

## Cryptography

# Secret Key Encryption

(“Symmetric Encryption Systems”)

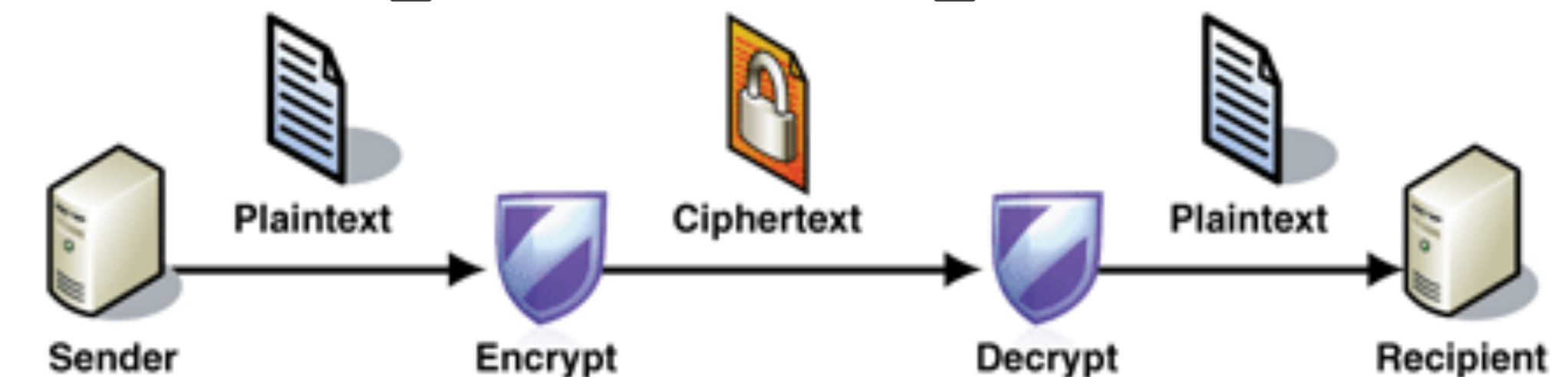
---

Professor Travis Peters  
CSCI 476 - Computer Security  
Spring 2020

Some slides and figures adapted from Wenliang (Kevin) Du's  
**Computer & Internet Security: A Hands-on Approach (2nd Edition)**.  
Thank you Kevin and all of the others that have contributed to the SEED resources!

# Introduction & Overview

- **Encryption** is the process of *encoding a message in such a way that only authorized parties can read the content of the original message*
- History of encryption dates back to 1900 BC!
- Two types of encryption:
  - **secret-key encryption:** same key for encryption and decryption (*this week!*)
  - **public-key encryption:** different keys for encryption and decryption (*later!*)



We study this topic for completeness.

Security education should IMHO include education on crucial topics, including cryptography.

HOWEVER... Beyond these exercises...

**Never Roll Your Own Crypto!**

# Substitution Ciphers

*This Video Covers:*

- Substitution cipher
- Example: using (and breaking) a monoalphabetic substitution cipher

# Substitution Cipher

- **Encryption** is done by replacing units of plaintext with ciphertext, according to a fixed system.

**Example: ROT13 "Key"**

abcdefghijklmnopqrstuvwxyz  
nopqrstuvwxyzabcdefghijklm

- **Units** may be:
  - single letters, pairs of letters, etc.

**Example: Random "Key"**

abcdefghijklmnopqrstuvwxyz  
mcpohfnqrgjuzkltywbvexsai

- **Decryption** simply performs the inverse substitution.
- Two typical substitution ciphers:
  - **Monoalphabetic**: fixed substitution over the entire message
  - **Polyalphabetic**: a number of substitutions at different positions in the message

# Monoalphabetic Substitution Cipher

- We can **encrypt** and **decrypt** a message using the **tr** program:

```
$ cat plaintext  
this is my super secret message
```

```
# The Substitution Cipher "Key"  
abcdefghijklmnopqrstuvwxyz  
mcpohfnqrgjuzkltywbdivexsai
```

```
# "Encryption"  
$ tr 'a-z' 'mcpohfnqrgjuzkltywbdivexsai' < plaintext > ciphertext  
$ cat ciphertext  
dqrbrb za bvthw bhpwhd zhbbmnh
```

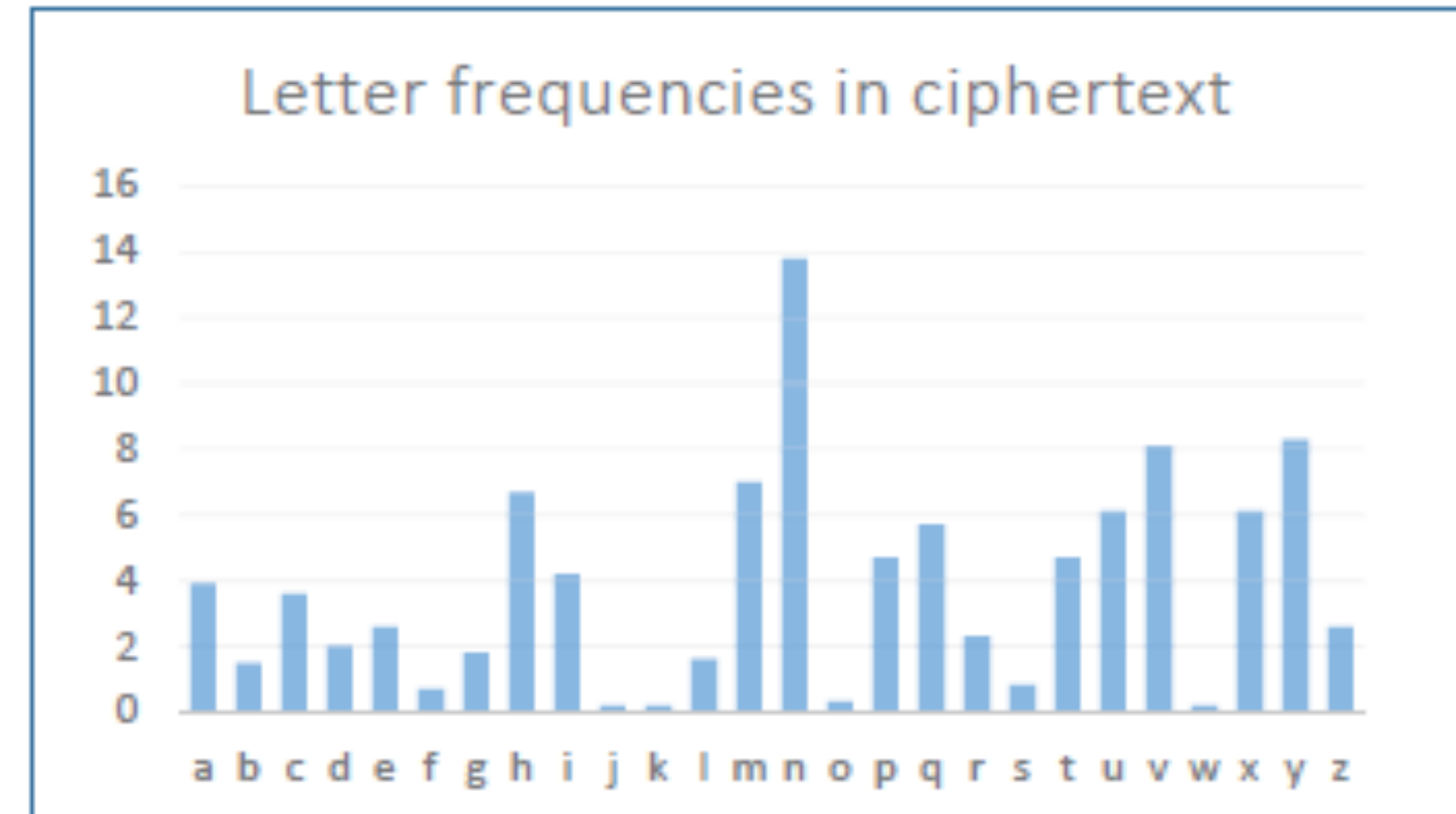
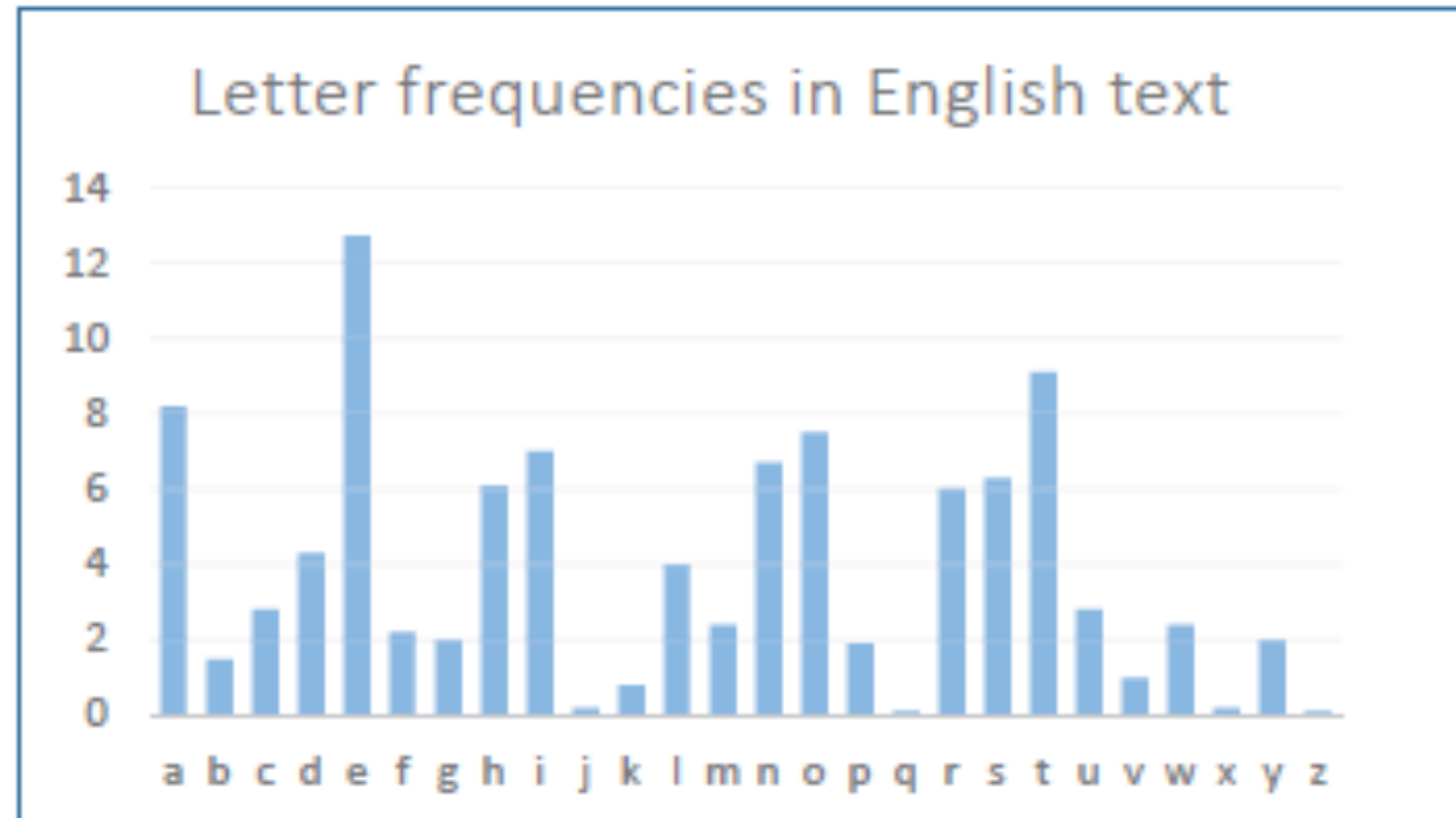
```
# "Decryption"  
$ tr 'mcpohfnqrgjuzkltywbdivexsai' 'a-z' < ciphertext > plaintext_recovered  
$ cat plaintext_recovered  
this is my super secret message
```

*Nice! Are there issues with this type of cipher?  
Surely if a plaintext is large, it is hard to foresee any issues here, right...?*

# Breaking the Monoalphabetic Substitution Cipher

- **Frequency analysis** is the study of the frequency of letters or groups of letters in a ciphertext.
  - Common letters : T, A, E, I, O
  - Common 2-letter combinations (bigrams): TH, HE, IN, ER
  - Common 3-letter combinations (trigrams): THE, AND, and ING

## Example: Frequency Analysis for Letters in an English







# Breaking the Monoalphabetic Substitution Cipher

## Example: Bigram Frequency Analysis

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

Bigram frequency in chiphertext (The top-10 patterns)								
tn	:	77	np	:	50			
yt	:	76	hn	:	45			
nh	:	61	nu	:	44			
nq	:	51	mu	:	42			
vu	:	51	cv	:	42			

## Example: Trigram Frequency Analysis

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

Trigram frequency in chiphertext (The top-10 patterns)								
ytn	:	60	tnh	:	13			
vup	:	26	pyt	:	13			
nhc	:	16	hcv	:	13			
nhn	:	15	tne	:	13			
nuy	:	14	mrc	:	13			



# Breaking the Monoalphabetic Substitution Cipher

## Applying partial mappings...

```
$ tr ntyhqu EHTRSN < ciphertext
THE ENmrcv cvaHmNES lERE v SERmES xb EiEaTRxcEaHvNmavi RXTxR ameHER
cvaHmNES pEfEixeEp vNp zSEp mN THE EvRid Tx cmpTH aENTzRd Tx
eRxTEaT axccERamvi pmeixcvTma vNp cmimTvRd axcczNmavTmxN ENmrcv lvS
mNfENTEp gd THE rERcvN ENrmNEER vRTHzR SaHERgmzS vT THE ENp xb
lxRip lvR m EvRid cxpEiS lERE zSEp axccERamviid bRxc THE EvRid S
vNp vpxeTEp gd cmimTvRd vNp rxfERNcENT SERfmaES xb SEfERvi
```

```
$ tr ntyhquvmxbpz EHTRSNAIOFDU < ciphertext
THE ENIrcA cAaHINES lERE A SERIES OF EiEaTROcEaHANIAi ROTOR aIeHER
cAaHINES DEfEiOeED AND USED IN THE EARid TO cIDTH aENTURd TO
eROTEaT aOccERaIAi DieiOcATia AND cIiITARD aOccUNIAATION ENIrcA lAS
INfENTED gd THE rERcAN ENrINEER ARTHUR SaHERgiUS AT THE END OF
lORid lAR I EARid cODEiS lERE USED aOccERaIAiid FROc THE EARid S
```

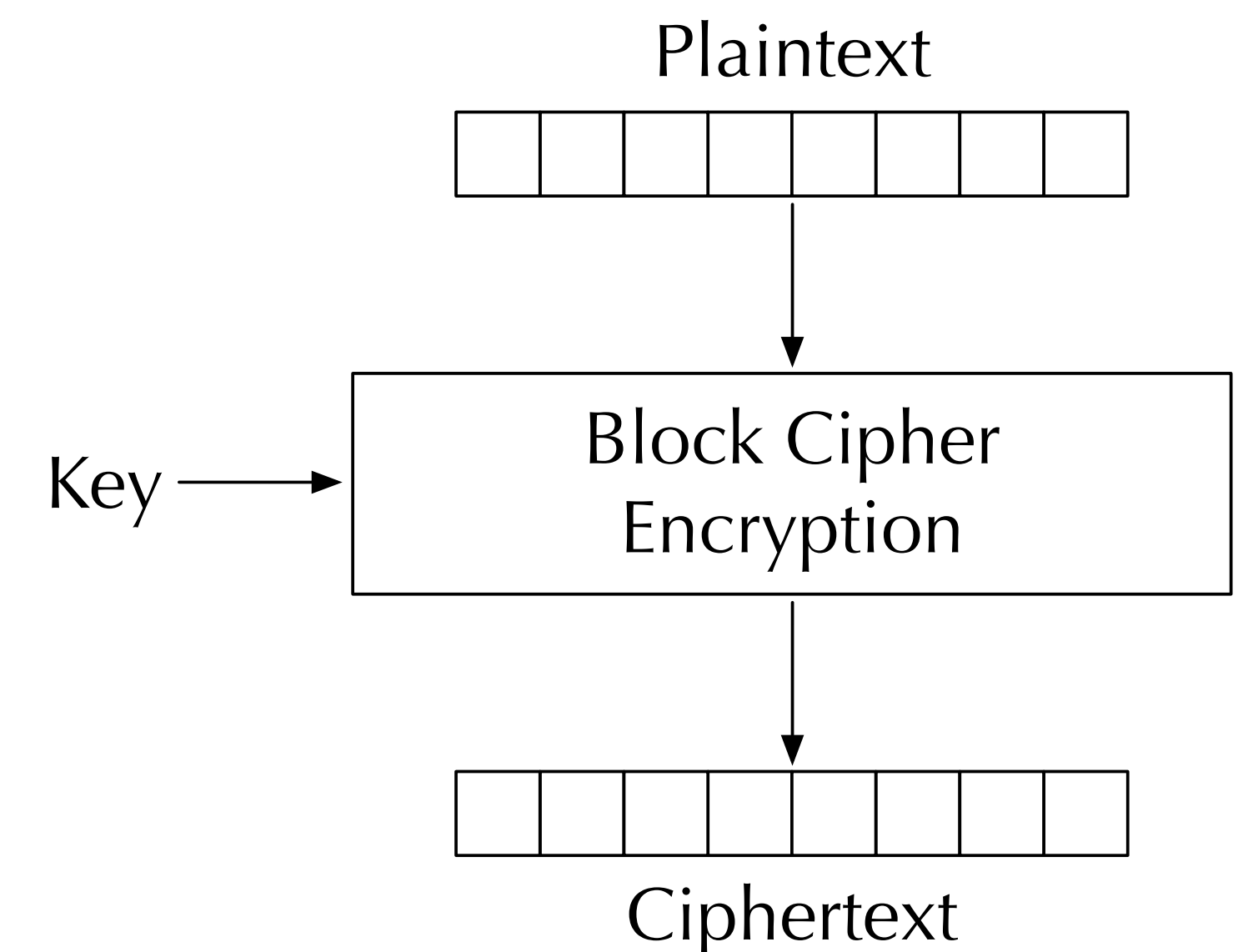
```
$ tr ntyhquvmxbpzfrcei EHTRSNAIOFDUVMPL < ciphertext
THE ENIGMA MAaHINES lERE A SERIES OF ELEaTROMEaHANIAAL ROTOR aIPHER
MAaHINES DEVELOPED AND USED IN THE EARld TO MIDTH aENTURd TO
PROTEaT aOMMERaIAL DIPLOMATia AND MILITARD aOMMUNIAATION ENIGMA lAS
INVENTED gd THE GERMAN ENGINEER ARTHUR SaHERgiUS AT THE END OF
lORLD lAR I EARld MODELS lERE USED aOMMERaIALld FROM THE EARld S
AND ADOPTED gd MILITARD AND GOVERNMENT SERVIAES OF SEVERAL
aOUNTRIES MOST NOTAgld NAWI GERMAND gEFORE AND DURING lORLD lAR II
SEVERAL DIFFERENT ENIGMA MODELS lERE PRODUaED gUT THE GERMAN
MILITARD MODELS HAVING A PLUGgOARD lERE THE MOST aOMPLEk oAPANESE
AND ITALIAN MODELS lERE ALSO IN USE ...
```



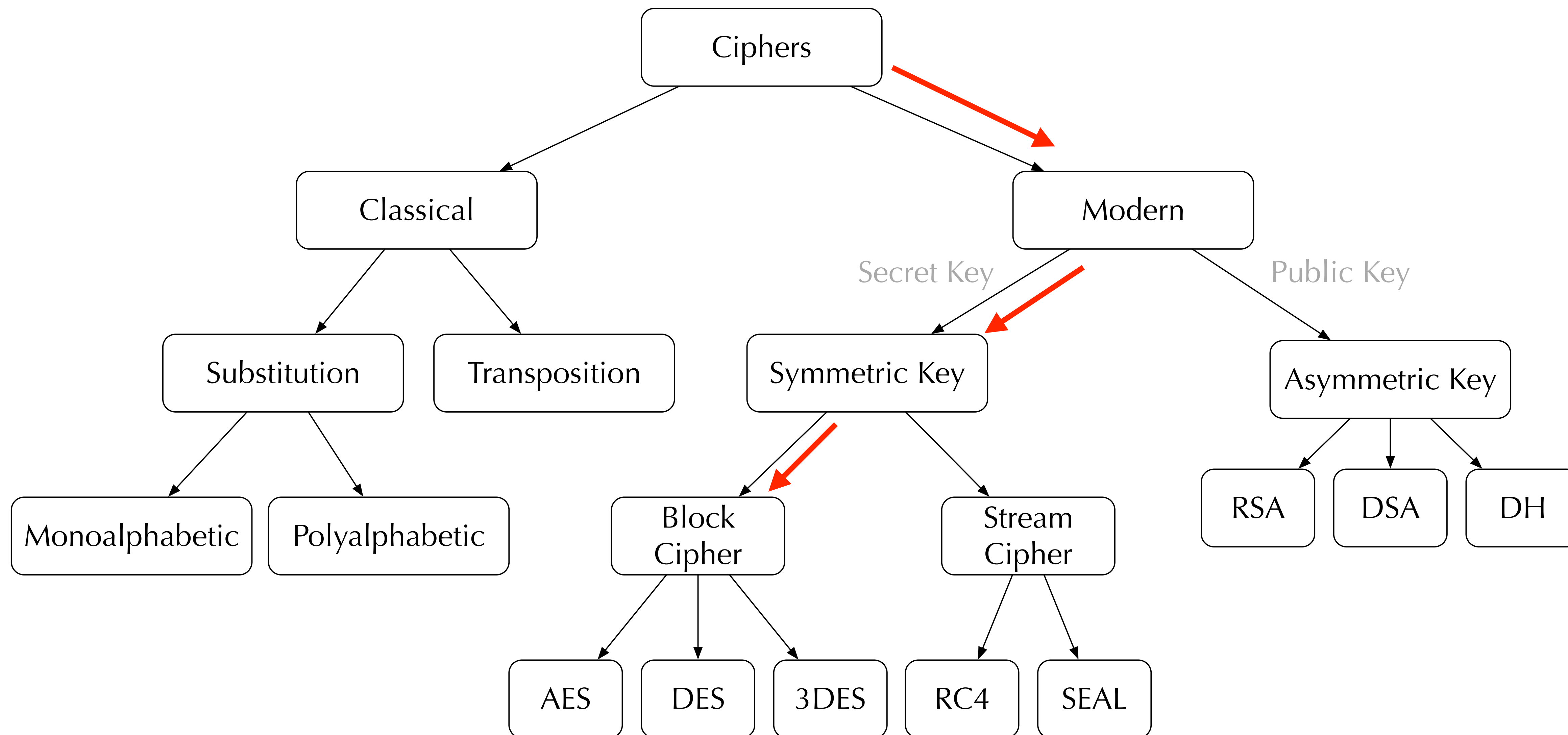
# Block Ciphers and Modes of Operation

*This Video Covers:*

- Block Ciphers: DES, AES
- Modes of Operation: ECB, CBC, CFB, OFB, CTR

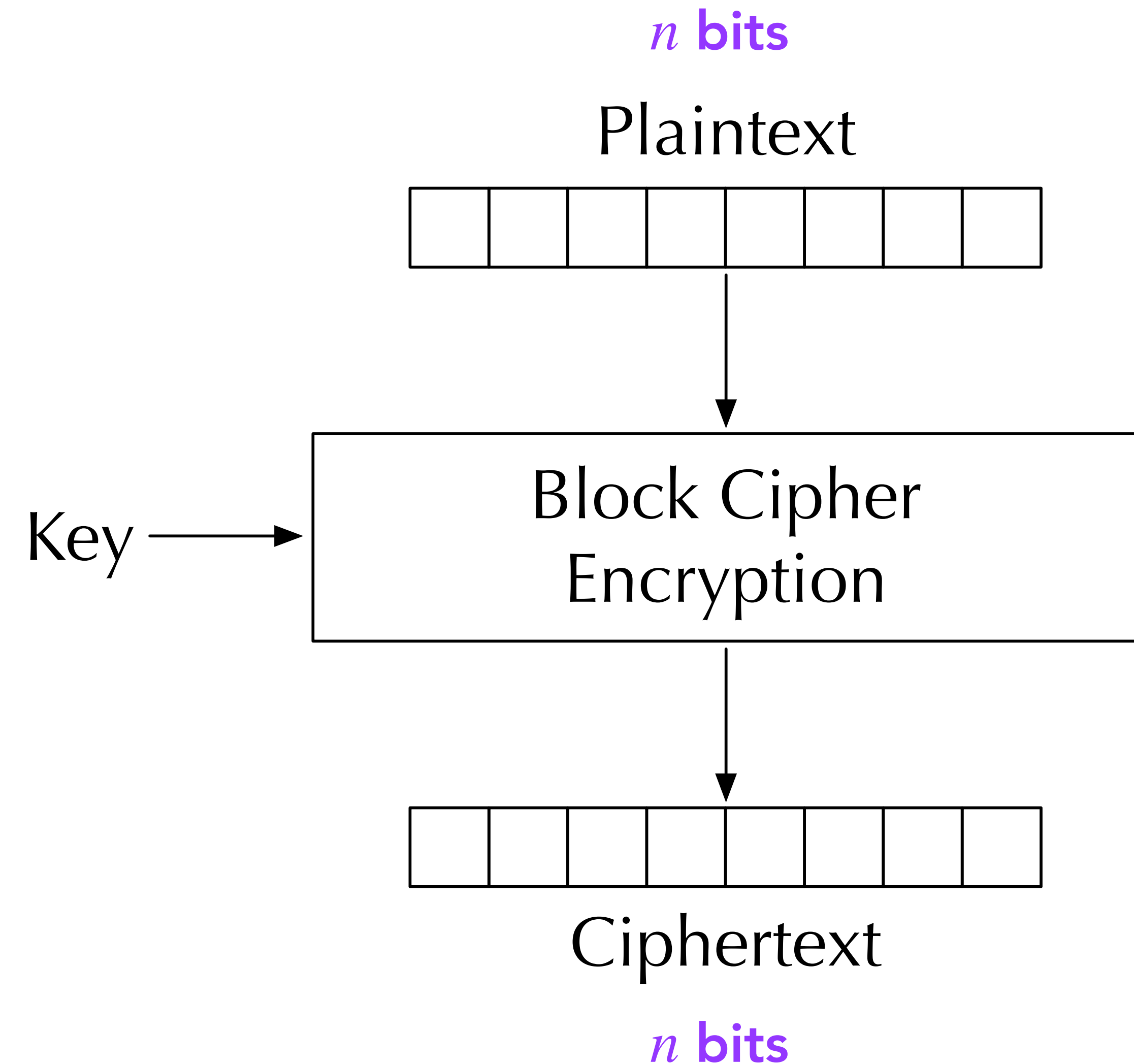


# A (High-Level) Taxonomy of Encryption Methods



# Block Cipher

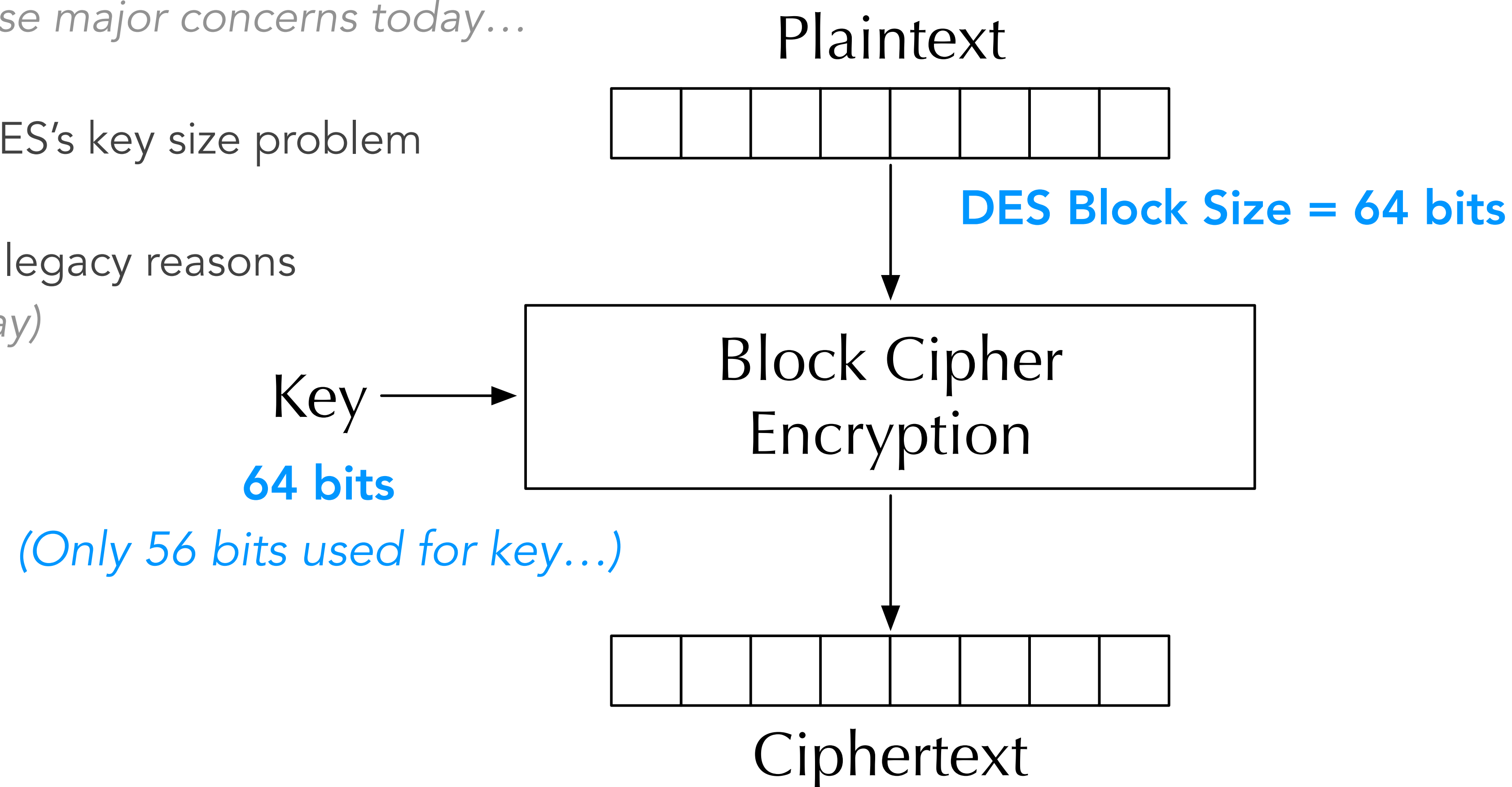
- A symmetric **encryption** which operates on **blocks** of data  
(e.g., 64-bit blocks, 128-bit blocks)
- Break larger messages into small, fixed-size blocks
- Takes one block (**plaintext**) at a time and transforms it into another block of the same length (**ciphertext**) using a **secret key**
- **Decryption** is performed by applying the reverse transformation to ciphertext blocks
- **Important Property:** Even *small* differences in plaintext result in different ciphertexts





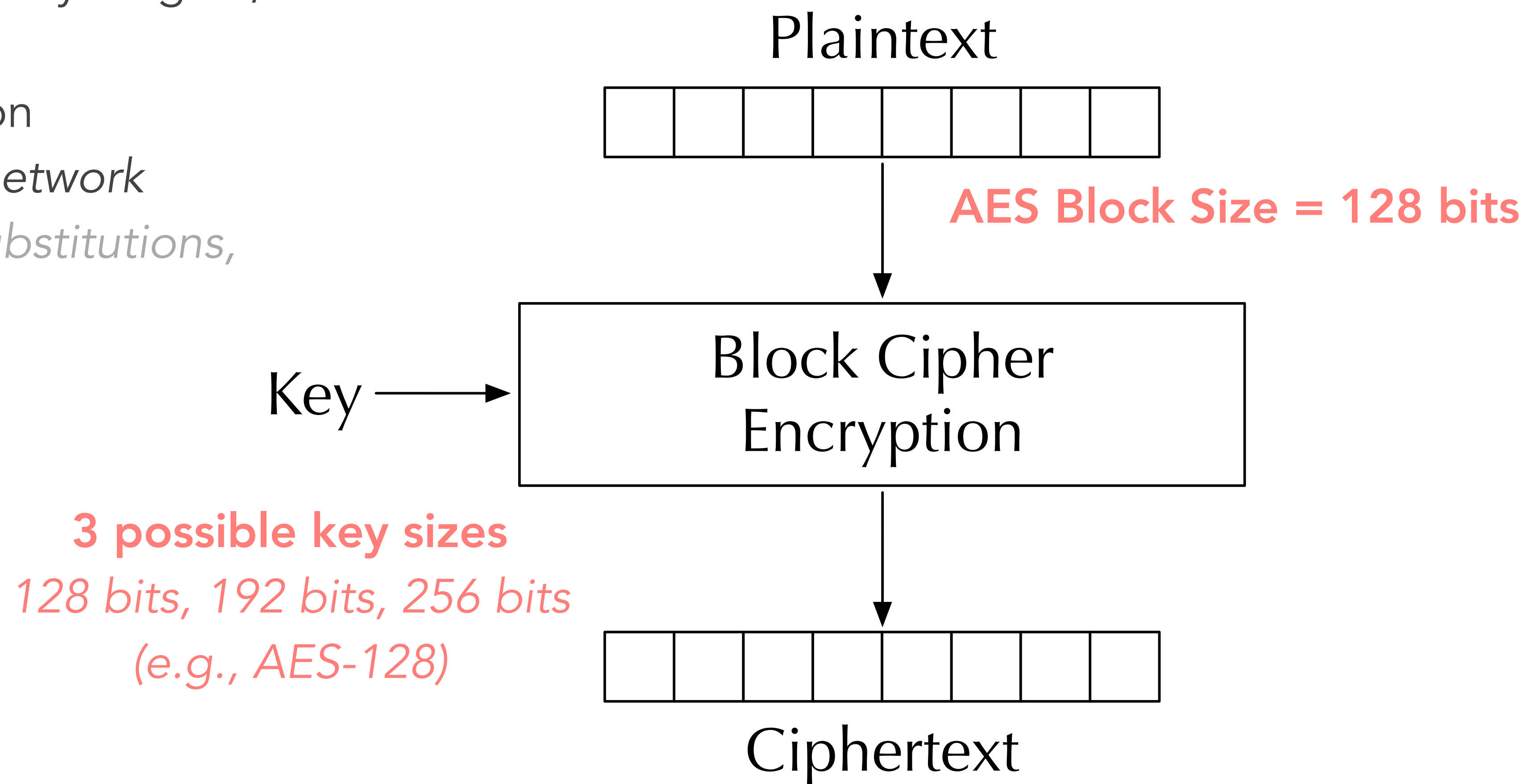
# Data Encryption Standard (DES)

- Based on a *Feistel Cipher*
- Theoretical attacks have been identified...  
*None practical enough to cause major concerns today...*
- *Triple DES (3DES)* can solve DES's key size problem
- Continued support mostly for legacy reasons  
*(Though AES is preferred today)*



# Advanced Encryption Standard (AES)

- Winner of a NIST competition (*no gov. intervention...*)
- Faster than 3DES, variable key lengths, 128-bit blocks
- Underlying design based on *substitution-permutation network* (*multiple rounds of byte substitutions, shifts, mixing, and adding*)



# Encryption Modes

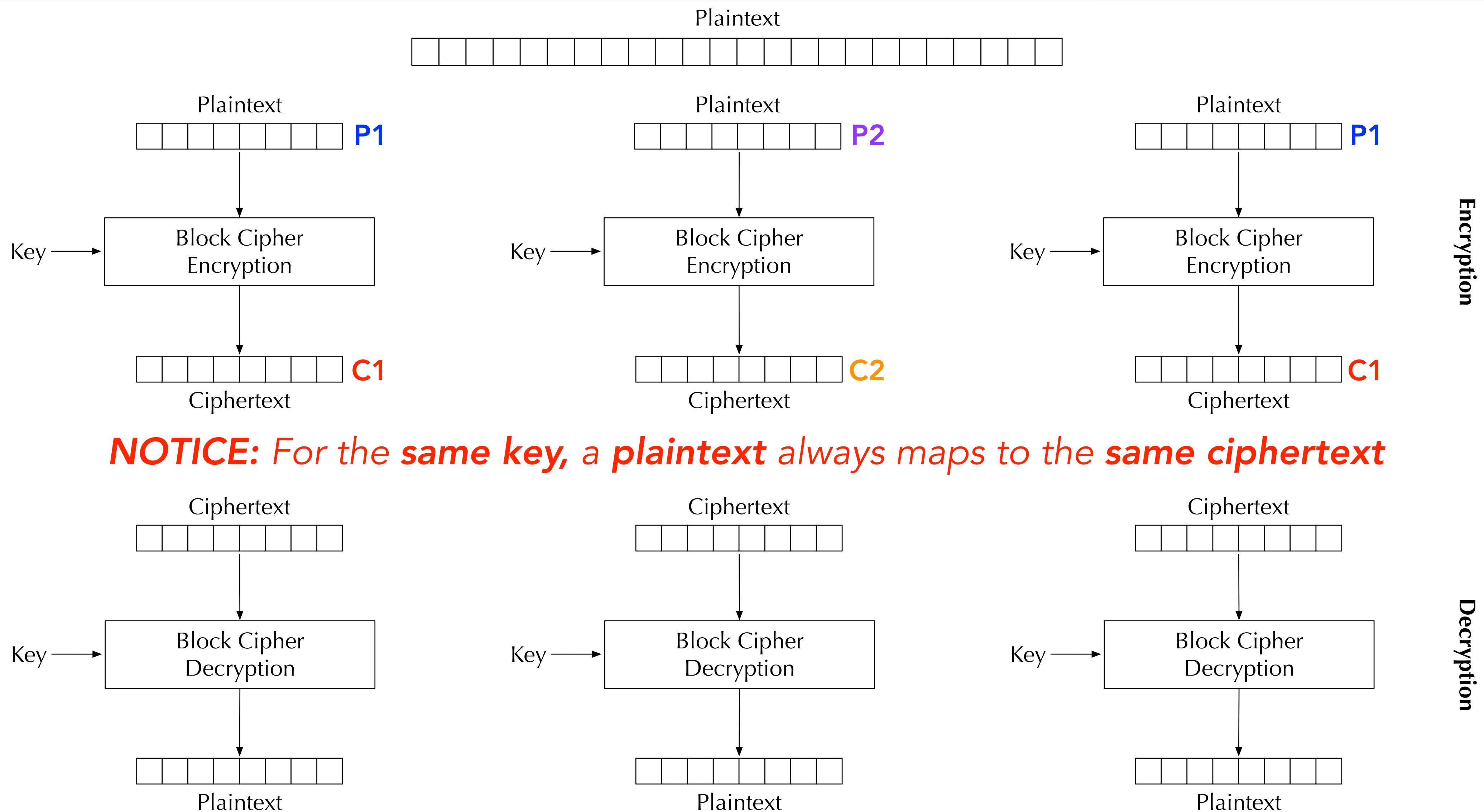
---

- An **encryption mode** or **mode of operation** refers to the many ways to make the input of an encryption algorithm different.
- **Examples:** Electronic Codebook (ECB), Cipher Block Chaining (CBC), Propagating CBC (PCBC), Cipher Feedback (CFB), Output Feedback (OFB), Counter (CTR), ...

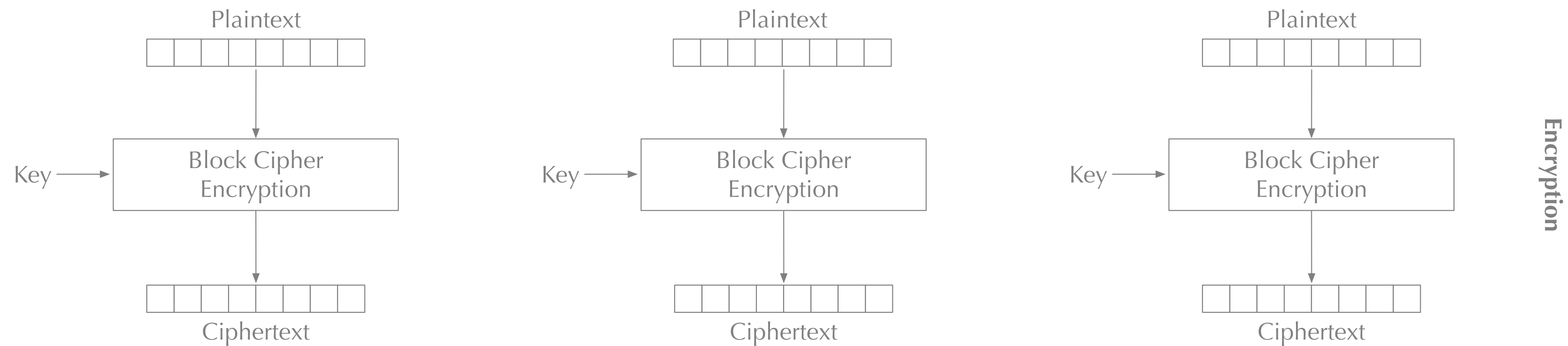
*If we aren't careful about **how** we conduct encryption operations, we may accidentally reveal information about the plaintext...*



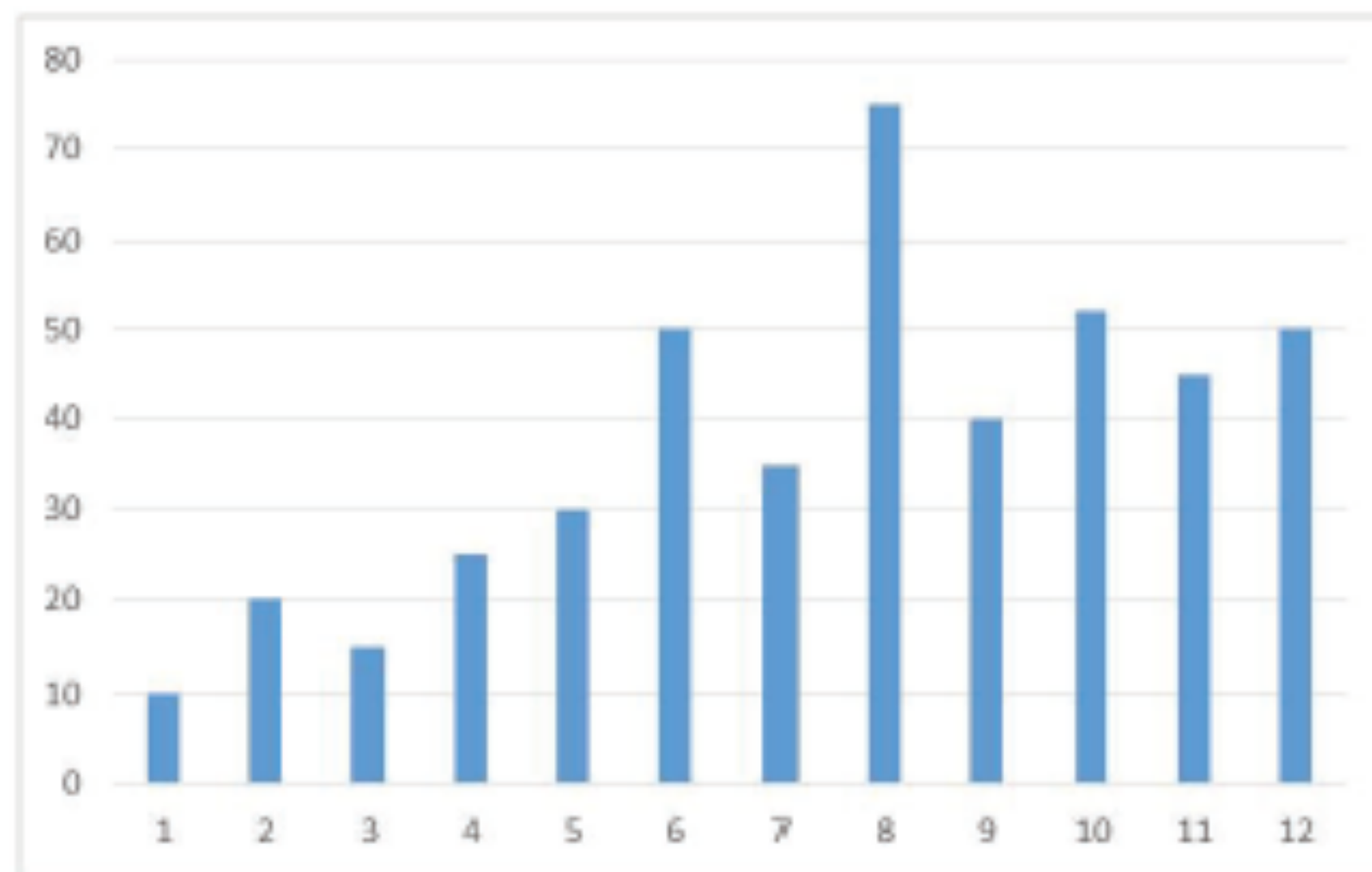
# Electronic Codebook (ECB) Mode



# Electronic Codebook (ECB) Mode *(cont.)*



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



(a) The original image (pic\_original.bmp)



(b) The encrypted image (pic\_encrypted.bmp)

# Electronic Codebook (ECB) Mode *(cont.)*

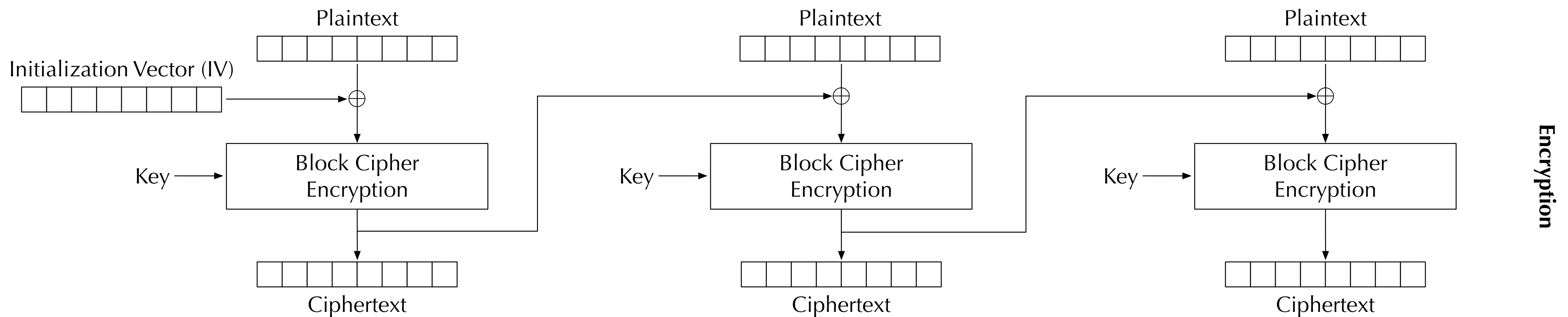
- Using **openssl enc** command:

```
$ openssl enc -aes-128-ecb -e -in plain.txt -out cipher.txt \  
-K 00112233445566778899AABBCCDDEEFF  
  
$ openssl enc -aes-128-ecb -d -in cipher.txt -out plain2.txt \  
-K 00112233445566778899AABBCCDDEEFF
```

- The **-aes-128-ecb** option specifies 128-bit (key size) AES algorithm w/ ECB mode
- The **-e** option indicates encryption
- The **-d** option indicate decryption
- The **-K** option is used to specify the encryption/decryption key



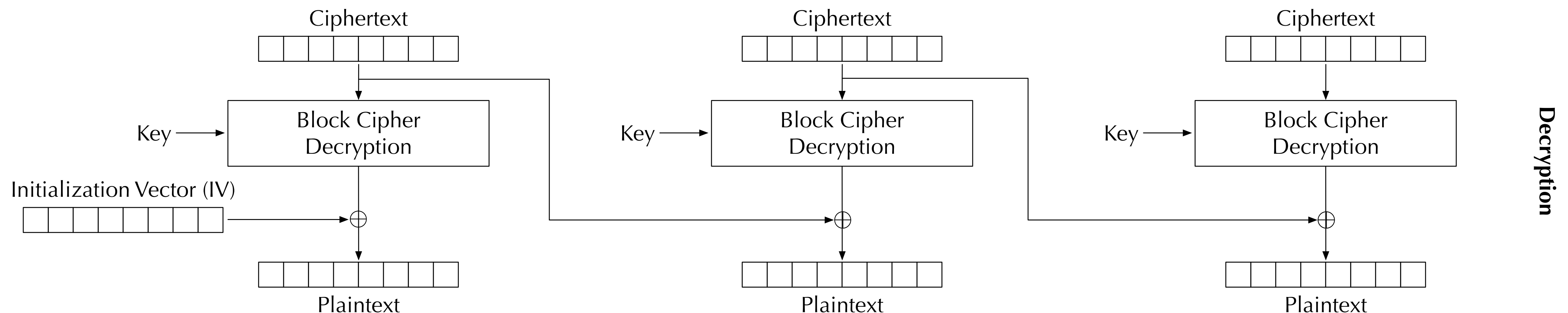
# Cipher Block Chaining (CBC) Mode



- CBC introduces **block dependency**; i.e.,  $C_i = E_K(P_i \oplus C_{i-1})$
- The main purpose of the **initialization vector (IV)** is to ensure that even if two plaintexts are identical, their ciphertexts are still different, because different IVs will be used.
- Encryption **cannot** be parallelized
- Decryption **can** be parallelized; i.e.,  $P_i = D_K(C_i) \oplus C_{i-1}$

# Cipher Block Chaining (CBC) Mode (cont.)

- CBC introduces **block dependency**; i.e.,  $C_i = E_K(P_i \oplus C_{i-1})$
- The main purpose of the **initialization vector (IV)** is to ensure that even if two plaintexts are identical, their ciphertexts are still different, because different IVs will be used.
- Encryption **cannot** be parallelized
- Decryption **can** be parallelized; i.e.,  $P_i = D_K(C_i) \oplus C_{i-1}$



# Cipher Block Chaining (CBC) Mode *(cont.)*

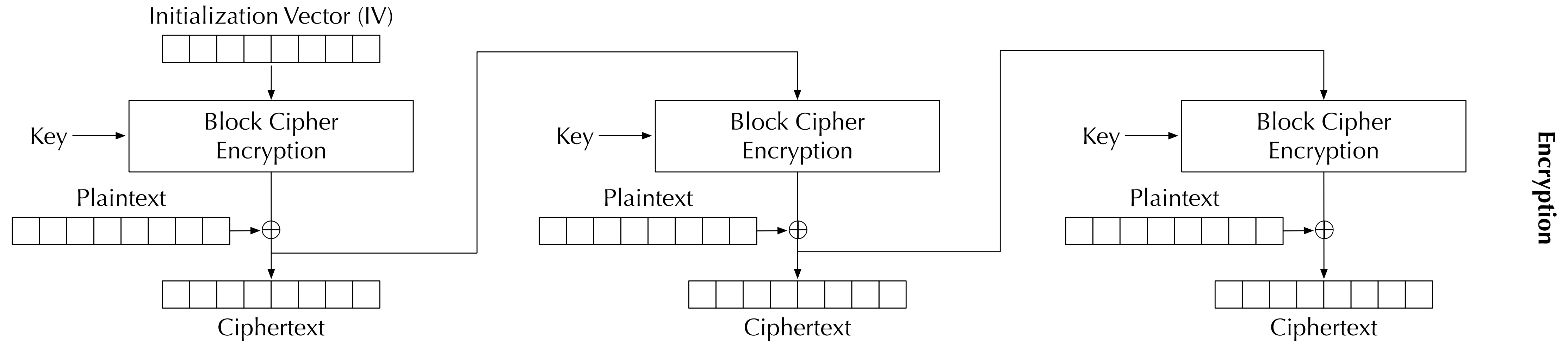
- Using **openssl enc** command to encrypt the same plaintext, with the same key, but a different IV:

```
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.txt \  
    -K 00112233445566778899AABBCCDDEEFF \  
    -iv 000102030405060708090A0B0C0D0E0F  
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.txt \  
    -K 00112233445566778899AABBCCDDEEFF \  
    -iv 000102030405060708090A0B0C0D0E0E  
$ xxd -p cipher1.txt  
...  
$ xxd -p cipher2.txt  
...
```

- The **-aes-128-cbc** option specifies 128-bit (key size) AES algorithm w/ CBC mode
- The **-e** option indicates encryption
- The **-K** option is used to specify the encryption/decryption key
- The **-iv** option is used to specify the initialization vector (IV)

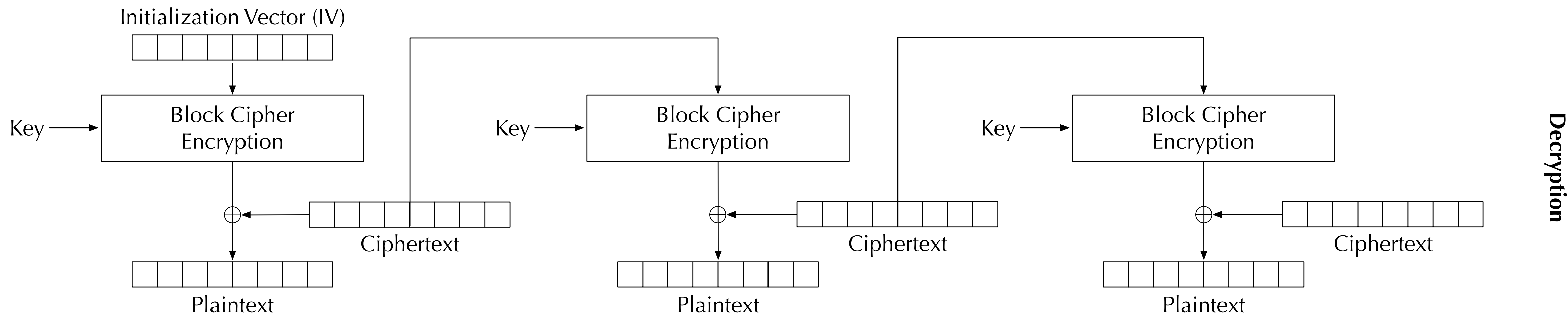


# Cipher Feedback (CFB) Mode



- 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*
- Decryption using the CFB mode *can* be parallelized

# Cipher Feedback (CFB) Mode *(cont.)*



- 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*
- Decryption using the CFB mode *can* be parallelized

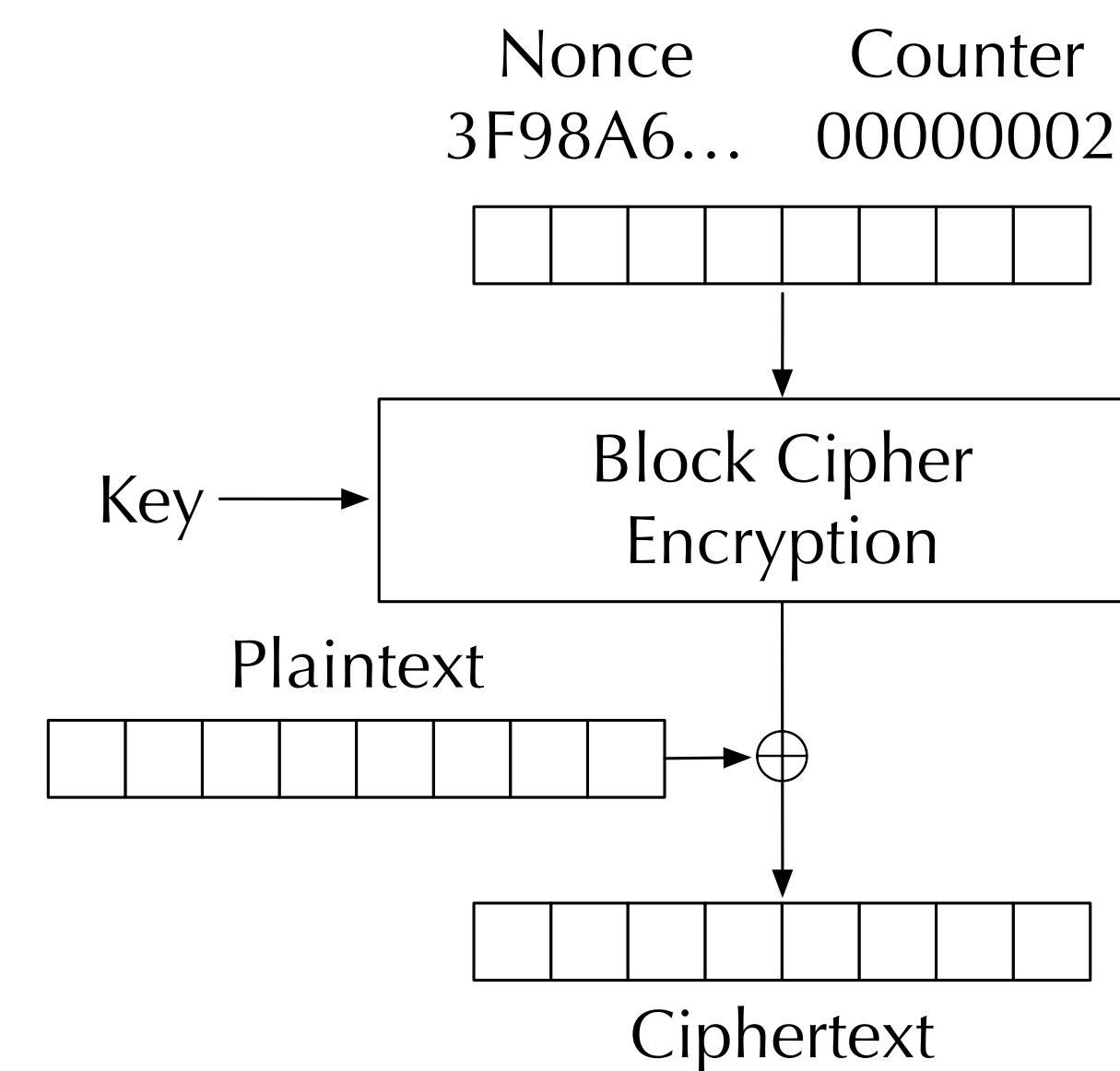
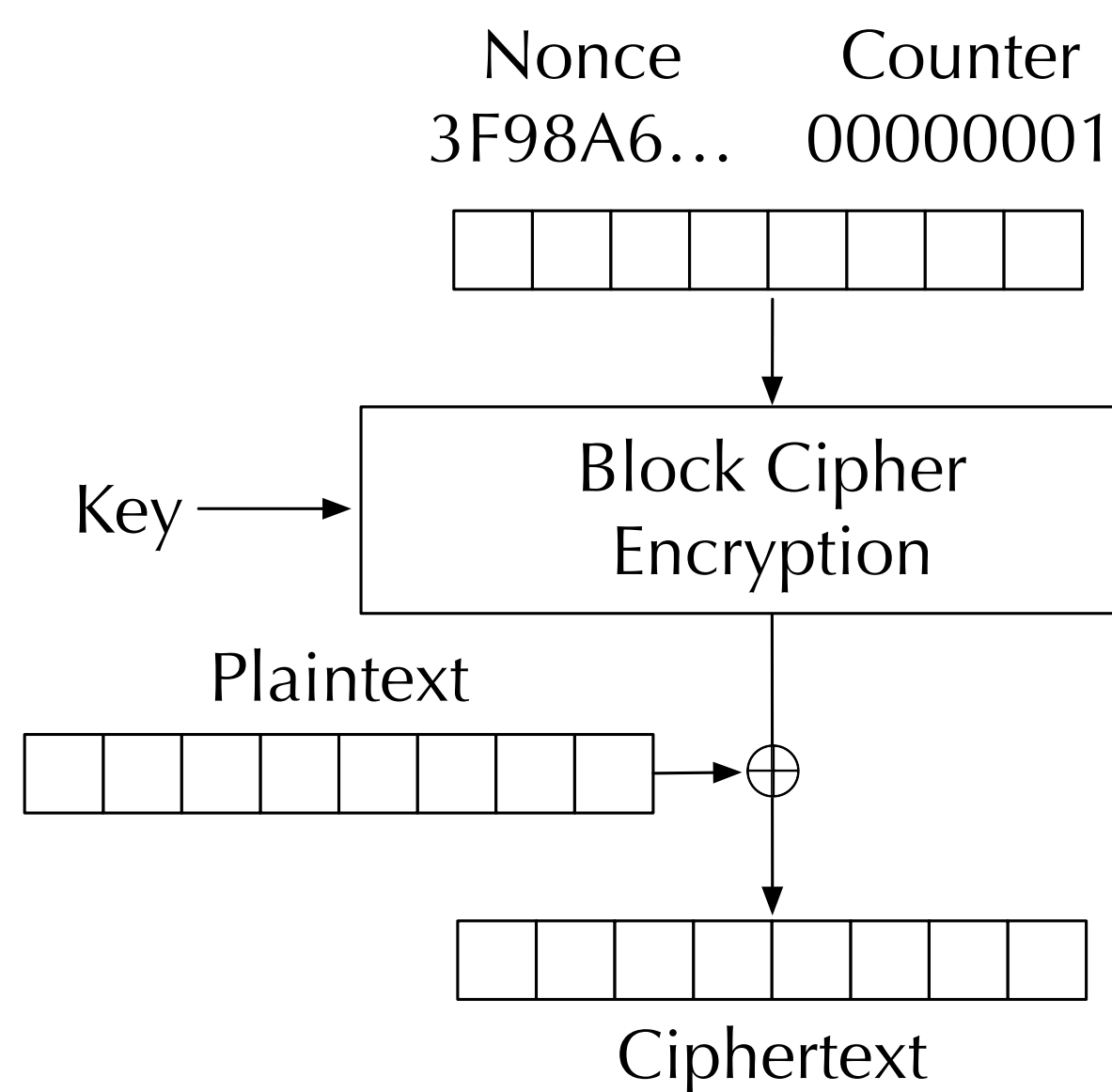
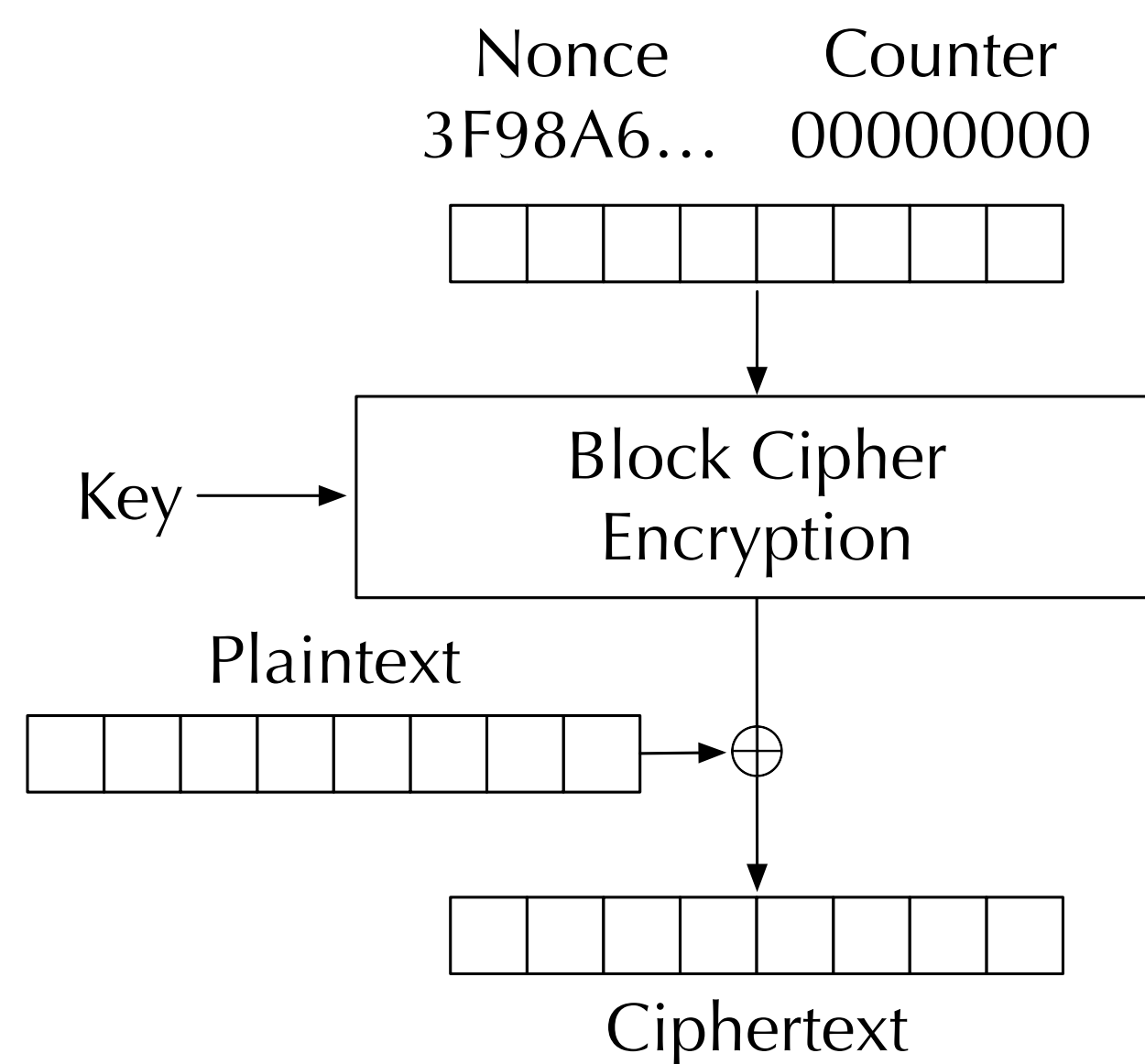
# Comparing Encryption with CBC and CFB

```
$ echo -n 'abcdefghijklmnopqrstuvwxyz' > plain.txt
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.txt \
    -K 00112233445566778899AABBCCDDEEFF \
    -iv 000102030405060708090A0B0C0D0E0F
$ openssl enc -aes-128-cfb -e -in plain.txt -out cipher2.txt \
    -K 00112233445566778899AABBCCDDEEFF \
    -iv 000102030405060708090A0B0C0D0E0F
$ ls -l *.txt
-rw-rw-r-- 1 seed seed 32 Apr  1 20:02 cipher1.txt
-rw-rw-r-- 1 seed seed 26 Apr  1 20:02 cipher2.txt
-rw-rw-r-- 1 seed seed 26 Apr  1 20:02 plain.txt
```

- Plaintext size is 26 bytes
- **CBC mode:** ciphertext is 32 bytes due to padding
- **CFB mode:** ciphertext size is the same as the plaintext (26 bytes)

# 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 **nonce** (*same idea as the IV*)
- Both encryption and decryption can be parallelized
- The key stream in the CTR mode can be calculated in parallel during the encryption

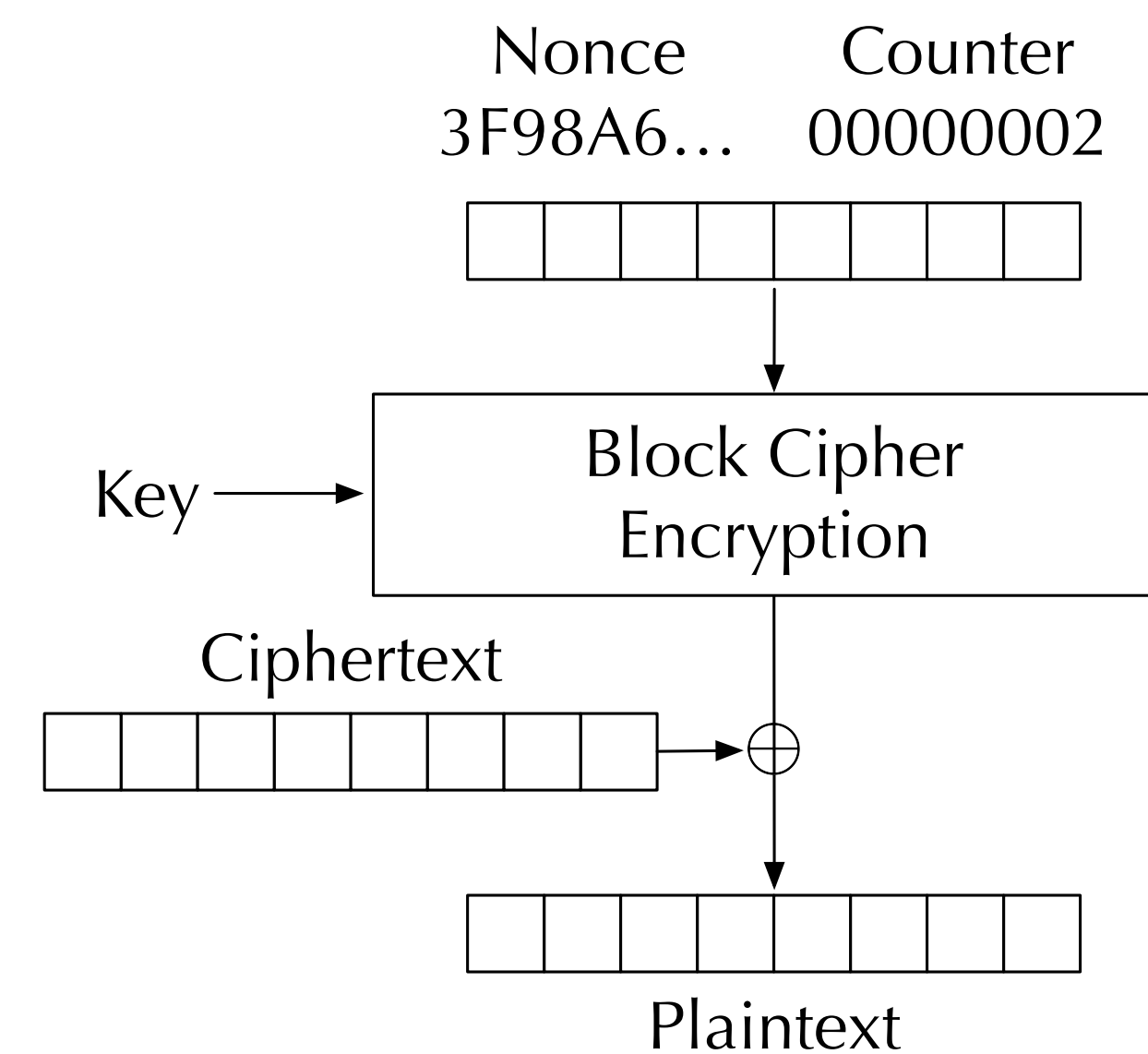
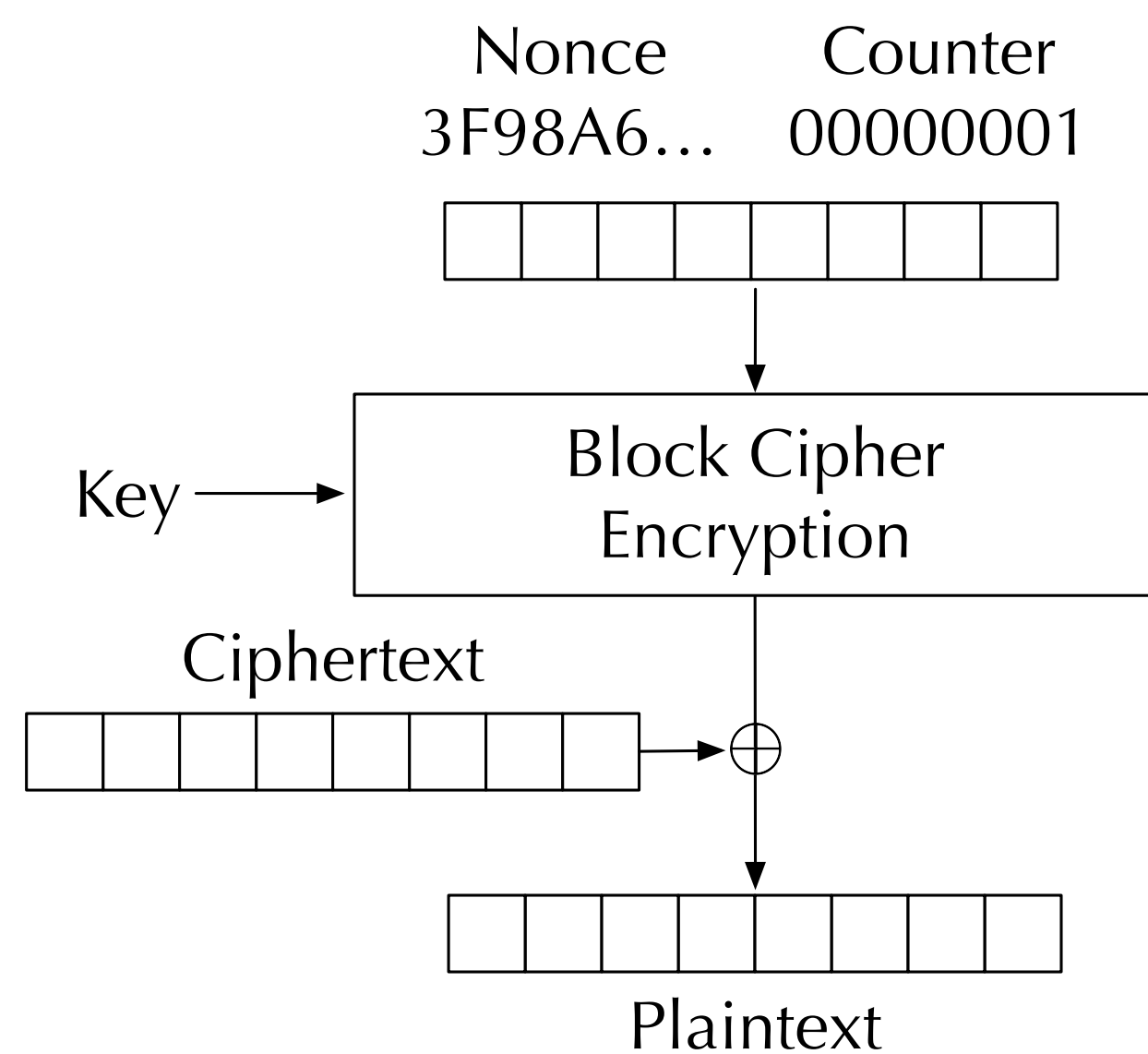
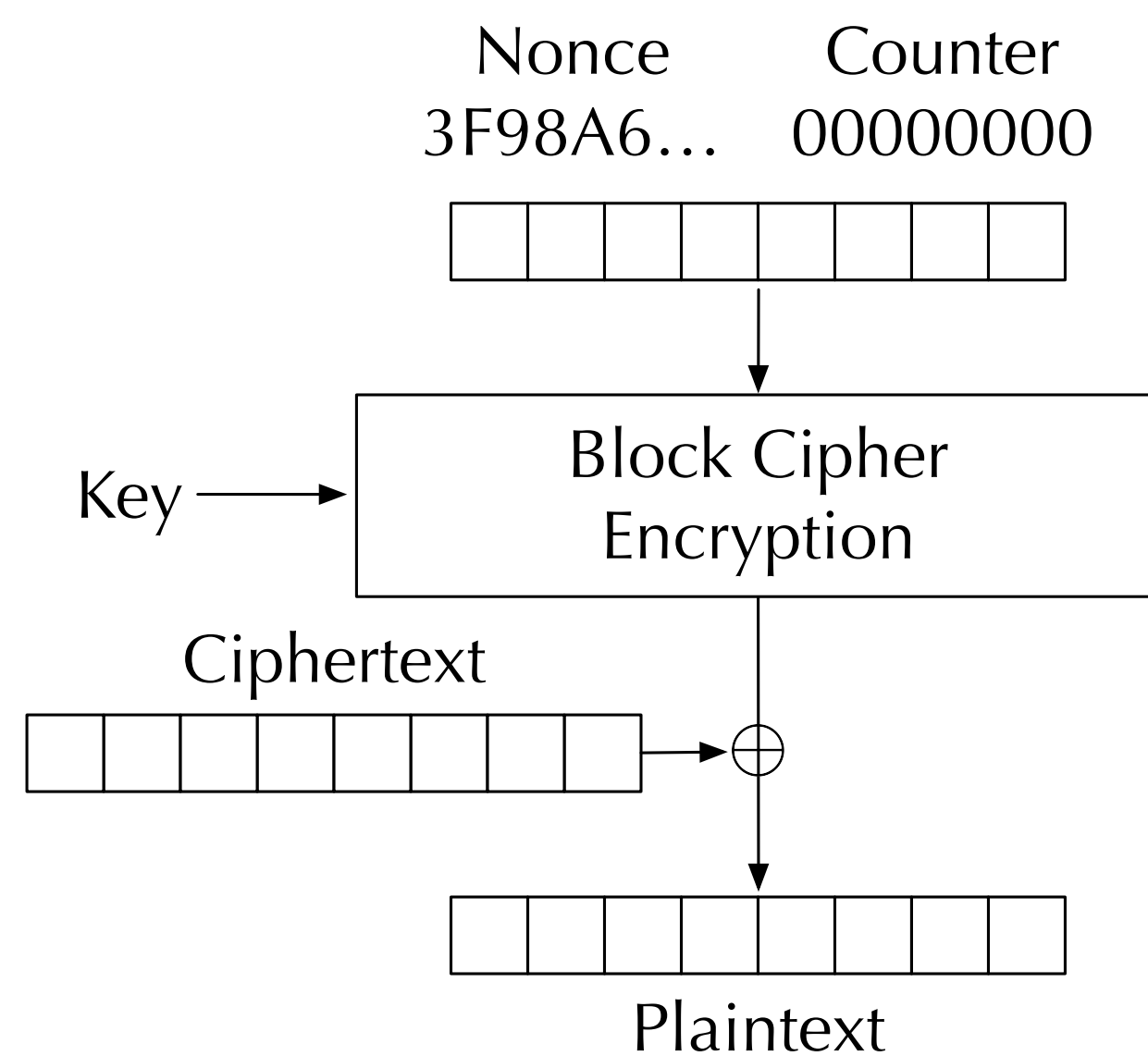


Encryption



# Counter (CTR) Mode *(cont.)*

- 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 **nonce** (*same idea as the IV*)
- Both encryption and decryption can be parallelized
- The key stream in the CTR mode can be calculated in parallel during the encryption



Decryption

# Modes for Authenticated Encryption (AE)

---

- We've now seen various encryption modes: EBC, CBC, CFB, CTR
  - *Others discussed in the textbook (e.g., OFB)*
- None of the encryption modes discussed so far can be used to achieve **message authentication** (*only message confidentiality...*)
- There are modes of operation that have been designed to combine **encryption** and **message authentication codes (MAC)**
  - GCM (Galois/Counter Mode)
  - CCM (Counter with CBC-MAC)
  - OCB mode (Offset Codebook Mode)

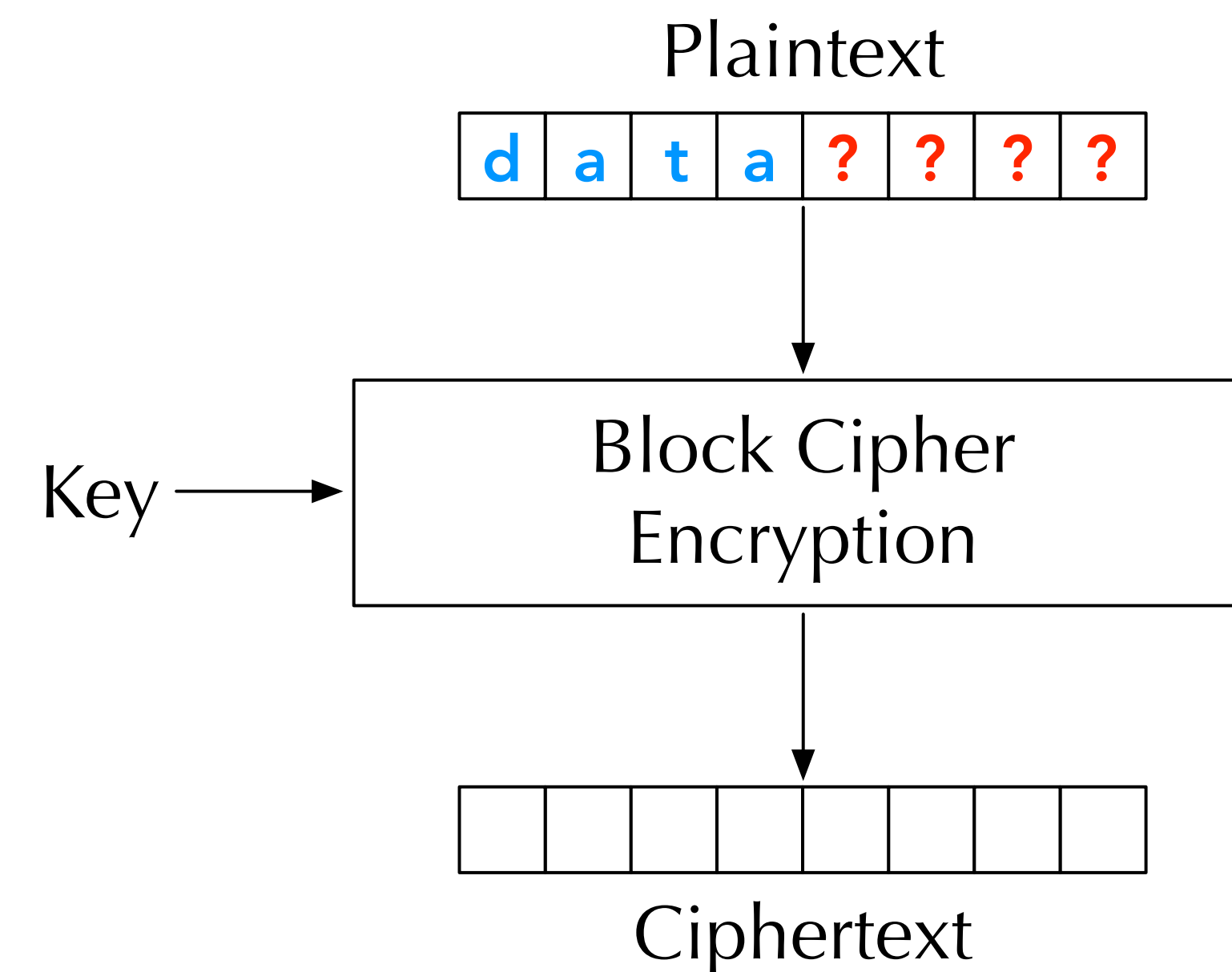
# Padding

*This Video Covers:*

- Padding: *what is it?* and *why do we need it?*
- Experiments with `openssl` and padding

# Padding

- Block cipher encryption modes divide plaintext into blocks; the size of each block should match the cipher's block size.
- No guarantee that the size of the last block matches the cipher's block size.
- The last block of the plaintext needs **padding**; i.e. before encryption, extra data needs to be added to the last block of the plaintext, so its size equals to the cipher's block size.
- Padding schemes (e.g., **PKCS#5**) need to clearly mark where the padding starts, **so decryption can remove the padded data**.





# An Experiment w/ Padding (#1)

## *What happens when data is smaller than the block size?*

```
$ echo -n "123456789" > plain.txt
$ ls -ld plain.txt # plaintext is 9 bytes
-rw-rw-r-- 1 seed seed 9 Mar 18 20:49 plain.txt
```

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

```
$ ls -ld cipher.bin # ciphertext becomes 16 bytes!
-rw-rw-r-- 1 seed seed 16 Mar 18 20:49 cipher.bin
```

```
$ openssl enc -aes-128-cbc -d -in cipher.bin -out plain2.txt \
  -K 00112233445566778899AABBCCDDEEFF \
  -iv 000102030405060708090A0B0C0D0E0F
```

```
$ ls -ld plain2.txt # decrypted ciphertext goes back to 9 bytes!
-rw-rw-r-- 1 seed seed 9 Mar 18 20:49 plain2.txt
```

## An Experiment w/ Padding (#2)

*How does decryption software know where padding starts?*

```
$ openssl enc -aes-128-cbc -d -in cipher.bin -out plain3.txt \  
  -K 00112233445566778899AABBCCDDEEFF \  
  -iv 000102030405060708090A0B0C0D0E0F -nopad  
$ ls -ld plain3.txt  
-rw-rw-r-- 1 seed seed 16 Mar 18 20:49 plain3.txt
```

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

7 bytes of **0x07** are added as the padding data

**In general**, for block size  $B$  and last block w/  $K$  bytes,  
 $B - K$  bytes of value  $B - K$  are added as the padding

# An Experiment w/ Padding (#3 – a special case)

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

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

# Initialization Vectors & Common Mistakes

*This Video Covers:*

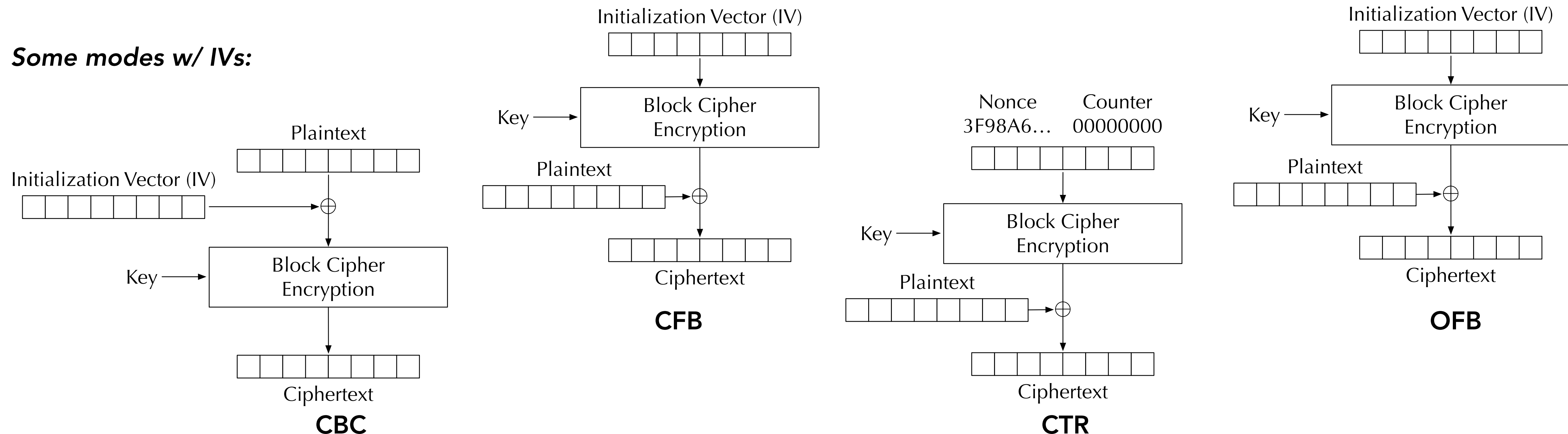
- Requirements for IVs
- Attacking Poorly Chosen IVs



# Initialization Vector and Common Mistakes

- Initialization Vectors have the following requirements:
  - IV is supposed to be stored or transmitted in plaintext
  - IV should not be reused → uniqueness
  - IV should not be predictable → pseudorandom

## Some modes w/ IVs:

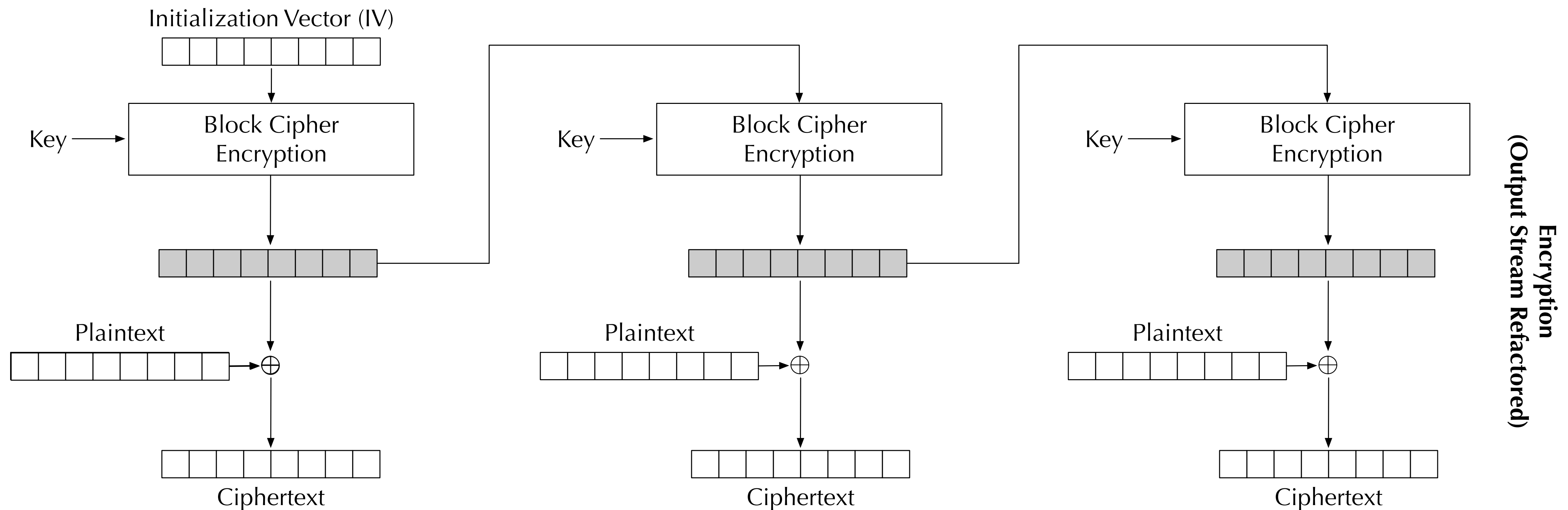


# IV should not be reused

## Scenario:

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

**Attacker Goal:**  
decrypt other plaintexts

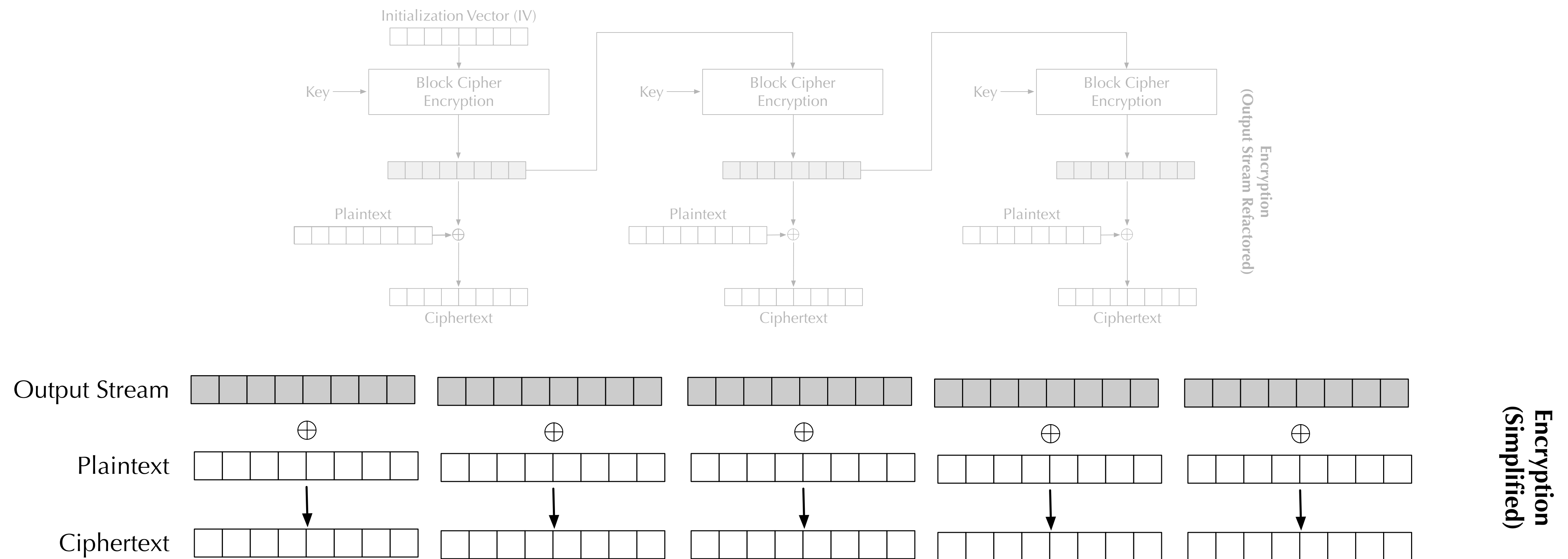


# IV should not be reused...

## Scenario:

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

**Attacker Goal:**  
decrypt other plaintexts

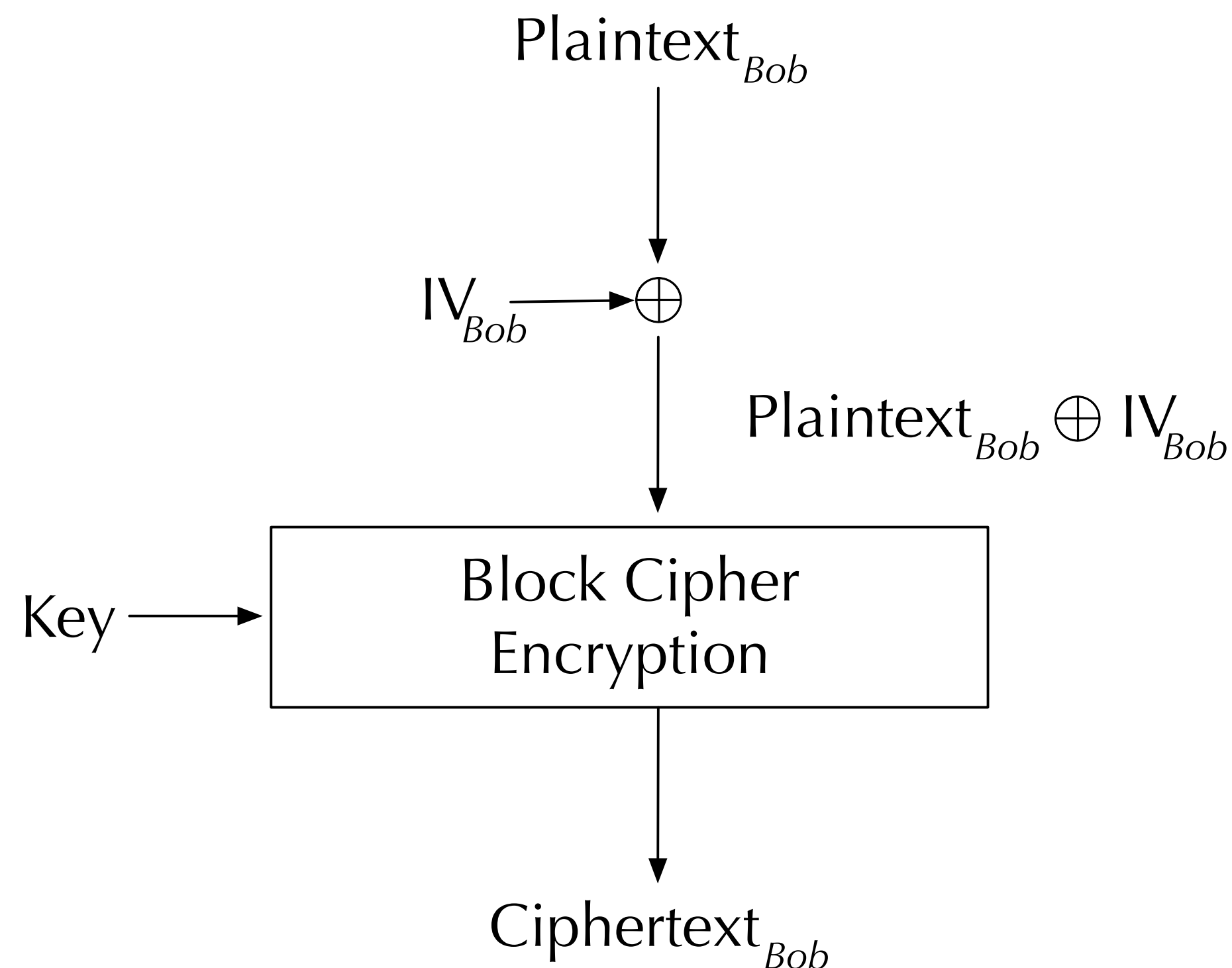


# IV should not be predictable

## Scenario:

- Suppose attacker can get victim to encrypt some chosen plaintexts ("**chosen-plaintext attack**")
- Plaintext messages are highly structured / there are few options (e.g., "Yes"/"No"; name of presidential candidate)
- Plaintexts encrypted using AES-128-CBC **and the IVs are predictable...**

**Attacker Goal:**  
Learn contents of other plaintexts

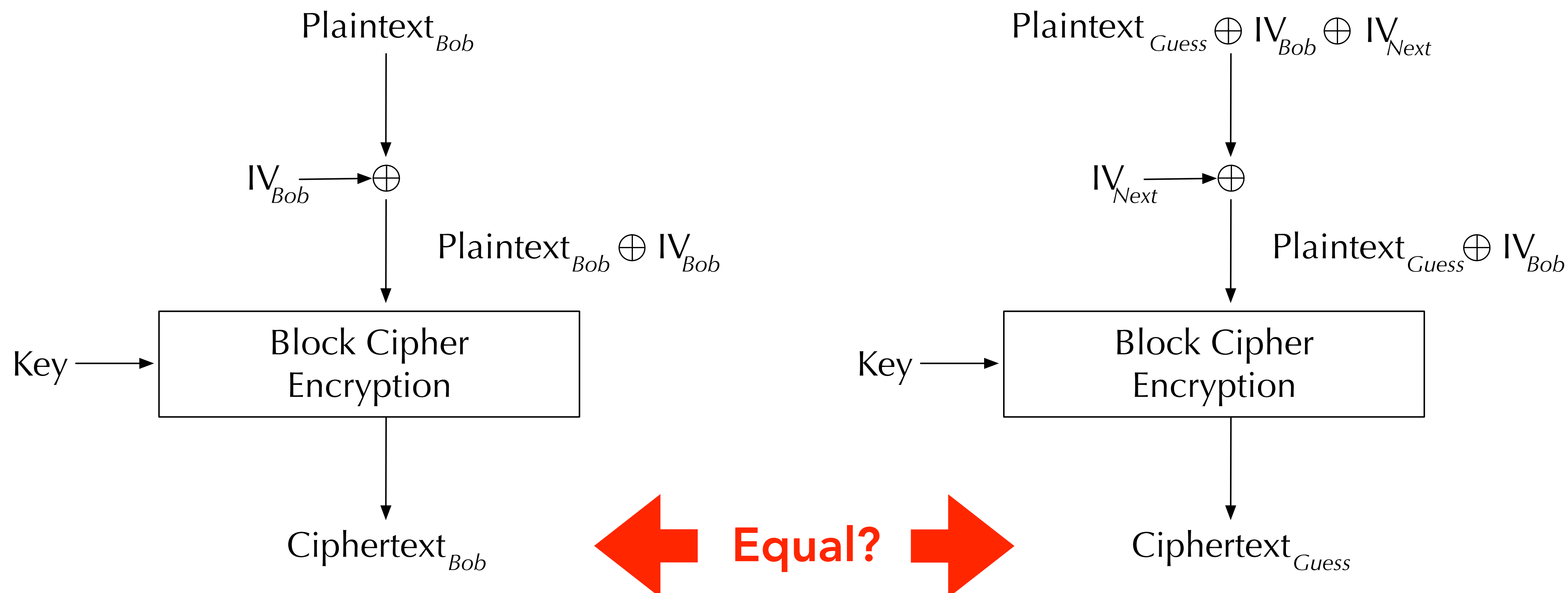


# IV should not be predictable...

## Scenario:

- Suppose attacker can get victim to encrypt some chosen plaintexts ("**chosen-plaintext attack**")
- Plaintext messages are highly structured / there are few options (e.g., "Yes"/"No"; name of presidential candidate)
- Plaintexts encrypted using AES-128-CBC **and the IVs are predictable...**

**Attacker Goal:**  
Learn contents of other plaintexts





# Programming Using Crypto APIs (Part I)

*This Video Covers:*

- Example of encryption & decryption using PyCryptodome
- Experiment: Attacking the integrity of ciphertext
- Authenticated Encryption (AE) with the GCM mode of operation

# Programming using Crypto APIs

```
#!/usr/bin/python3

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

key_hex_string = '00112233445566778899AABBCCDDEEFF'
iv_hex_string = '000102030405060708090A0B0C0D0E0F'
key = bytes.fromhex(key_hex_string)
iv = bytes.fromhex(iv_hex_string)
data = b'The quick brown fox jumps over the lazy dog'
print("Length of data: {0:d}".format(len(data))) # 43 bytes

# Encrypt the data piece by piece
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(data[0:32])
ciphertext += cipher.encrypt(Padding.pad(data[32:], 16))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Encrypt the entire data
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(Padding.pad(data, 16))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = cipher.decrypt(ciphertext)
print("Plaintext: {0}".format(Padding.unpad(plaintext, 16)))
```

- We use PyCryptodome package's APIs.
- Setup:
  1. Set the Key & IV (*hex string*)
  2. Define the data (*byte literal*)

# Programming using Crypto APIs

```
#!/usr/bin/python3

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

key_hex_string = '00112233445566778899AABBCCDDEEFF'
iv_hex_string = '000102030405060708090A0B0C0D0E0F'
key = bytes.fromhex(key_hex_string)
iv = bytes.fromhex(iv_hex_string)
data = b'The quick brown fox jumps over the lazy dog'
print("Length of data: {0:d}".format(len(data))) # 43 bytes

# Encrypt the data piece by piece
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(data[0:32])
ciphertext += cipher.encrypt(Padding.pad(data[32:], 16))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Encrypt the entire data
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(Padding.pad(data, 16))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = cipher.decrypt(ciphertext)
print("Plaintext: {0}".format(Padding.unpad(plaintext, 16)))
```

- We use PyCryptodome package's APIs.
- Approach 1:
  1. Setup
  2. Initialize cipher
  3. Encrypts first 32 bytes of data
  4. Encrypts the rest of the data
  5. Initialize cipher (start new chain)
  6. Encrypt the entire data
  7. Initialize cipher for decryption
  8. Decrypt

# Programming using Crypto APIs

```
#!/usr/bin/python3

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

key_hex_string = '00112233445566778899AABBCCDDEEFF'
iv_hex_string = '000102030405060708090A0B0C0D0E0F'
key = bytes.fromhex(key_hex_string)
iv = bytes.fromhex(iv_hex_string)
data = b'The quick brown fox jumps over the lazy dog'
print("Length of data: {0:d}".format(len(data))) # 43 bytes

# Encrypt the data piece by piece
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(data[0:32])
ciphertext += cipher.encrypt(Padding.pad(data[32:], 16))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Encrypt the entire data
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(Padding.pad(data, 16))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = cipher.decrypt(ciphertext)
print("Plaintext: {0}".format(Padding.unpad(plaintext, 16)))
```

- We use PyCryptodome package's APIs.
- Approach 2:
  1. Setup
  2. Initialize cipher
  3. Encrypts first 32 bytes of data
  4. Encrypts the rest of the data
  5. Initialize cipher (start new chain)
  6. Encrypt the entire data
  7. Initialize cipher for decryption
  8. Decrypt

# Programming using Crypto APIs

- Modes that ***do not need padding*** include CFB, OFB, and CTR.
- For these modes, the data fed into the **`encrypt()`** method can have an arbitrary length, and no padding is needed (*everything else is the same*)

```
# Encrypt the data piece by piece
cipher = AES.new(key, AES.MODE_OFB, iv)
ciphertext = cipher.encrypt(data[0:20])
ciphertext += cipher.encrypt(data[20:])
```



# Programming Using Crypto APIs (Part II)

*This Video Covers:*

- Example of encryption & decryption using PyCryptodome
- Experiment: Attacking the integrity of ciphertext
- Authenticated Encryption (AE) with the GCM mode of operation

Revisiting ***data integrity*** now that we have new tools...

# Attacks on Ciphertext Integrity

- Suppose an attacker makes changes to the ciphertext:

```
data = b'The quick brown fox jumps over the lazy dog'

# Encrypt the entire data
cipher = AES.new(key, AES.MODE_OFB, iv)
ciphertext = bytearray(cipher.encrypt(data))

# Change the 10th byte of the ciphertext
ciphertext[10] = 0xE9

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_OFB, iv)
plaintext = cipher.decrypt(ciphertext)
print(" Original Plaintext: {0}".format(data))
print("Decrypted Plaintext: {0}".format(plaintext))
```

- Result:

```
Original Plaintext: b'The quick brown fox jumps over the lazy dog'
Decrypted Plaintext: b'The quick grown fox jumps over the lazy dog'
```

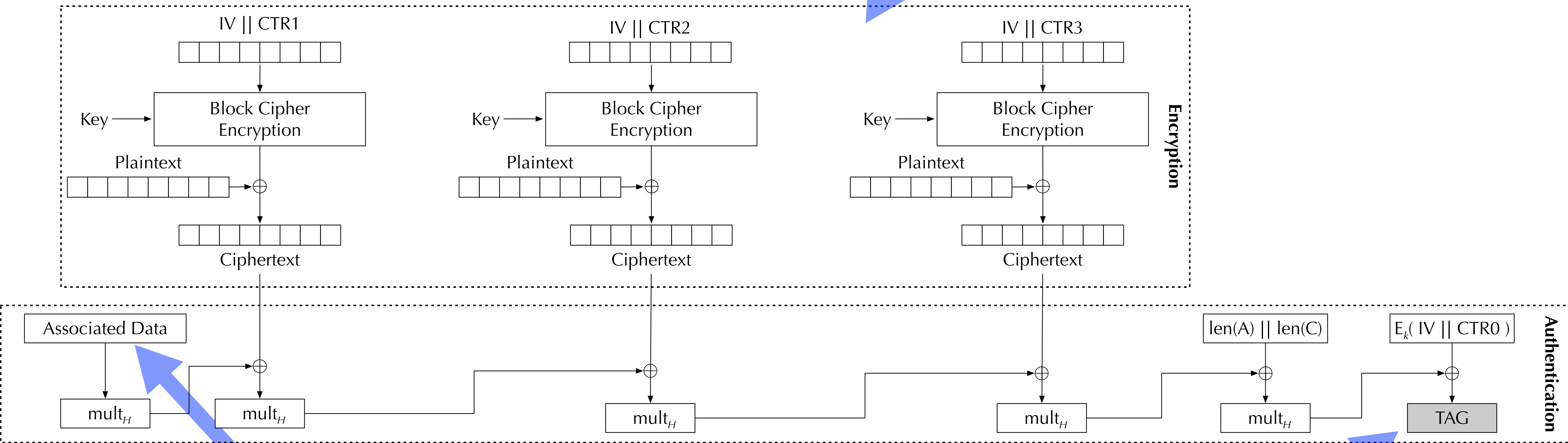
# Authenticated Encryption (AE)

---

- To protect the data integrity, the sender needs to generate a **Message Authentication Code (MAC)** from the ciphertext using a secret shared by the sender and the receiver.
  - The **ciphertext+MAC** will be sent to the receiver
  - Receiver will re-compute the MAC from the received ciphertext.
  - If the MAC is the same as the one received, the ciphertext has not been modified.
- Two operations are needed to achieve integrity of ciphertext:
  - one for **encrypting data**, and
  - one for **generating MAC**.
- **Authenticated Encryption** combines these two separate operations into one encryption mode. E.g., GCM, CCM, OCB.

# The Galois/Counter Mode (GCM)

Encrypt data using a familiar **block cipher & mode of operation**  
(AES w/ CTR mode)




**Associated Data:** unencrypted, but integrity protected  
(e.g., packet headers)

**MAC ("Tag")** that covers integrity of **ciphertext**  
and **associated data**



# Programming using the GCM Mode

In practice, use something like  
`get_random_bytes(N)` to get *N* random bytes



- The unique part of the GCM code is the **tag generation** and **verification**.
- Note the use of **digest()** to get the authentication tag, which is generated from the ciphertext.

```
#!/usr/bin/python3

# ...snip...
data = b'The quick brown fox jumps over the lazy dog'

# Encrypt the data
cipher = AES.new(key, AES.MODE_GCM, iv)
cipher.update(b'header')
ciphertext = bytearray(cipher.encrypt(data))
print("Ciphertext: {0}".format(ciphertext.hex()))
Ciphertext: ed1759cf244fa97f87de552c1254b1894d1d...

# Get the MAC tag
tag = cipher.digest()
print("Tag: {0}".format(tag.hex()))
Tag: 701f3c84e2da10aae4b76c89e9ea8427

# ...continued on next slide...
```

# Programming using the GCM Mode *(cont.)*

- After feeding the ciphertext to the cipher, we invoke **verify()** to verify whether the tag is still valid.

```
# Corrupt the ciphertext
ciphertext[10] = 0x00

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_GCM, iv)
cipher.update(b'header')
plaintext = cipher.decrypt(ciphertext)
print("Plaintext: {0}".format(plaintext))
Plaintext: b'The quick brown fox jumps over the lazy dog'

# Verify the MAC tag
try:
    cipher.verify(tag)
except:
    print("*** Authentication failed ***")
else:
    print("*** Authentication is successful ***")
```

# Experiment - GCM Mode

*What happens if we modify the ciphertext by changing the 10th byte to (0x00), then decrypt the modified ciphertext and try to verify the tag?*

```
$ ./enc_gcm.py
```

```
Ciphertext: ed1759cf244fa97f87de552c1254b1894d1dad83d8f...a11d
Tag: 701f3c84e2da10aae4b76c89e9ea8427
Plaintext: b'The quick brown fox jumps over the lazy dog'
*** Authentication failed ***
```

Check out the docs for tips on more realistic deployment of encryption/decryption and tag verification code:  
<https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html#gcm-mode>

# Summary: *Secret Key Encryption*

- Block Ciphers — *DES, 3DES, AES*
- Modes of Operation — *ECB, CBC, CFB, CTR*
- Authenticated Encryption — *GCM*
- Practical Issues (Mistakes & Best Practices)  
— *Padding, Initialization Vectors, Nonces, Using OpenSSL & PyCryptodome*

# You Try!

*Exam-like problems that you can use for practice!*

---

- A message is encrypted twice using the same key and same algorithm, but the ciphertexts are different. What could be the reason? Explain.
- AES is a block cipher; can we use it as a stream cipher, i.e., encrypting messages bits by bits (without using paddings)?
- We need to protect a packet, such that the payload of the packet is encrypted, but the integrity of the entire packet, including its header, is protected. What encryption mode can we use to achieve this goal?
- Alice encrypts a message using AES, and sends the ciphertext to Bob. Unfortunately, during the transmission, the 2nd bit of the third block in the ciphertext is corrupted. How much of the plaintext can Bob still recover if the mode of encryption is one of the followings: CBC, CFB, OFB, or CTR?