Freedom Foundation (EFF) built special-purpose Deep Crack which could crack DES messages in 56 hours



## Solution: Triple DES

- Apply DES three times with different keys each time
- Key size now  $3 * 56 = 168$  bits, but slow to compute

**Cryptography –  
creating codes**

**Cryptanalysis –  
breaking codes**

## Better solution: Advanced Encryption Standard (AES) in 2000

# XOR has some nice properties and is often used in cryptography

## XOR

Truth table

	0	1
0	0	1
1	1	0

Returns 1 only if inputs are different, otherwise returns 0



# XOR has some nice properties and is often used in cryptography

## XOR

### Properties

1.  $A \oplus 0 = A$
2.  $A \oplus A = 0$
3.  $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
4.  $(B \oplus A) \oplus A = B \oplus 0 = B$

Alice XOR's Message  
with Key to create  
Cipher



Alice



Bob

### Problems?

- Frequency analysis
- Known ciphertext attack

### Alice

Key: 10101010

Message: 11000011

Cipher: 01101001

Eve sees cipher, but does not know  
Key, does not recover message

Alice sends Cipher to Bob  
Bob knows Key (shared secret)

Bob XOR's Cipher and Key to recover Message  
(property 4 above)

### Bob

Cipher--> 01101001

Key: 10101010

Message: 11000011

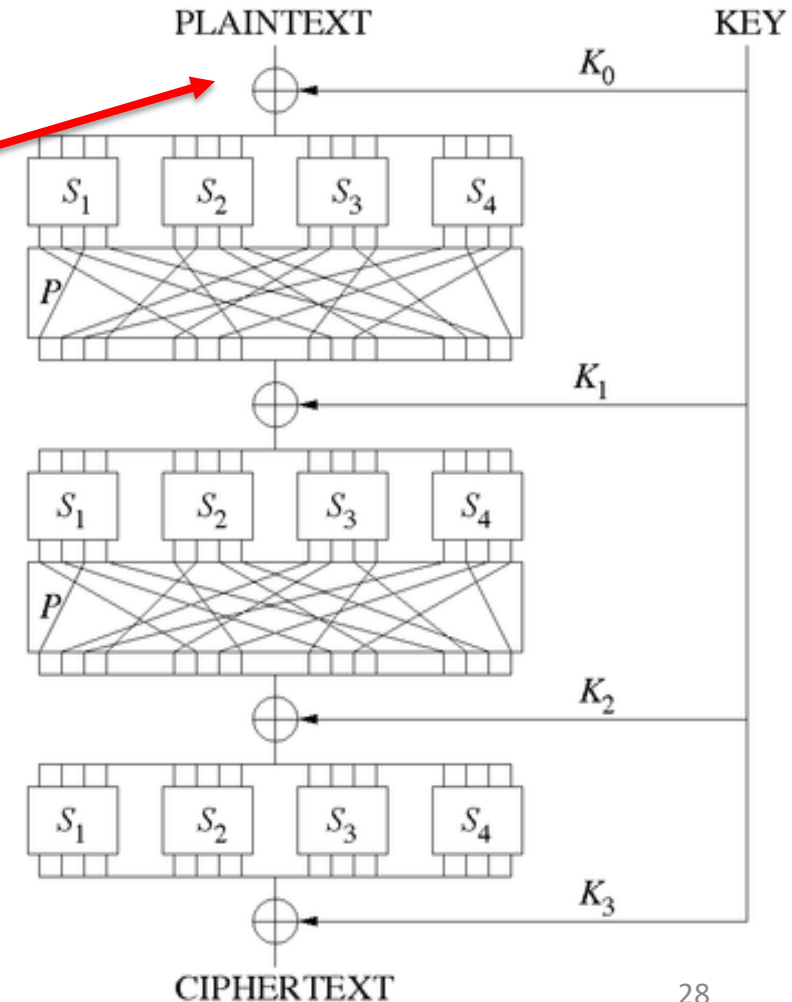
# S-P Networks: Intuition for how almost all modern symmetric key cryptography works

## Substitution-Permutation Network (S-P network)

Simplified

Key expanded into several “round keys” by key schedule algorithm

Fixed sized block of plaintext XOR'ed with key



# S-P Networks: Intuition for how almost all modern symmetric key cryptography works

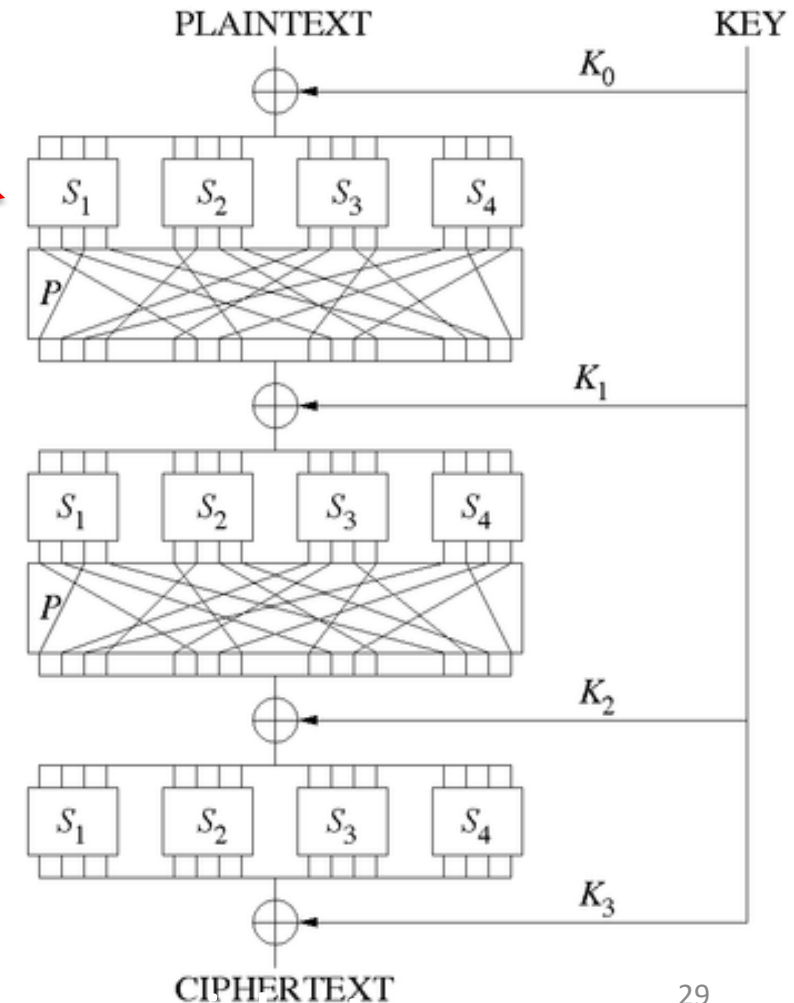
## Substitution-Permutation Network (S-P network)

Simplified

### Substitution (S-box)

- Look up table – given inputs bits, output different bits
- 1-to-1 mapping for invertibility (decryption)
- Often done in hardware for speed
- Provides confusion

Can think of S-box like a substitution cipher



# S-P Networks: Intuition for how almost all modern symmetric key cryptography works

## Substitution-Permutation Network (S-P network)

Simplified

### Permutation (P-box)

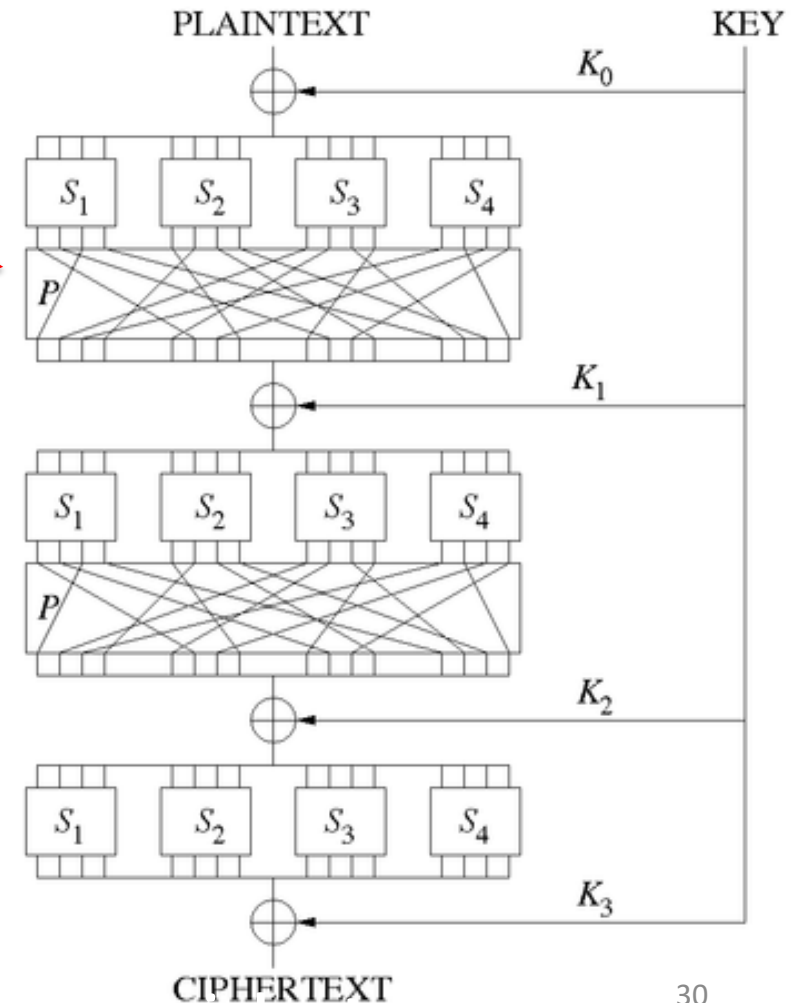
- Permutes bits (moves bits to new location)
- Distributes output bits to as many following S-boxes as possible

Changing one bit of input should change roughly half of the output bits (avalanche effect)

Therefore, each output bit depends on every input bit

Can think of P-box like a transposition cipher

Provides diffusion



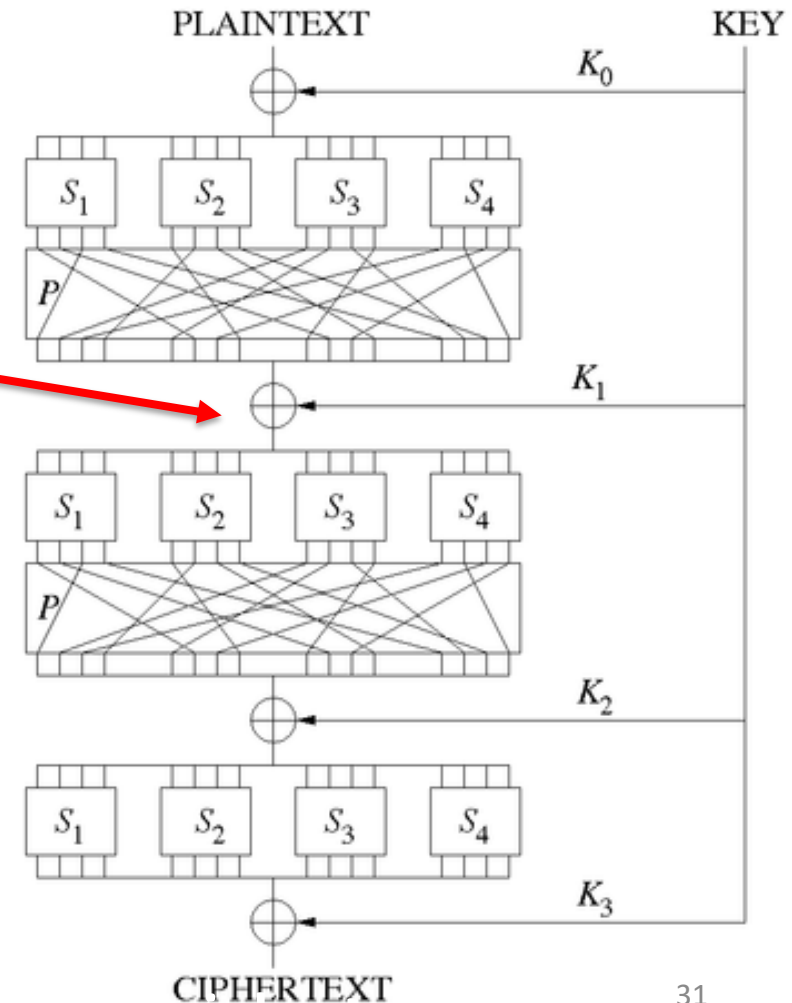
# S-P Networks: Intuition for how almost all modern symmetric key cryptography works

## Substitution-Permutation Network (S-P network)

Simplified

Output is XOR'ed with next round key

Process repeats for several rounds



# S-P Networks: Intuition for how almost all modern symmetric key cryptography works

## Substitution-Permutation Network (S-P network)

Simplified

A single S-box or P-box does not provide much cryptographic strength

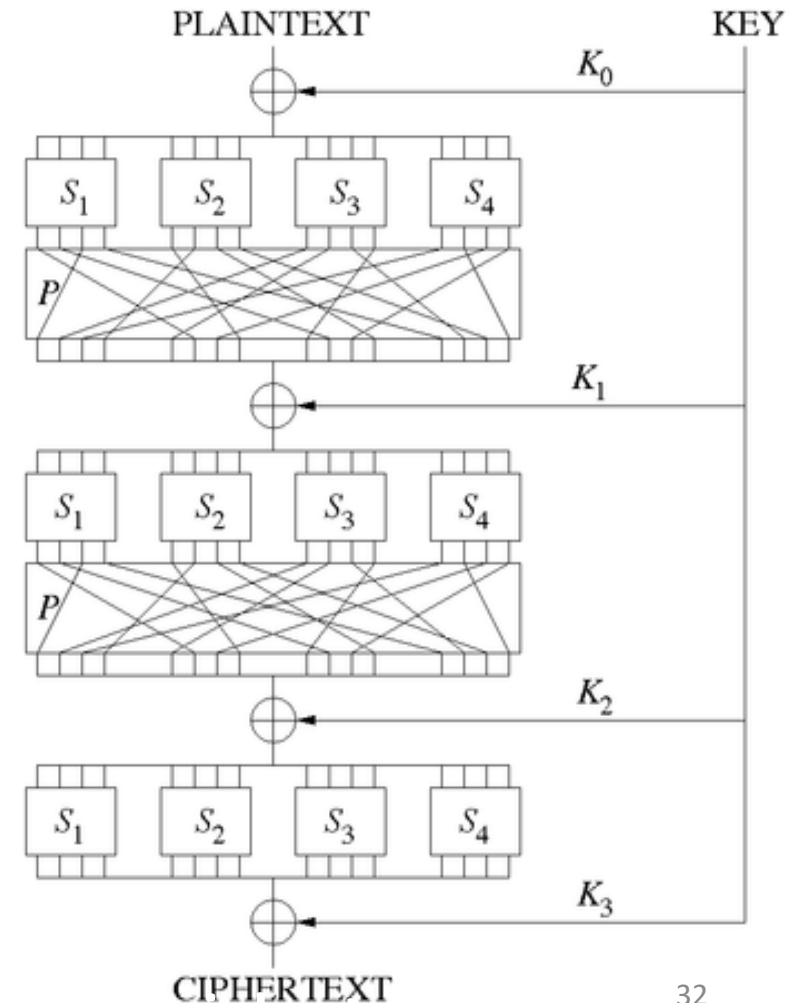
A S-P network with several rounds provides both confusion and diffusion

- If one input bit changes, several output bits will change
- Bits are distributed among several S-boxes

If  $i^{th}$  input bit flipped, probability  $j^{th}$  output bit flips is 50%

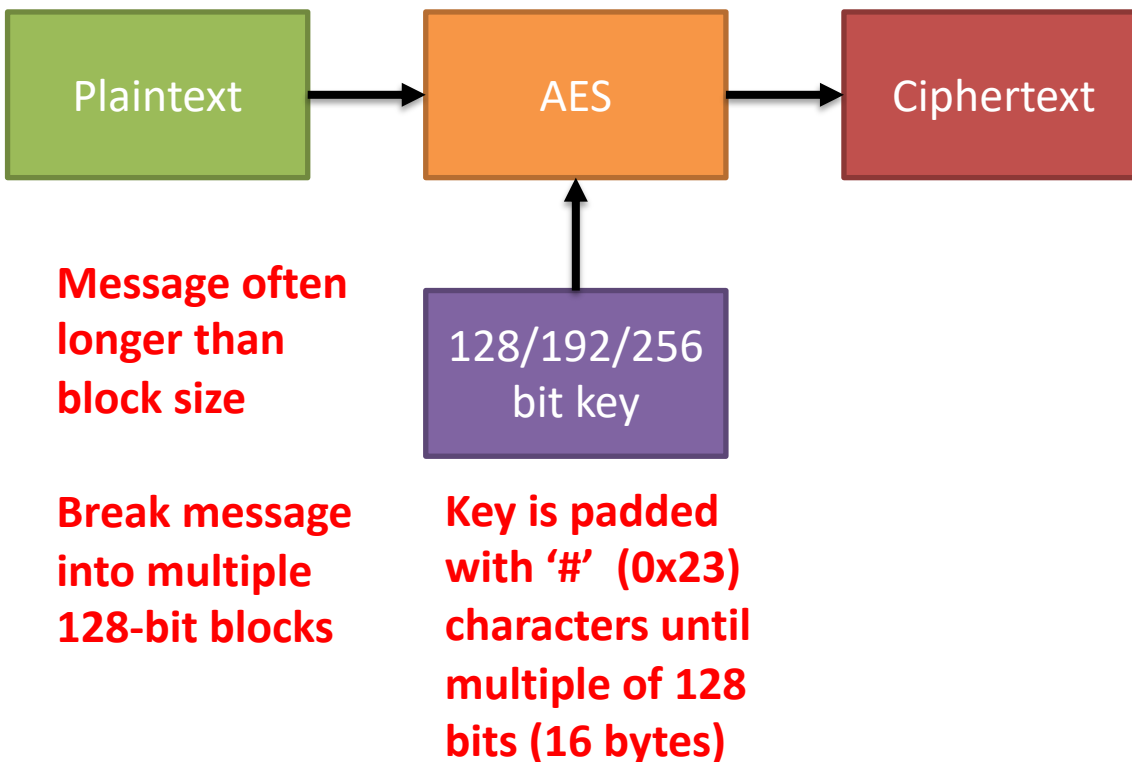
Similarly, if one output bit is flipped, 50% probability  $i^{th}$  input bit flips also

Changing one bit of key diffuses over all output bits





# Advanced Encryption Standard (AES) builds on the concept of an S-P network



## Advanced Encryption Standard (AES)

- Sometimes called Rijndael (pronounced like “rhine doll”) after inventor names
- Established by NIST in 2001 (but still secure)
- Uses 128-bit (16-byte) input blocks, outputs 128-bit ciphertext blocks
- Key can be 128, 192, or 256 bits (equals 10,12 or 14 rounds)

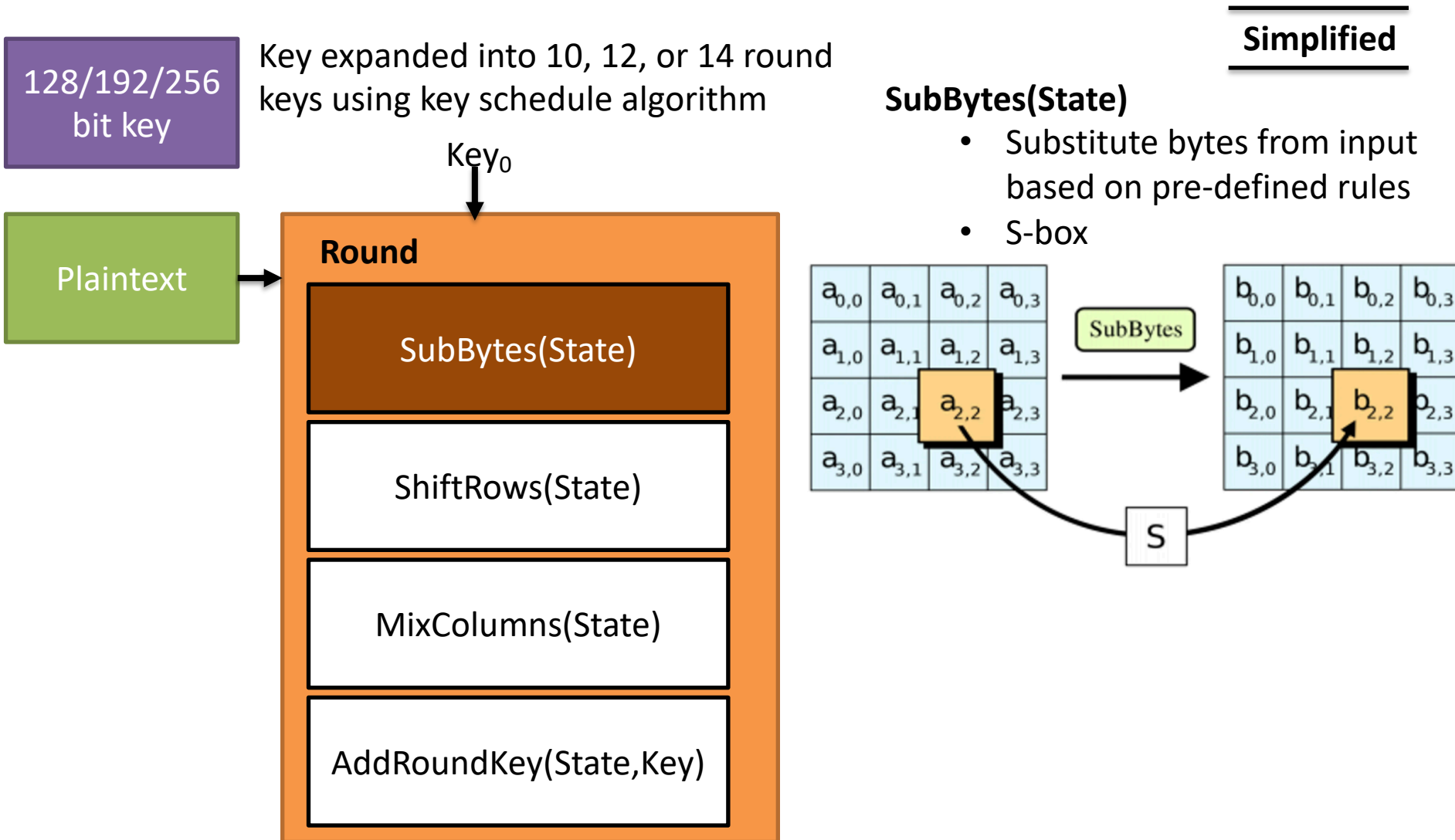
# Advanced Encryption Standard (AES) uses the concept of an S-P network

**Simplified**

128/192/256  
bit key

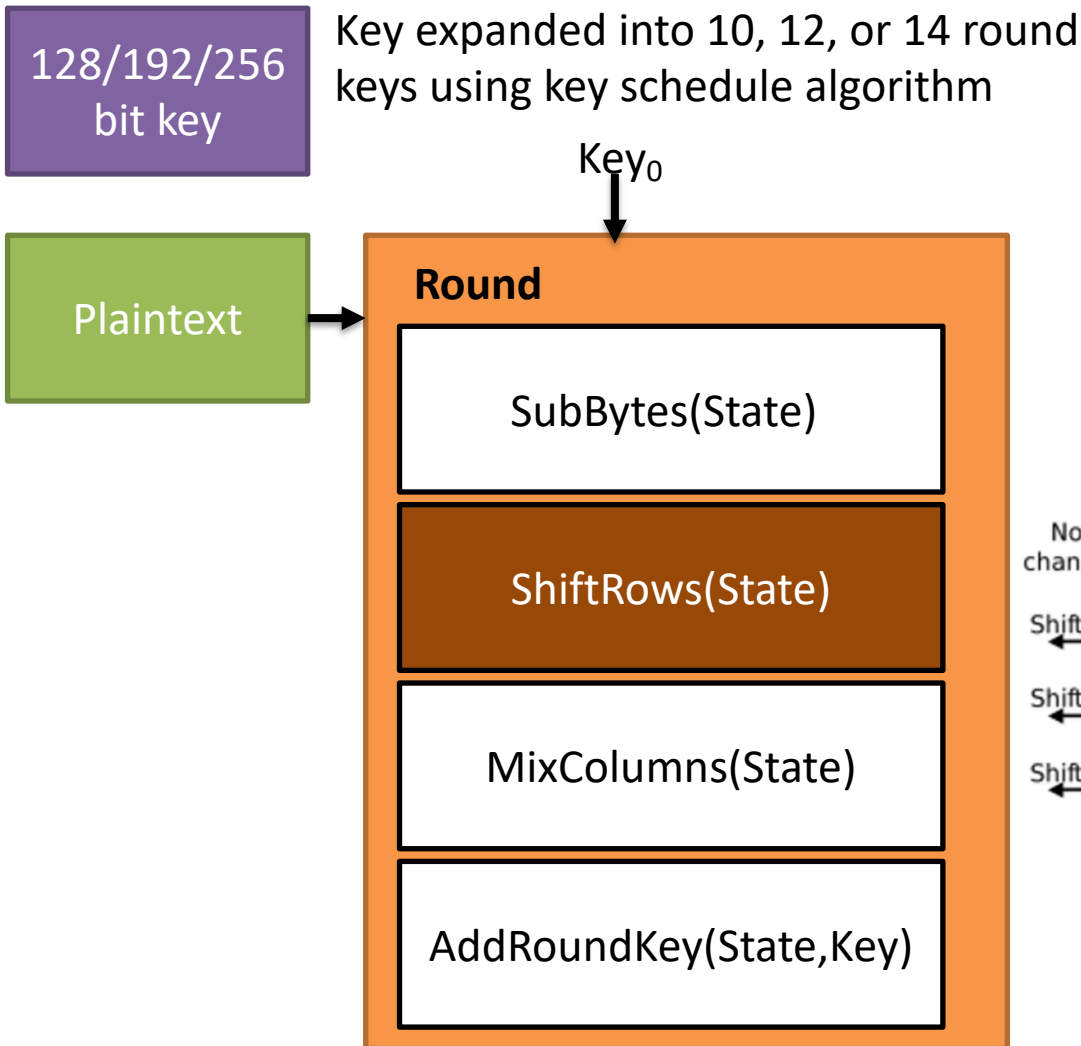
Key expanded into 10, 12, or 14 round  
keys using key schedule algorithm

# Advanced Encryption Standard (AES) uses the concept of an S-P network



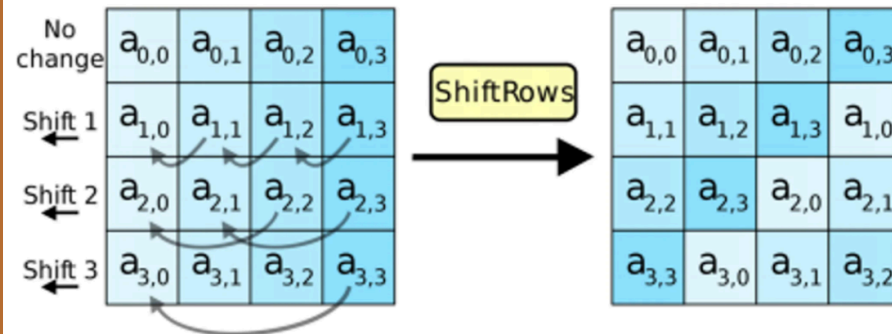
# Advanced Encryption Standard (AES) uses the concept of an S-P network

## Simplified



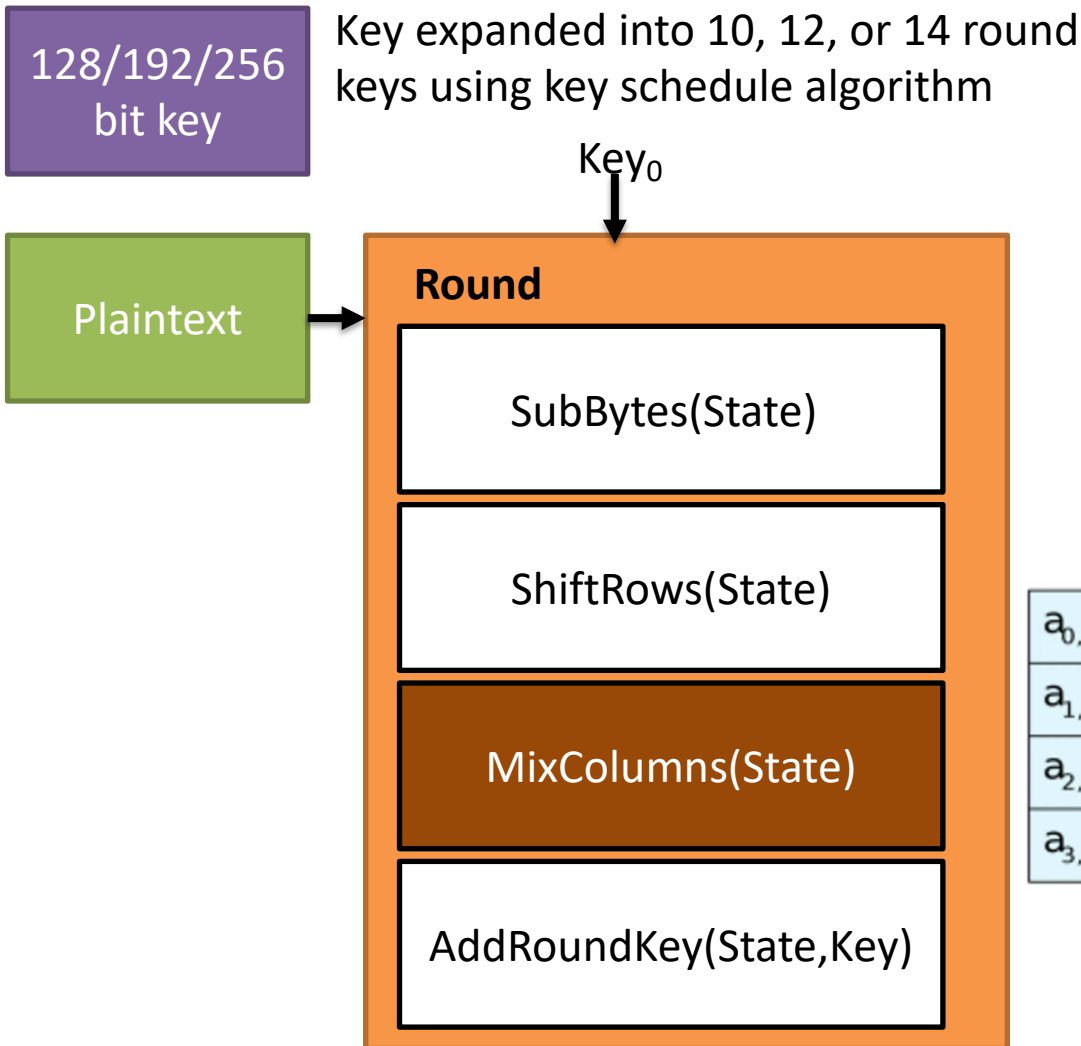
## ShiftRows(State)

- Permute message by shifting row
- Row 0 not changed; other rows shift by increasing amount
- P-box



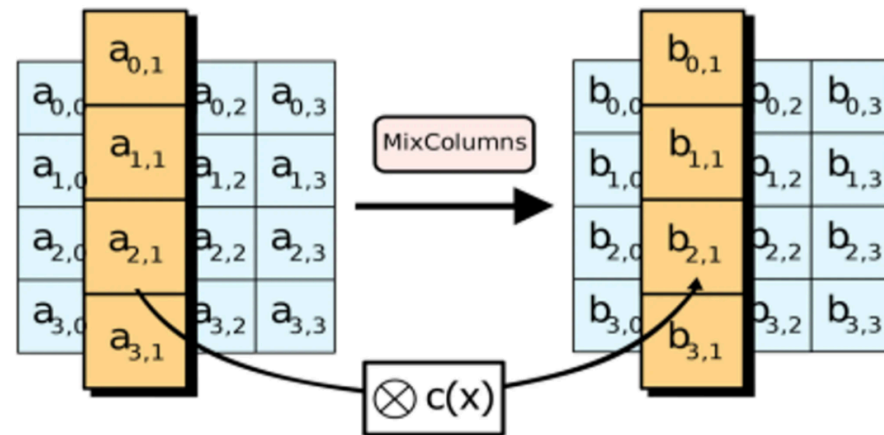
# Advanced Encryption Standard (AES) uses the concept of an S-P network

Simplified



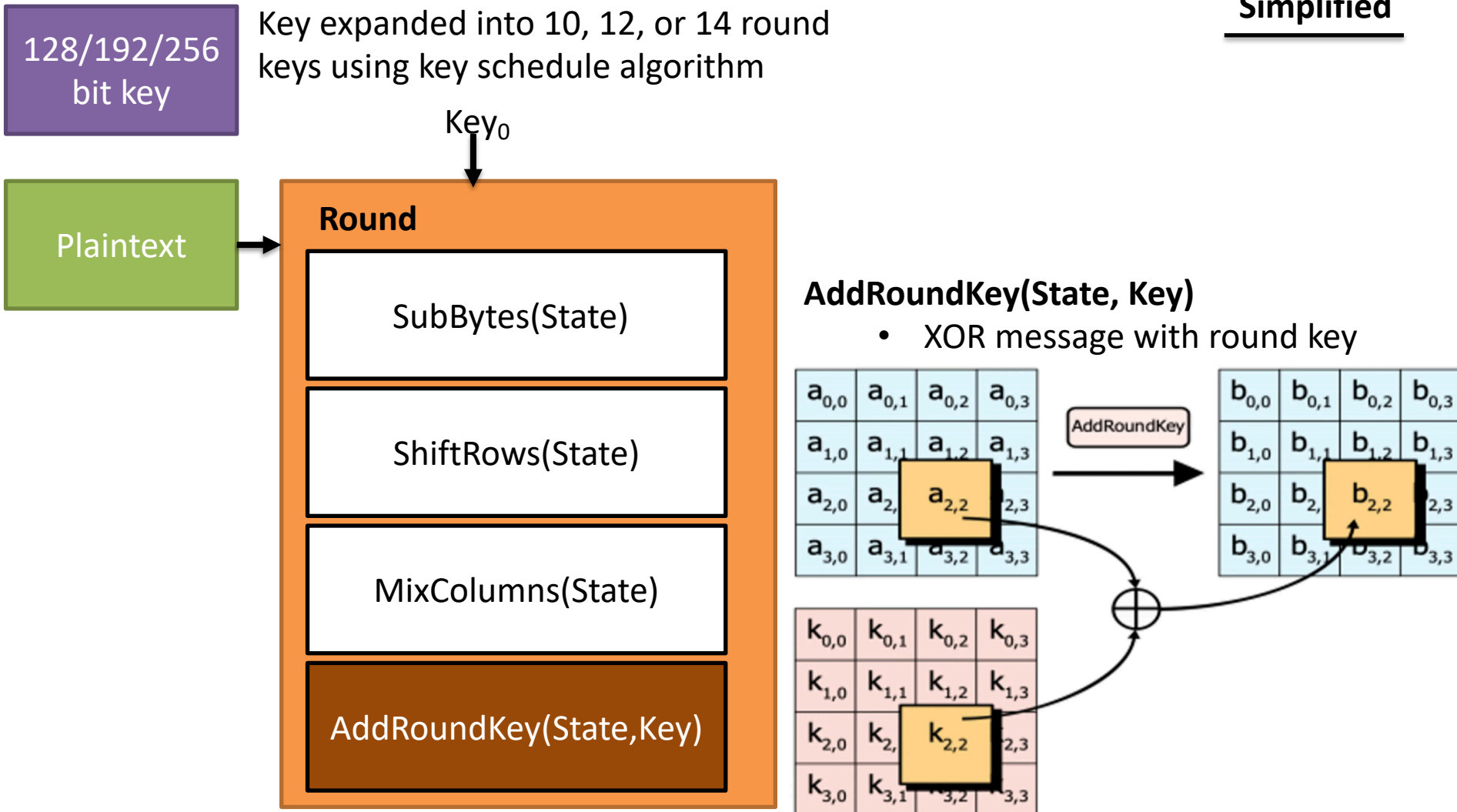
## MixColumns(State)

- Multiple columns by constant matrix

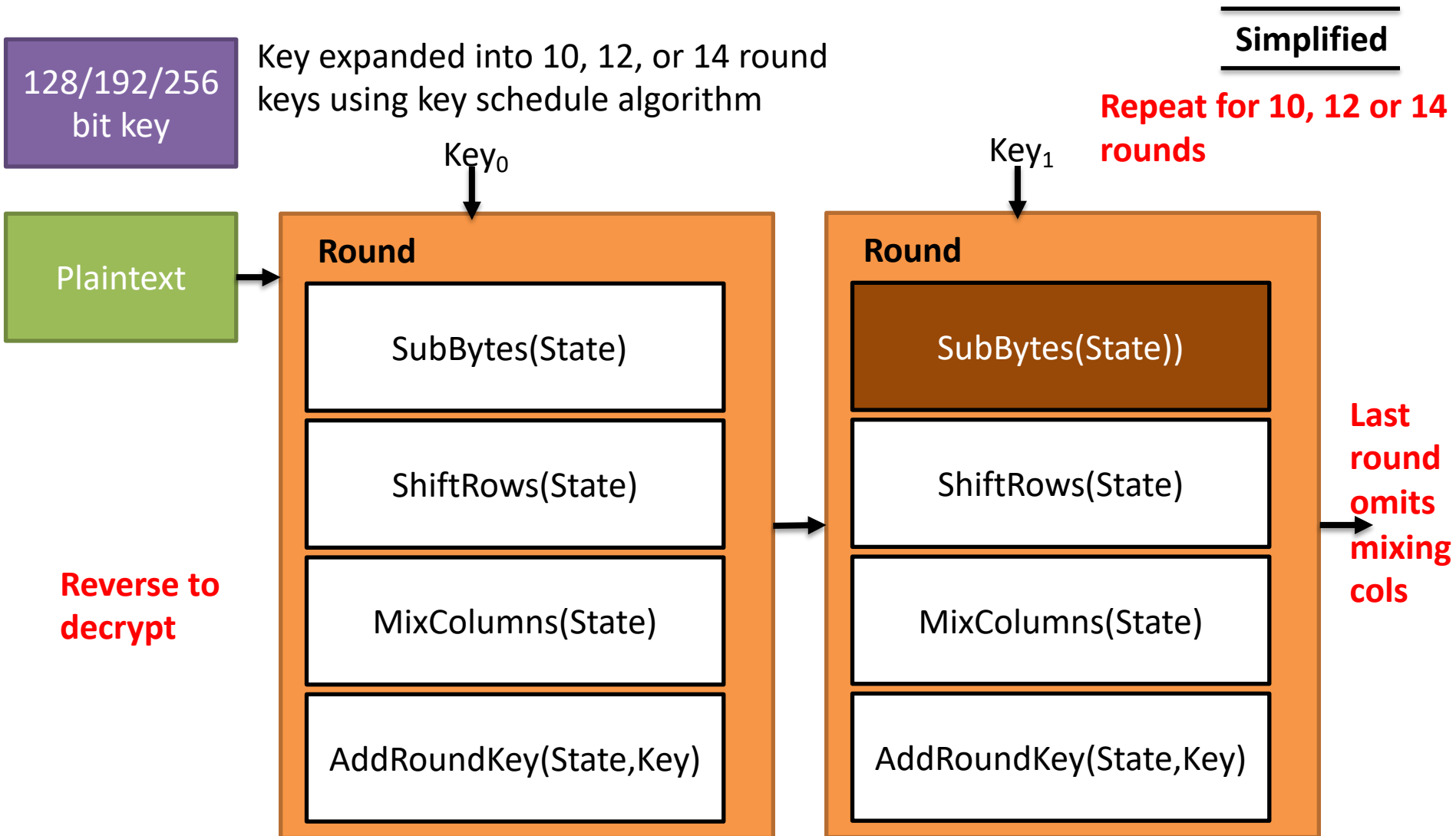


# Advanced Encryption Standard (AES) uses the concept of an S-P network


Simplified



# Advanced Encryption Standard (AES) uses the concept of an S-P network



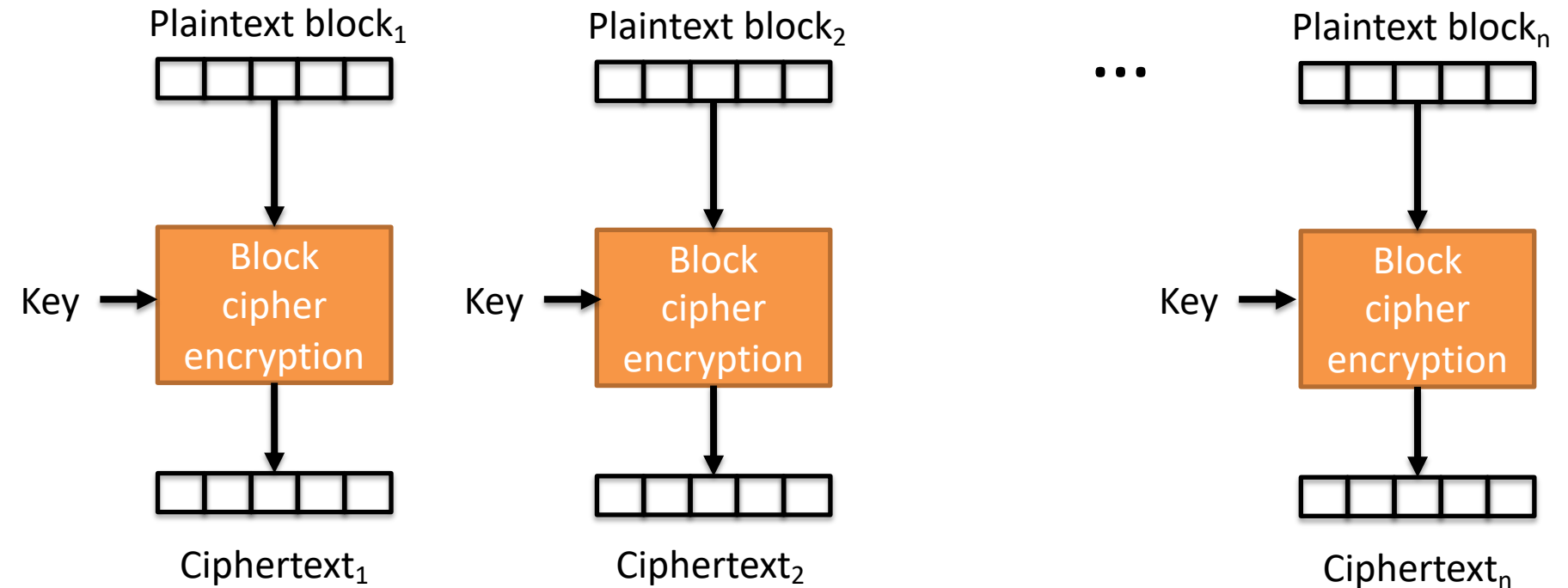
# Agenda

1. Substitution and transposition ciphers
2. Modern symmetric ciphers
-  3. AES Modes
4. Using crypto APIs



# Electronic Code Book (ECB) outputs same ciphertext for same block of plaintext

## Electronic Code Book (ECB)



**Fixed size block of text fed into AES block cipher**

**AES uses 128-bit = 16-byte blocks (wait until there are at least 128 bits to encrypt)**

**AES generates ciphertext for block**

**Ciphertext blocks concatenated together if plaintext is longer than one block (pad if shorter)**

**Reverse process to decrypt**

**Problems?**

# The same plaintext results in the same ciphertext, this could cause problems

Secret document you don't want your competition to see

**Most of this chart is the same (white space)**

**Many blocks will have the same input and produce the same ciphertext**



# The same plaintext results in the same ciphertext, this could cause problems

Secret document you don't want your competition to see

**Most of this chart is the same (white space)**

**Many blocks will have the same input and produce the same ciphertext**



```
head -c 54 SecretNote.bmp > header #save first 54 bytes of header info (.bmp specific data)
openssl enc -aes-128-ecb -e -in SecretNote.bmp -out SecretNote_enc #encrypt with ECB (give pwd)
tail -c +55 SecretNote_enc > body #save encrypted body of picture
cat header body > SecretNote_test #put unencrypted header and encrypted body together
eog SecretNote_test #look at result
```

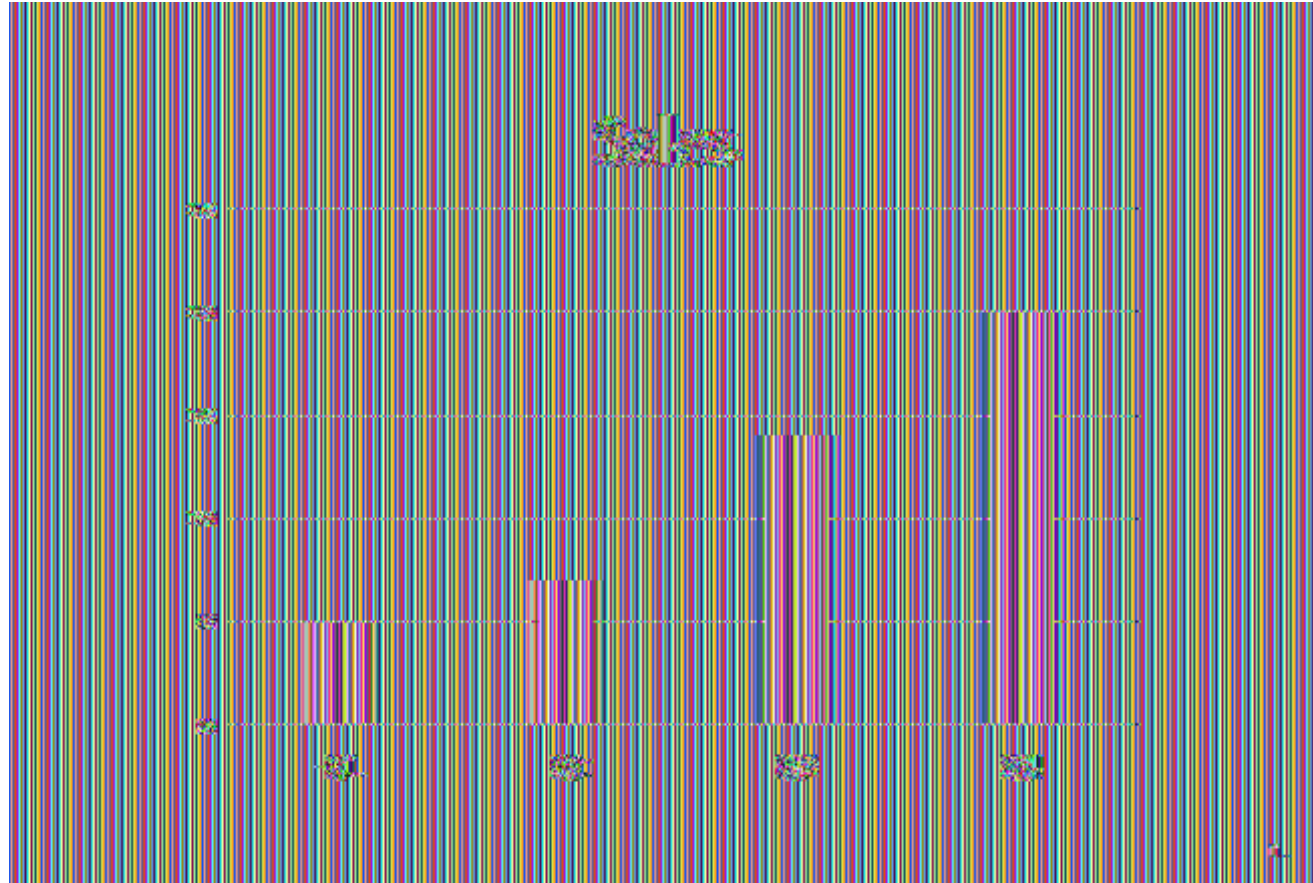
# The same plaintext results in the same ciphertext, this could cause problems

Secret document you don't want your competition to see

**Competitor can still glean that sales are increasing rapidly, even though chart is encrypted**

**One plus, however, is that if a block is lost, other blocks can still be decrypted**

**Generally avoid this mode**



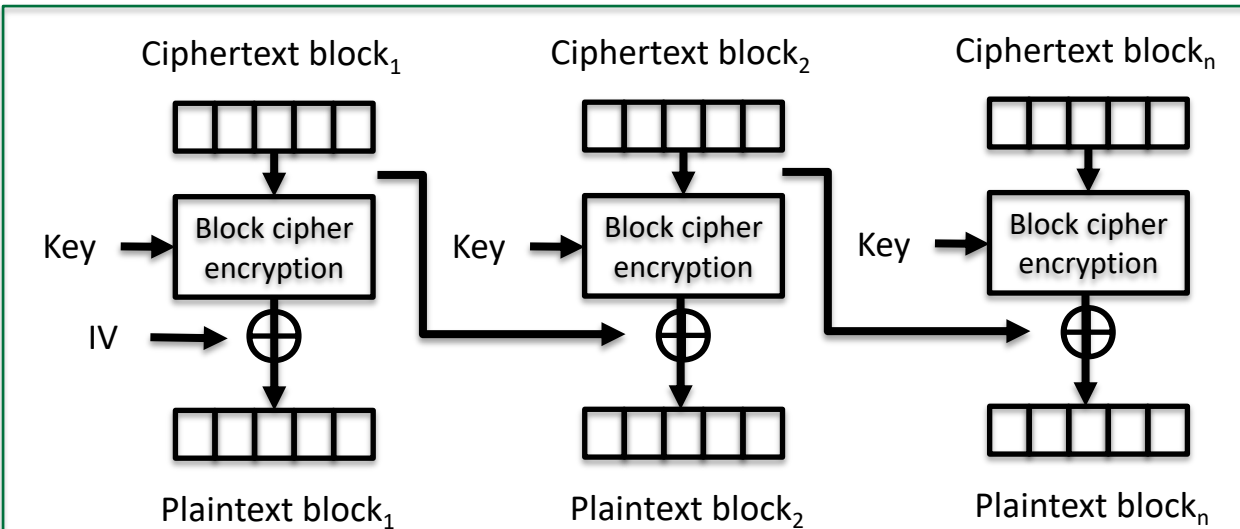
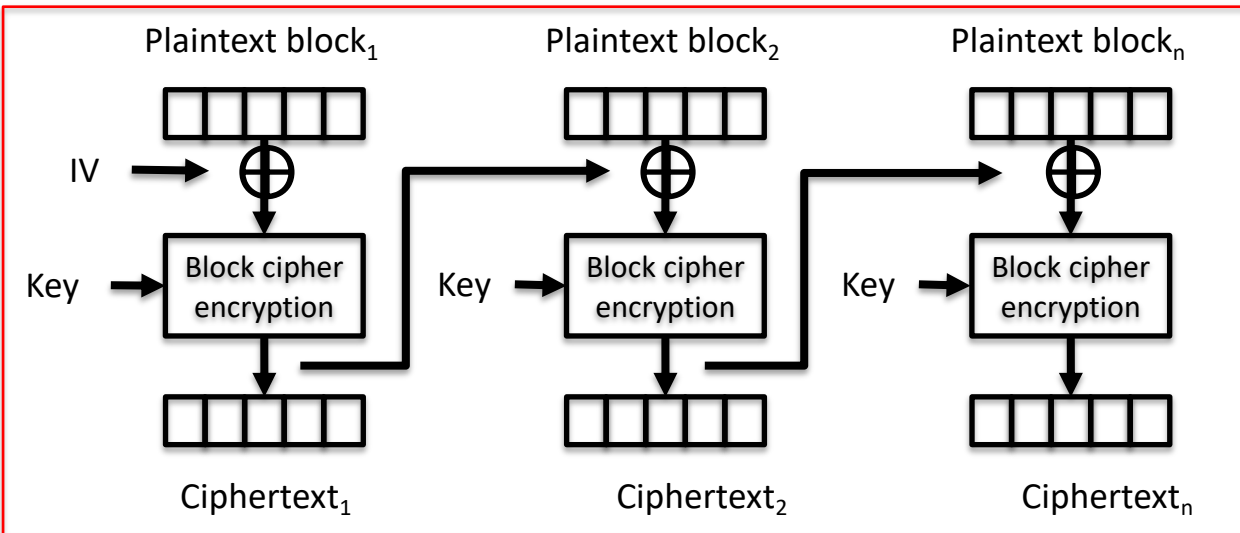
```
head -c 54 SecretNote.bmp > header #save first 54 bytes of header info (.bmp specific data)
openssl enc -aes-128-ecb -e -in SecretNote.bmp -out SecretNote_enc #encrypt with ECB (give pwd)
tail -c +55 SecretNote_enc > body #save encrypted body of picture
cat header body > SecretNote_test #put unencrypted header and encrypted body together
eog SecretNote_test #look at result
```

# Cipher Block Chaining (CBC) uses Initialization Vector (IV) to avoid problems

## Cipher Block Chaining (CBC)

Encrypt

Decrypt



- **CBC most used mode**
- **Initialization Vector (IV) used for first block (not secret)**
- **Each block XOR'ed with previous ciphertext block**
- **Each block depends on all the previous blocks**
- **Change IV every time you send a message to ensure same document gives different results**
- **Decryption is reversed; can do in parallel (but encryption cannot be in parallel)**

# CBC: changing the IV by one bit makes a big difference in ciphertext

## Cipher Block Chaining (CBC)

```
cat short.txt
```

```
This is a secret!
```

```
openssl enc -aes-128-cbc -e -in short.txt -out cipher1.txt -K 001122334455 -iv 1234567890
```

```
openssl enc -aes-128-cbc -e -in short.txt -out cipher2.txt -K 001122334455 -iv 1234567891
```

```
xxd -p cipher1.txt
```

```
b572be26c0f4be246ffbc5e62b7af84f9b7c5f7498ac890488d1118a9a00606b
```

```
xxd -p cipher2.txt
```

```
65cbd9a76773bbab0a7b8ca43345093ab72aba77bcdfb53c89d8d134218fed8d
```

**Same message encrypted with same Key**

**Changing the IV by only one bit changes the output entirely**

**DO NOT reuse the same IV when sending the same message (make unique)!**

**DO NOT make IV predictable (e.g., one greater than last message as we did here!)**

# Padding is needed to match fixed block size

## Padding

- Message size is unlikely to be an integer multiple of block size
- Need to add padding to get last block to be block size
- Must clearly mark where padding begins for decryption
- Pad with the number of bytes (if 7 bytes short of block, pad with 07)

```
#create 9-byte long plaintext file
```

```
$ echo -n "123456789" > plain9.txt
```

```
#encrypt plaintext file using AES with CBC (AES uses 128-bit = 16-byte block size)
```

```
$ openssl enc -aes-128-cbc -e -in plain9.txt -out cipher9.bin -K 0123456 -iv 0987654321
```

```
#check length of ciphertext file
```

```
$ ls -ld cipher9.bin
```

```
-rw-rw-r-- 1 seed seed 16 Aug 26 10:12 cipher9.bin #result is 16 bytes (7 bytes padding)
```

```
#decrypt
```

```
$ openssl enc -aes-128-cbc -d -in cipher9.bin -out plain9a.txt -K 0123456 -iv 0987654321
```

```
$ ls -ld plain9a.txt
```

```
-rw-rw-r-- 1 seed seed 9 Aug 26 10:14 plain9a.txt #plaintext is 9 bytes (padding removed)
```

```
$ cat plain9a.txt
```

```
123456789 #confirm plaintext recovered properly
```

# Padding is needed to match fixed block size

## Padding

- Message size is unlikely to be an integer multiple of block size
- Need to add padding to get last block to be block size
- Must clearly mark where padding begins for decryption
- Pad with the number of bytes (if 7 bytes short of block, pad with 07)

#remember plaintext file

```
$cat plain9.txt
```

```
123456789
```

**What if message length is a multiple of block size?  
Add a whole block of padding!**

#add nopad option to keep padding in decrypted output

```
$ openssl enc -aes-128-cbc -d -in cipher9.txt -out plain9b.txt -K 0123456 -iv 0987654321 -nopad
```

```
$ ls -ld plain9b.txt
```

```
-rw-rw-r-- 1 seed seed 16 Aug 26 10:24 plain9b.txt #output now 16 bytes long (includes padding)
```

#compare original recovered plaintext with plaintext that includes padding

```
$ xxd -g l plain9a.txt
```

```
00000000: 313233343536373839
```

```
123456789. #no padding (9 bytes)
```

```
$ xxd -g l plain9b.txt
```

```
00000000: 313233343536373839070707070707 123456789..... #padding is 7 bytes of 07
```

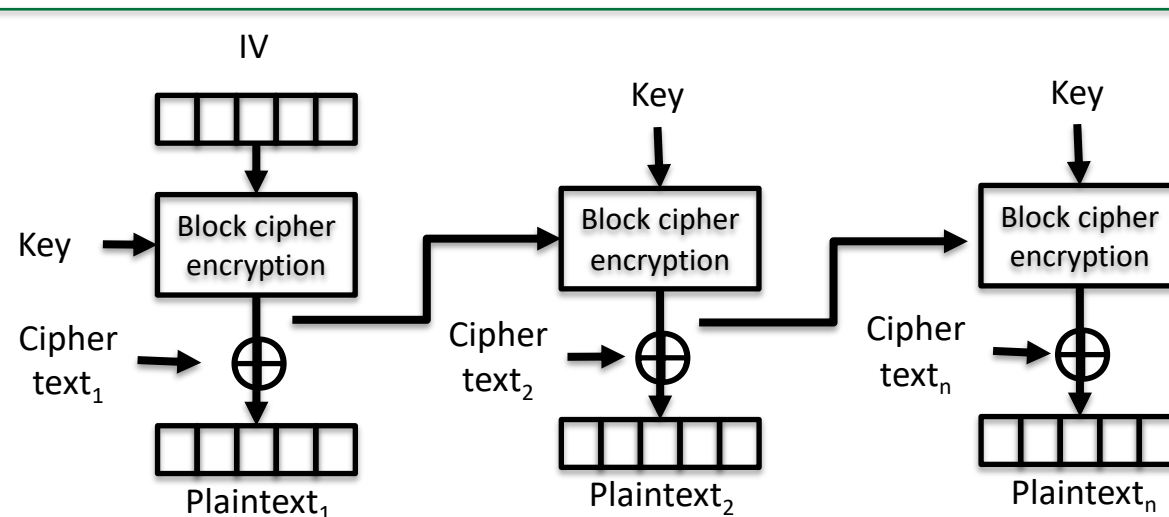
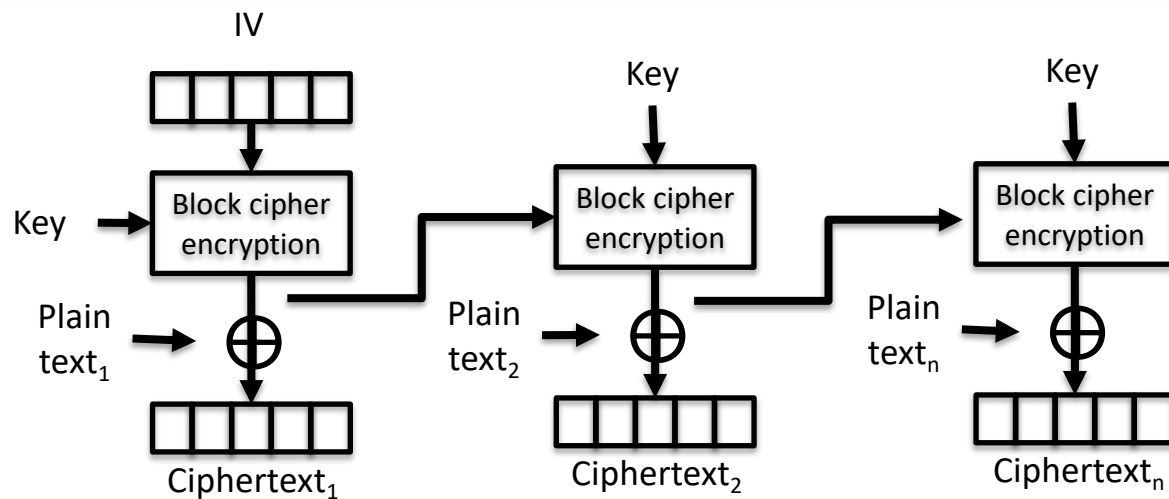


# Output Feedback (OFB) can be used as a stream cipher and can be parallelized

## Output Feedback (OFB)

Encrypt

Decrypt



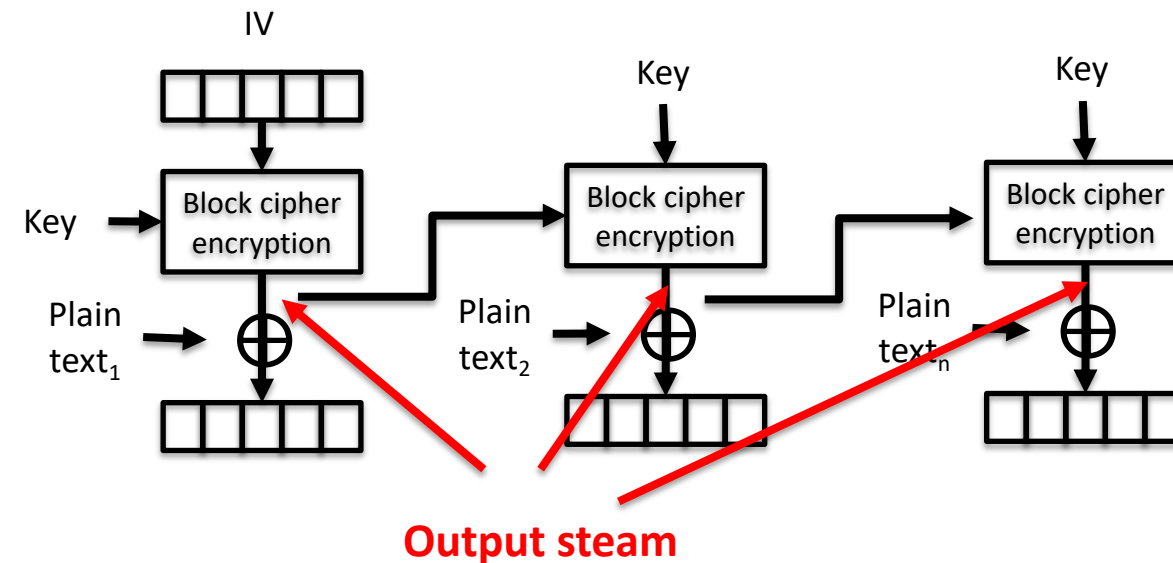
- Can be a stream cipher
  - Plaintext XOR'ed with prev block
  - XOR is bit-wise op
- OFB does not need padding
- Ciphertext has the same number of bytes as plaintext (no padding)
- Encryption can be parallelized if have all plaintext
  - First block based on IV and Key
  - Next blocks can be precomputed
  - Just XOR as plaintext becomes available

# Do not reuse the same IV with the same key on different messages!

## Mistake: reuse IV

- The IV is not a secret so some people assume it is not important
- Sometimes people use all zeros
- Sometimes people reuse the same IV for all messages

### Output Feedback (OFB)



- Eve tricks Alice into encrypting known plaintext  $P_1$
- Eve now has plaintext  $P_1$  and matching ciphertext  $C_1$
- Output stream  $OS = P_1 \oplus C_1$
- If reuse same IV
  - For observed ciphertext  $C_2$  from new message  $P_2$
  - $P_2 = OS \oplus C_2$
- All future messages decrypted even though Eve does not know Key

# Always use a new random IV for each message!

```
#save two plaintext messages
$ echo -n "This is a known message" > P1
$ echo -n "This is TOP secret" > P2
#encrypt both with OFB using same Key and IV
$ openssl enc -aes-128-ofb -e -in P1 -out C1 -K 0123456789 -iv 0000000000
$ openssl enc -aes-128-ofb -e -in P2 -out C2 -K 0123456789 -iv 0000000000
#convert each to hex
$ xxd -p P1
546869732069732061206b6e6f776e206d657373616765
$ xxd -p C1
1809b94f5add4ad8ac26dd4c159167ed06d67777bc6e82
$ xxd -p C2
1809b94f5add4ad89949e60209836abf0ec7
#OS = P1 XOR C1
$ python3 xor.py 546869732069732061206b6e6f776e206d657373616765
1809b94f5add4ad8ac26dd4c159167ed06d67777bc6e82
4c61d03c7ab439f8cd06b6227ae609cd6bb30404dd09e7
#P2 = OS XOR C2
$ python3 xor.py 4c61d03c7ab439f8cd06b6227ae609cd6bb30404dd09e7
1809b94f5add4ad89949e60209836abf0ec7
5468697320697320544f5020736563726574
#Convert hex back to ascii
$ echo -n "5468697320697320544f5020736563726574" | xxd -r -p
This is TOP secret
```

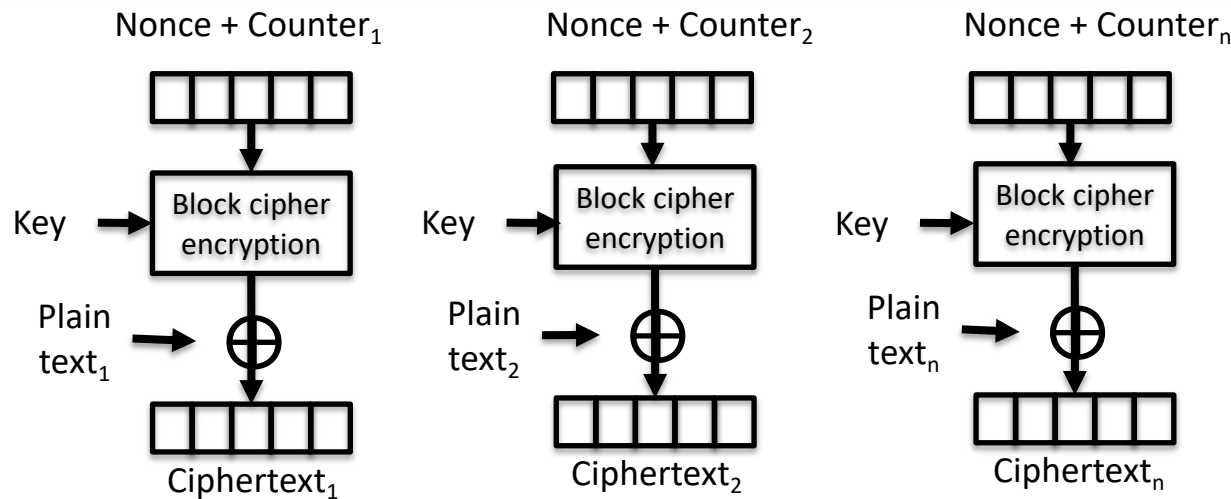
- Eve tricks Alice into encrypting known plaintext  $P_1$
- Eve now has plaintext  $P_1$  and matching ciphertext  $C_1$
- Output stream  $OS = P_1 \oplus C_1$
- If reuse same IV
  - For observed ciphertext  $C_2$  from new message  $P_2$
  - $P_2 = OS \oplus C_2$
- All future messages decrypted even though Eve does not know Key

# Counter mode uses a nonce and a changing counter for each block

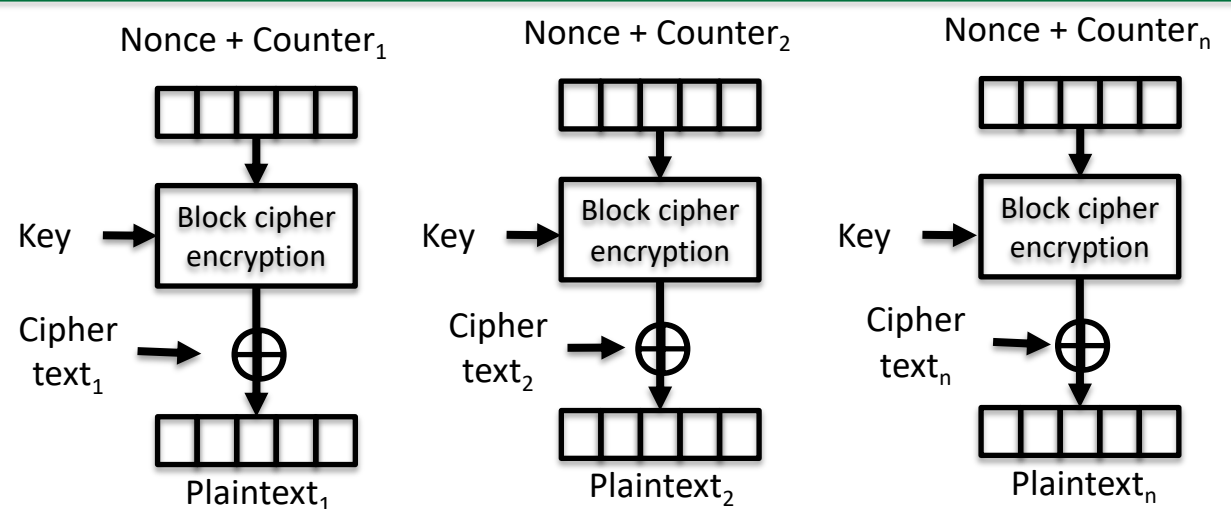
## Counter Mode (CTR)

Encrypt


Decrypt



- **Nonce and initial counter chosen**
- **Counter changes value for each block (normally +1)**
- **Nonce serves same purpose as IV, to ensure each ciphertext is different, even if same plaintext**
- **Blocks do not depend on prior block**
- **Can be parallelized on both encryption and decryption**



# Agenda

1. Substitution and transposition ciphers
2. Modern symmetric ciphers
3. AES Modes
-  4. Using crypto APIs

# You can use Python's crypto libraries in your code

aes.py

Install crypto library from: <https://pycryptodome.readthedocs.io/>

```
from Crypto.Cipher import AES
from Crypto.Util import Padding
```

```
block_size = 16 #AES uses 16-byte blocks
```

```
def encrypt(key, iv, message):
```

```
    #create new cipher using CBC
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)
```

```
    #pad data to be a multiple of 16-byte block size
```

```
    padded = Padding.pad(message, block_size)
```

```
    #create ciphertext
```

```
    ciphertext = cipher.encrypt(padded)
```

```
    return ciphertext
```

```
def decrypt(key, iv, ciphertext):
```

```
    #create new cipher using CBC
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)
```

```
    plaintext = cipher.decrypt(ciphertext)
```

```
    return plaintext
```

This code uses CBC, but is easily converted to other modes

Change AES.MODE\_CBC to AES.MODE\_OFB for OFB (then do not need padding)

Both sender and receiver must know Key and IV (we will address that soon)

