Secret-Key Encryption

Introduction

- 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
  - pubic-key encryption : different keys for encryption and decryption

- We focus on secret-key encryption in this chapter

Substitution Cipher

- Encryption is done by replacing *units* of plaintext with ciphertext, according to a fixed system.
- *Units* may be single letters, pairs of letters, triplets of letters, mixtures of the above, and so forth
- 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

- Encryption and decryption

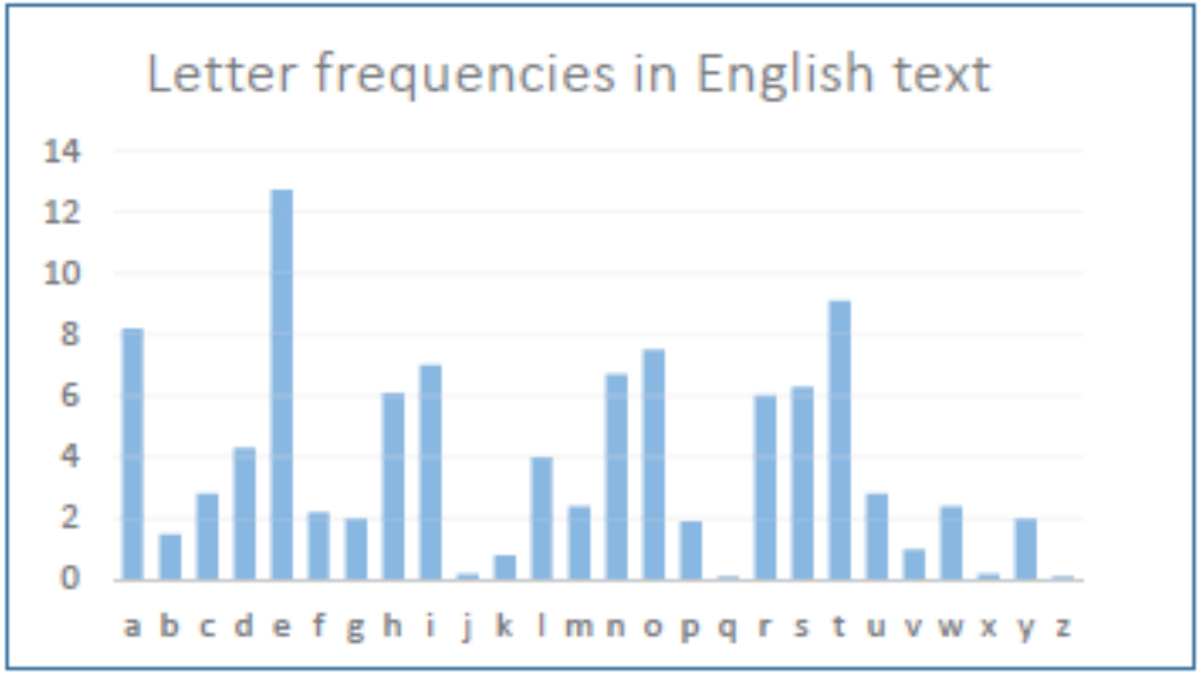
```
# Encryption
$ tr 'a-z' 'vgapnbrtmosicuxejhqyzflkdw' < plaintext  > ciphertext

# Decryption
$ tr 'vgapnbrtmosicuxejhqyzflkdw' 'a-z' < ciphertext > plaintext_new
```

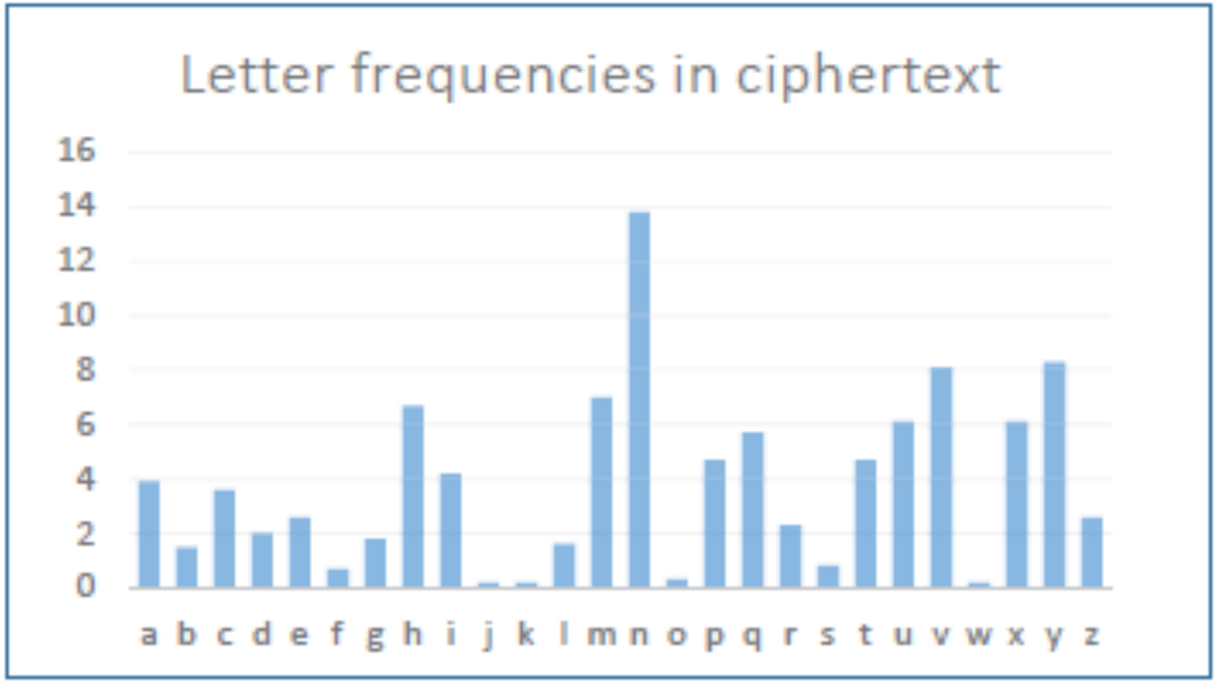
Breaking Monoalphabetic Substitution Cipher

- 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

Breaking Monoalphabetic Substitution Cipher

- **Letter** Frequency Analysis results:





Breaking Monoalphabetic Substitution Cipher

- **Bigram** Frequency Analysis results:

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

Breaking Monoalphabetic Substitution Cipher

- **Trigram** Frequency analysis results:

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 Monoalphabetic Substitution Cipher

- Applying the 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
axzNTRmES cxST NxTvgic
SEfERvi pmbbERENT ENmr
cmimTvRd cxpEiS HvfmNr
vNp mTvimvN cxpEiS lEF
```

```
$ tr ntyhquvmxbpz EHTRSNAIOFDU < ciphertext
THE ENIrcA cAaHINES lERE A SERIES OF EiEaTROcEaHANIaAi 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
AND ADOeTED gd cIiITARd AND rO
aOUNTRIES cOST NOTAgid NAwI rE
SEfERAi DIFFERENT ENIrcA cODEi
```

```
$ tr ntyhquvmxbpzfrcei EHTRSNAIOFDUVGMPL < 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 ...
```

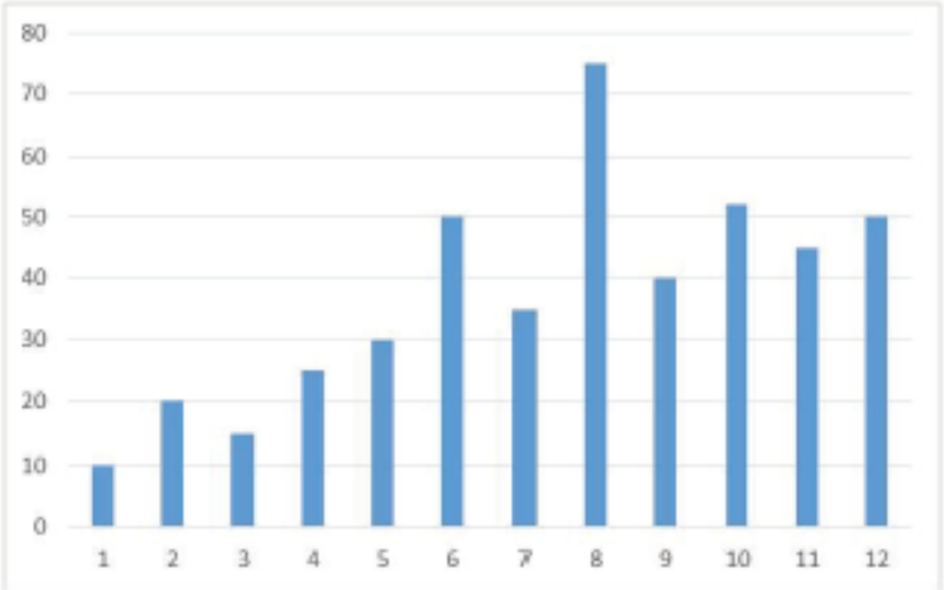
Data Encryption Standard (DES)

- DES is a block cipher - can only encrypt a block of data
- Block size for DES is 64 bits
- DES uses 56-bit keys although a 64-bit key is fed into the algorithm
- Theoretical attacks were identified. None was practical enough to cause major concerns.
- Triple DES can solve DES's key size problem

Advanced Encryption Standard (AES)

- AES is a block cipher
- 128-bit block size.
- Three different key sizes: 128, 192, and 256 bits

Encryption Modes



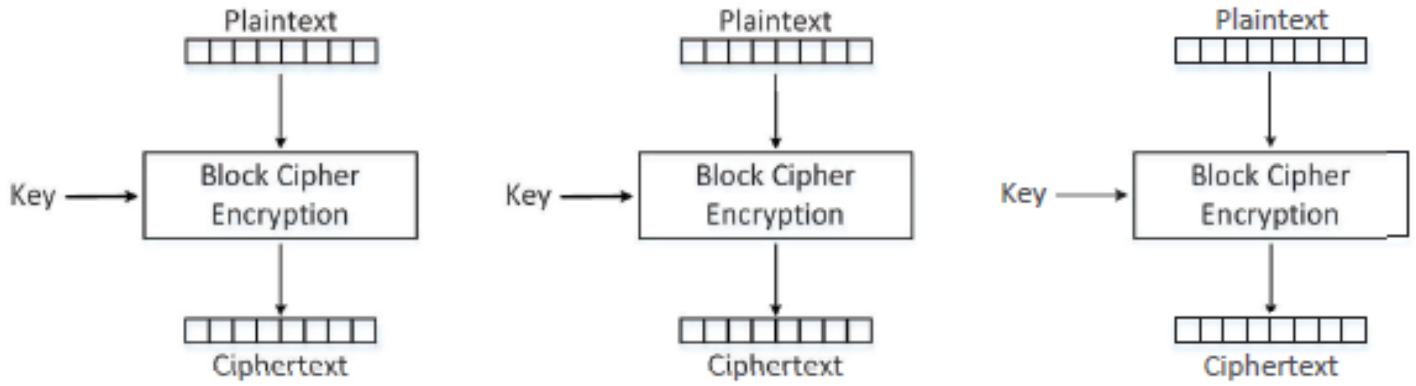
(a) The original image (`pic_original.bmp`)

(b) The encrypted image (`pic_encrypted.bmp`

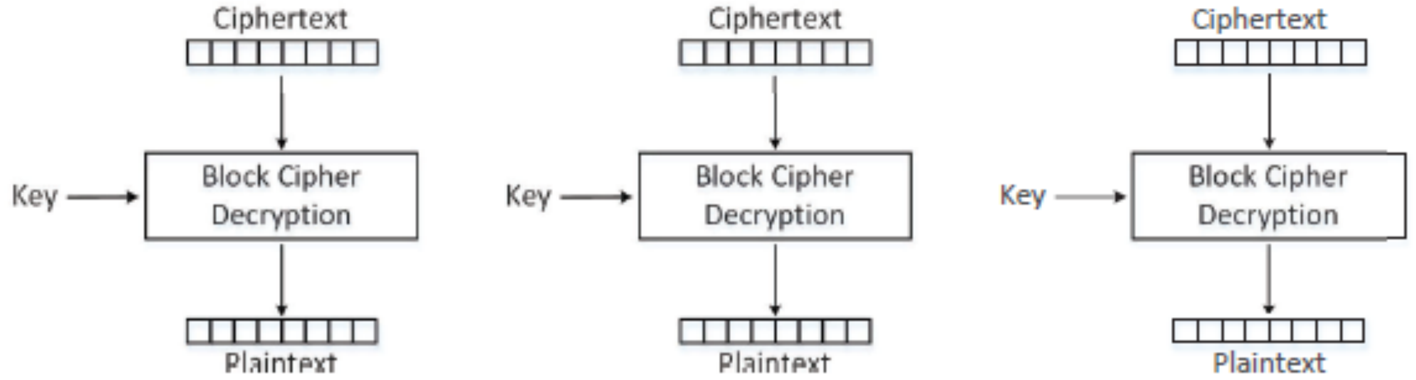
Encryption Modes

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

Electronic Codebook (ECB) Mode



(a) Electronic Codebook (ECB) mode encryption



(b) Electronic Codebook (ECB) mode decryption

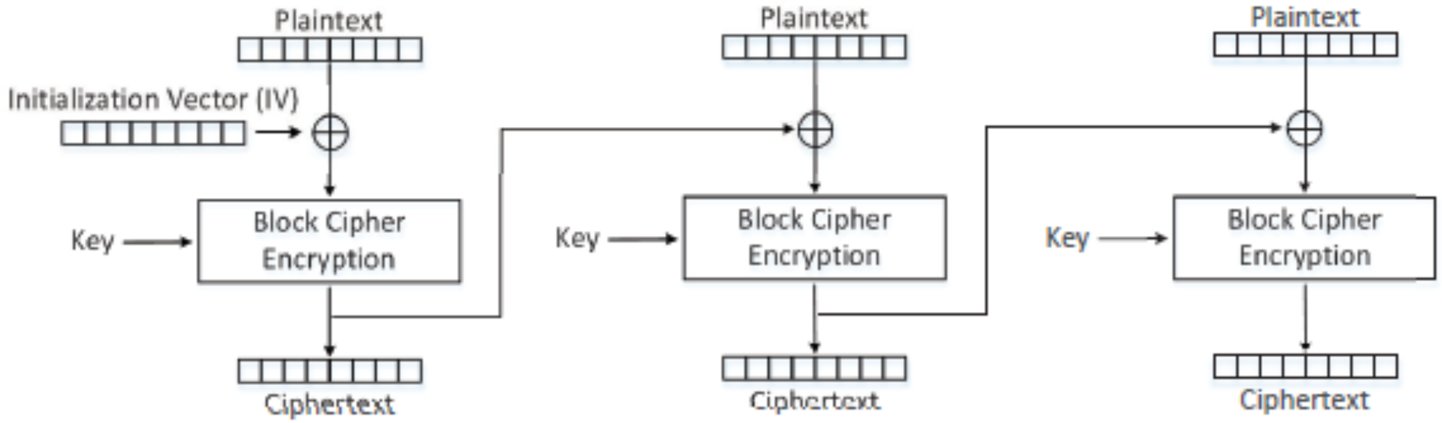
Electronic Codebook (ECB) Mode

- Using `openssl enc` command:

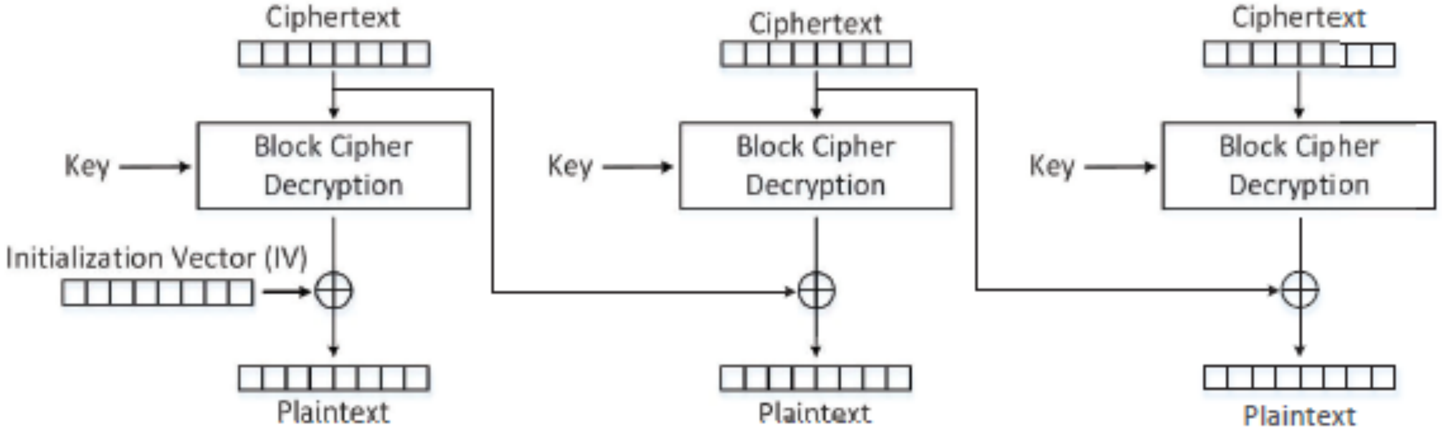
```
$ openssl enc -aes-128-ecb -e -in plain.txt -out cipher.txt \
          -K 00112233445566778899AABBCCDDEEFF
$ openssl enc -aes-128-ecb -d -in cipher.txt -out plain2.txt \
          -K 00112233445566778899AABBCCDDEEFF
```

- We use the 128-bit (key size) AES algorithm
- The **-aes-128-ecb** option specifies 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



(a) Cipher Block Chaining (CBC) mode encryption



(b) Cipher Block Chaining (CBC) mode decryption

- The main purpose of **IV** is to ensure that even if two plaintexts are identical, their ciphertexts are still different, because different IVs will be used.
- Decryption **can** be parallelized
- Encryption **cannot** be parallelized

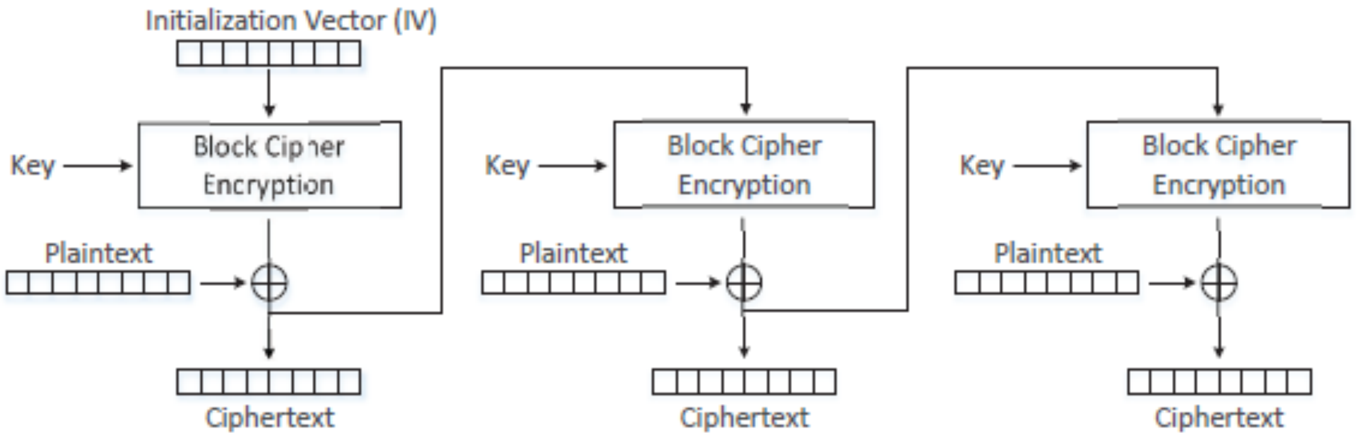
Cipher Block Chaining (CBC) Mode

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

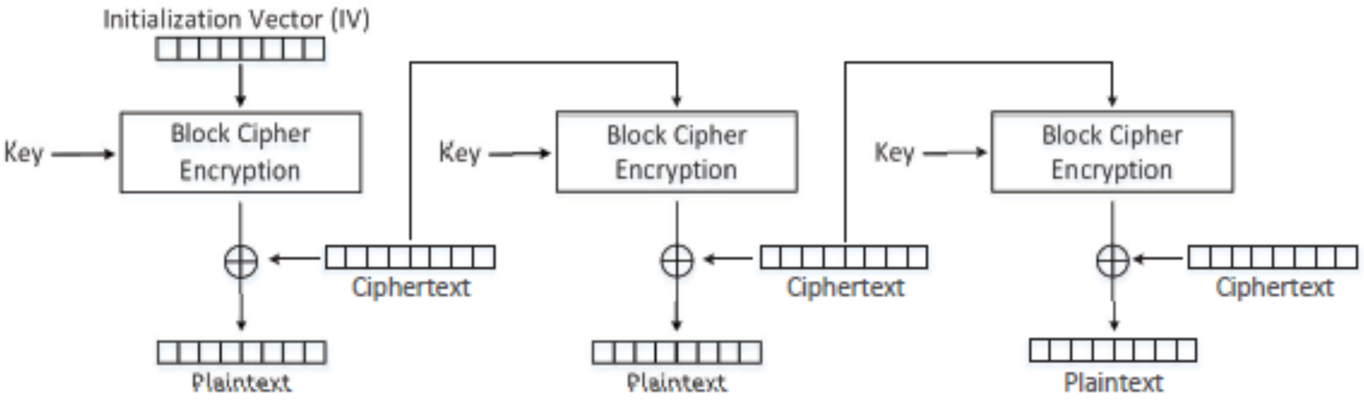
```
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.txt  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 000102030405060708090a0b0c0d0e0f
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher2.txt  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 000102030405060708090a0b0c0d0e0e
$ xxd -p cipher1.txt
52381c7726763ac132752bb29a32a68fc8dbcf20367fdfd03649b3a0d1744567
$ xxd -p cipher2.txt
50a9e3b81cc020d286d86fc7f1d8fb4268f9cd87c08126226c4626dbd4961d58
```

- We use the 128-bit (key size) AES algorithm
- The **-aes-128-cbc** option specifies CBC mode
- The **-e** option indicates encryption
- The **-iv** option is used to specify the Initialization Vector (IV)

Cipher Feedback (CFB) Mode



(a) Cipher Feedback (CFB) mode encryption



(b) Cipher Feedback (CFB) mode decryption

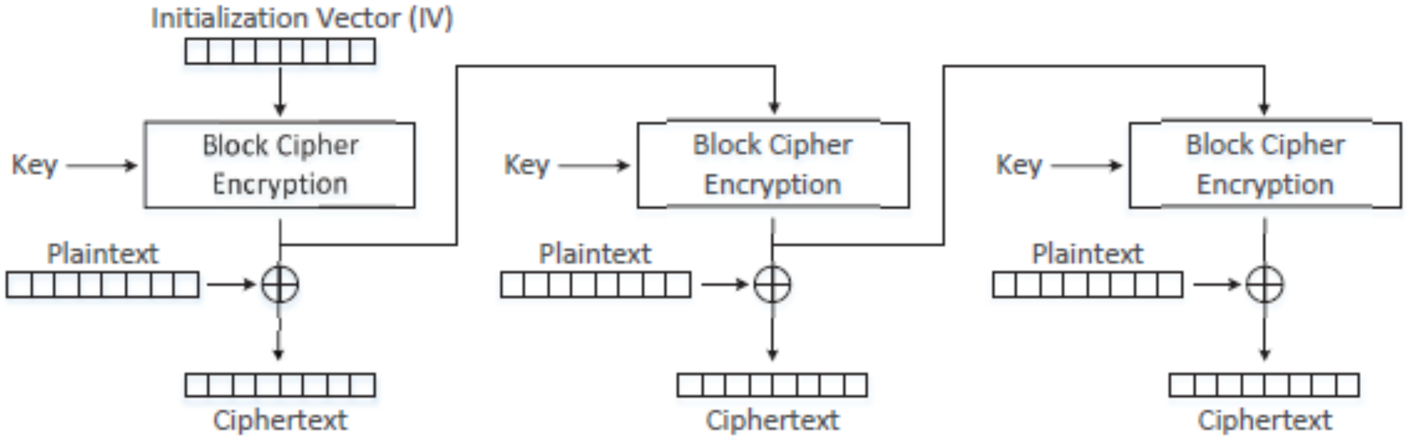
- A block cipher is turned into a stream cipher.
- Ideal for encrypting real-time data.
- Padding not required for the last block.
- decryption using the CFB mode can be parallelized, while encryption can only be conducted sequentially

Comparing encryption with CBC and CFB

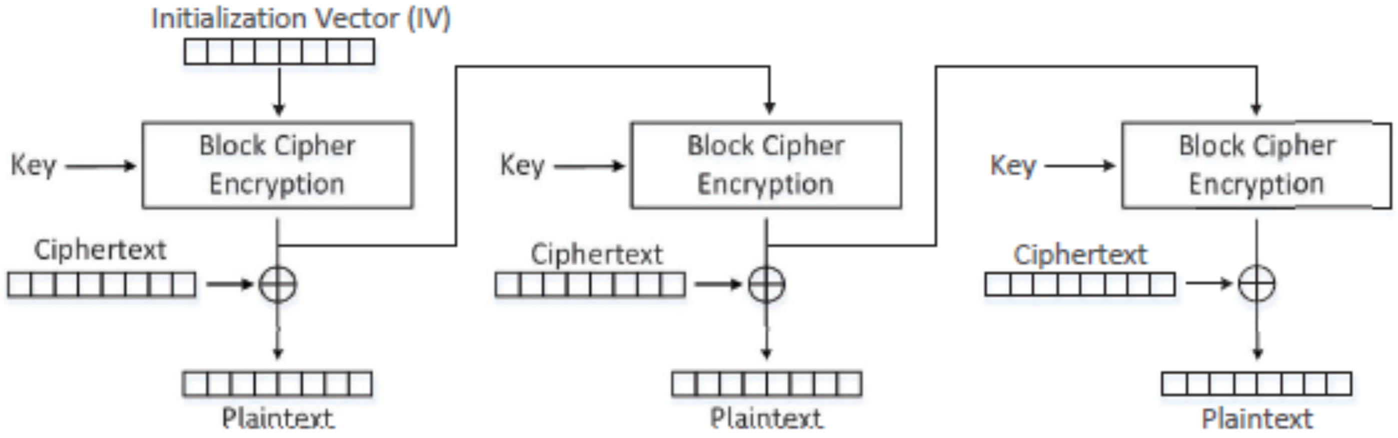
```
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.txt  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 000102030405060708090a0b0c0d0e0f
$ openssl enc -aes-128-cfb -e -in plain.txt -out cipher2.txt  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 000102030405060708090a0b0c0d0e0f
$ ls -l plain.txt cipher1.txt cipher2.txt
-rw-rw-r-- 1 seed seed 32 Jun 20 13:55 cipher1.txt
-rw-rw-r-- 1 seed seed 21 Jun 20 13:55 cipher2.txt
-rw-rw-r-- 1 seed seed 21 May 11 10:27 plain.txt
```

- Plaintext size is 21 bytes
- CBC mode: ciphertext is 32 bytes due padding
- CFB mode: ciphertext size is same as plaintext size (21 bytes)

Output Feedback (OFB) Mode



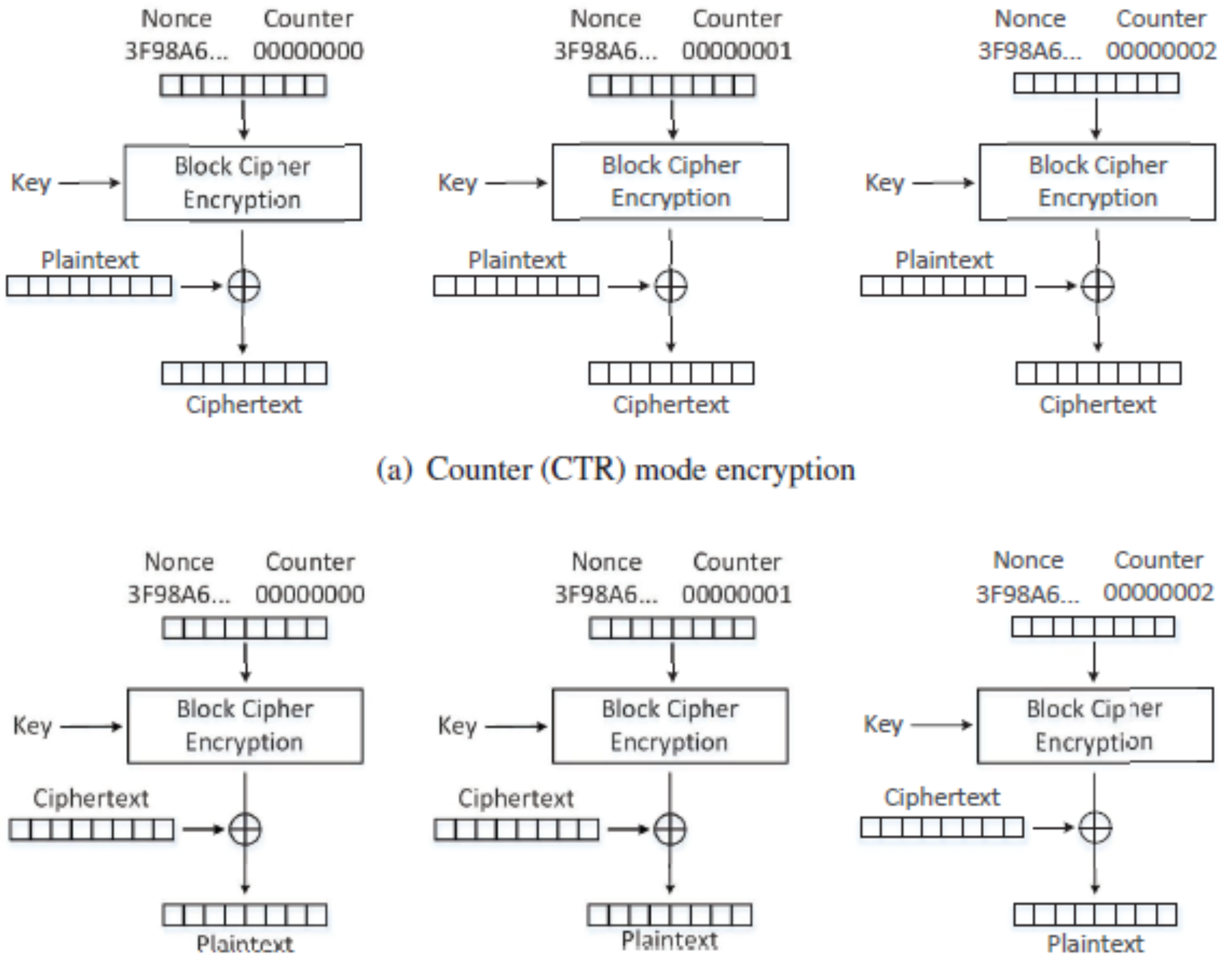
(a) Output Feedback (OFB) mode encryption



(b) Output Feedback (OFB) mode decryption

- Similar to CFB
  - Used as stream cipher
  - Does not need padding
  - Decryption can parallelized
- Encryption in the OFB mode can be parallelized

Counter (CTR) Mode



(a) Counter (CTR) mode encryption

(b) Counter (CTR) mode decryption

- It basically uses a counter to generate the key streams
- no key stream can be reused, hence the counter value for each block is prepended with a randomly generated value called *nonce*
- This nonce serves the same role as the IV does to the other encryption modes.
- both encryption and decryption can be parallelized
- the key stream in the CTR mode can be calculated in parallel during the encryption

Modes for Authenticated Encryption

- None of the Encryption modes discussed so far cannot be used to achieve message authentication
- A number of modes of operation have been designed to combine message authentication and encryption.
- Examples include
  - GCM (Galois/Counter Mode)
  - CCM (Counter with CBC-MAC)
  - OCB mode (Offset Codebook Mode)

Padding

- Block cipher encryption modes divide plaintext into blocks and 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.
- 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 need to clearly mark where the padding starts, so decryption can remove the padded data.
- Commonly used padding scheme is PKCS#5

Padding Experiment

```
$ echo -n "123456789" > plain.txt
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher.bin  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 0102030405060708090a0b0c0d0e0f
$ ls -ld cipher.bin
-rw-rw-r-- 1 seed seed 16 Jun 28 11:15 cipher.bin
$ openssl enc -aes-128-cbc -d -in cipher.bin -out plain2.txt \
              -K 00112233445566778889aabbccddeeff \
              -iv 0102030405060708
$ ls -ld plain2.txt
-rw-rw-r-- 1 seed seed  9 Jun 28 11:16 plain2.txt
```

- Plaintext size is 9 bytes.
- Size of ciphertext (cipher.bin) becomes 16 bytes

Padding Experiment

- How does decryption software know where padding starts?

```
$ openssl enc -aes-128-cbc -d -in cipher.bin -out plain3.txt \
              -K 00112233445566778889aabbccddeeff \
              -iv 0102030405060708 -(@*\textbf{nopad}@*)

$ ls -ld plain3.txt
-rw-rw-r-- 1 seed seed 16 Jun 28 11:18 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

Padding Experiment – Special case

- What if the size of the plaintext is already a multiple of the block size (so no padding is needed), and its last seven bytes are all 0x07

```
$ openssl enc -aes-128-cbc -e -in plain3.txt -out cipher3.bin \
              -K 00112233445566778889aabbccddeeff \
              -iv 0102030405060708
$ openssl enc -aes-128-cbc -d -in cipher3.bin -out plain3_new.txt \
              -K 00112233445566778889aabbccddeeff \
              -iv 0102030405060708 -(@*\textbf{nopad}@*)

$ ls -ld cipher3.bin
-rw-rw-r-- 1 seed seed 32 Jun 28 11:27 cipher3.bin
$ 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 full block is added as the padding).
- Therefore, in PKCS#5, if the input length is already an exact multiple of the block size B, then B bytes of value B will be added as the padding.

Initial Vector and Common Mistakes

- Initial vectors have the following requirements:
  - IV is supposed to be stored or transmitted in plaintext
  - IV should not repeat (uniqueness).
  - IV should not be predictable.

Experiment - IV should not be predictable

- Eve calculates the next IV

```
IV_bob:  4ae71336e44bf9bf79d2752e234818a5

# Encrypt Bob’s vote
$ echo -n "John Smith......" > P1
$ openssl enc -aes-128-cbc -e -in P1 -out C1  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 4ae71336e44bf9bf79d2752e234818a5

# Calculate IV_next from IV_bob
$ echo -n 4ae71336e44bf9bf79d2752e234818a5 | xxd -r -p > IV_bob
$ md5sum IV_bob
398d01fdf7934d1292c263d374778e1a

# Therefore, IV_next is 398d01fdf7934d1292c263d374778e1a
```

Experiment - IV should not be predictable

- Eve guesses that Bob voted for John Smith, so she creates *P1_guessed* and XOR it with IV_bob and IV_next, and finally constructs the name for a write-in candidate.

```
$ echo -n "John Smith......" > P1_guessed

# Convert the ascii string to hex string
$ xxd -p P1_guessed
4a6f686e20536d6974682e2e2e2e2e2e

# XOR P1_guessed with IV_bob
$ xor.py 4a6f686e20536d6974682e2e2e2e2e2e \
         4ae71336e44bf9bf79d2752e234818a5
00887b58c41894d60dba5b000d66368b

# XOR the above result with with IV_next
$ xor.py 00887b58c41894d60dba5b000d66368b \
         398d01fdf7934d1292c263d374778e1a
39057aa5338bd9c49f7838d37911b891

# Convert the above hex string to binary and save to P2
$ echo -n "39057aa5338bd9c49f7838d37911b891" | xxd -r -p > P2
```

Experiment - IV should not be predictable

- Eve gives her write-in candidate's name (stored in P2) to the voting machine, which encrypts the name using IV_next as the IV. The result is stored in C2.
- If C1 (Bob's encrypted vote) == C2, then Eve knows for sure that Bob has voted for "John Smith".

```
$ openssl enc -aes-128-cbc -e -in P2 -out C2  \
              -K  00112233445566778899AABBCCDDEEFF \
              -iv 398d01fdf7934d1292c263d374778e1a

# Compare C1 and C2
$ xxd -p C1
7380ee1c0f9eb7dae28c1ba6a1a7431011428 8f771139da8ec99dfb0036e38ce
$ xxd -p C2
7380ee1c0f9eb7dae28c1ba6a1a74310114288f771139da8ec99dfb0036e38ce
```

Programming using Cryptography 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)))

# Encrypt the data piece by piece
cipher = AES.new(key, AES.MODE_CBC, iv)                    ①
ciphertext  = cipher.encrypt(data[0:32])                   ②
ciphertext += cipher.encrypt(Padding.pad(data[32:], 16))   ③
print("Ciphertext: {0}".format(ciphertext.hex()))

# Encrypt the entire data
cipher = AES.new(key, AES.MODE_CBC, iv)                    ④
ciphertext = cipher.encrypt(Padding.pad(data, 16))         ⑤
print("Ciphertext: {0}".format(ciphertext.hex()))

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_CBC, iv)                    ⑥
plaintext = cipher.decrypt(ciphertext)                     ⑦
print("Plaintext: {0}".format(Padding.unpad(plaintext, 16)))
```

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

Programming using Cryptography 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.
- Example below shows OFB encryption

```
# 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:])
```

Attack on ciphertext’s integrity

- Attacker makes changes to ciphertext (Line 2)

```
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))          ①

# Change the 10th byte of the ciphertext
ciphertext[10] = 0xE9                                  ②

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_OFB, iv)
plaintext = cipher.decrypt(ciphertext)                 ③
print("Original  Plaintext: {0}".format(data))
print("Decrypted Plaintext: {0}".format(plaintext))
```

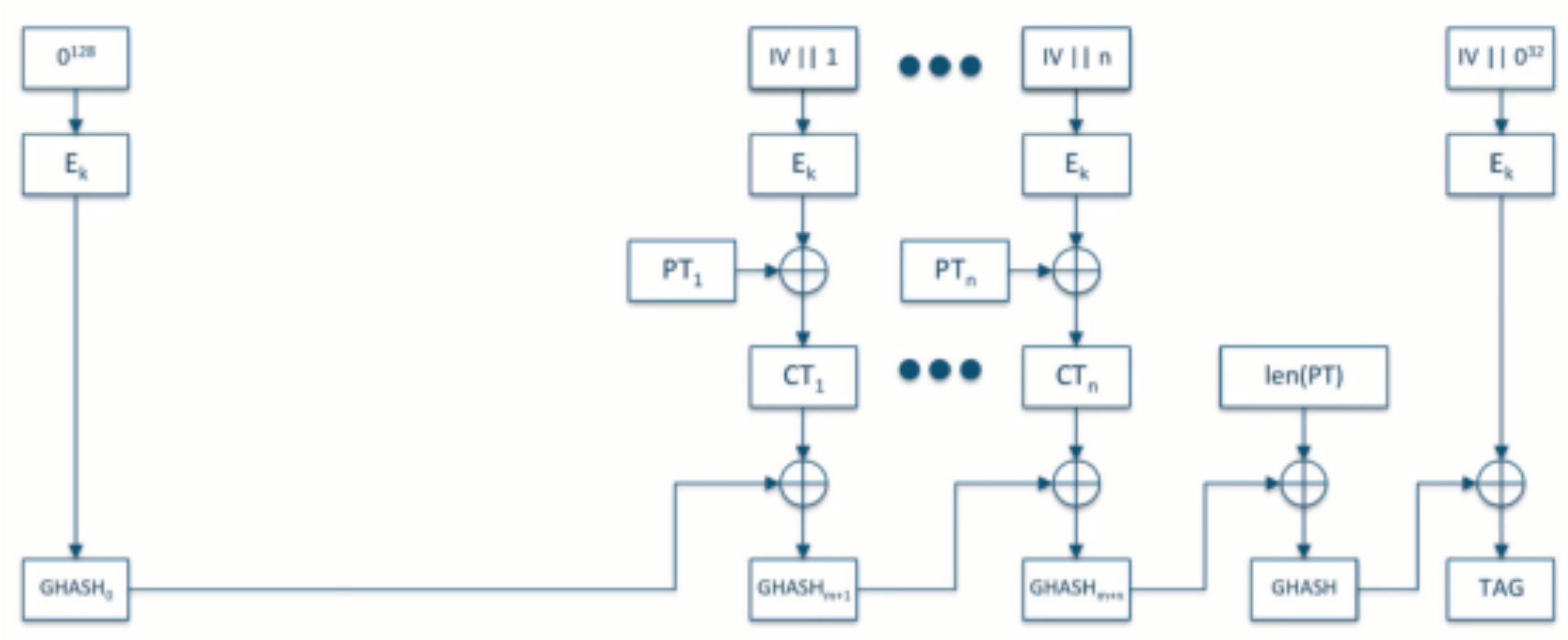
- Result

```
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

- To protect the 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 MAC and the ciphertext will be sent to the receiver, who will compute a MAC from the received ciphertext.
- If the MAC is the same as the one received, the ciphertext is not modified.
- Two operations are needed to achieve integrity of ciphertext: one for encrypting data and other for generating MAC.
- **Authenticated encryption** combines these two separate operations into one encryption mode. E.g GCM, CCM, OCB

The GCM Mode



0^{128}
E_k
$GHASH_0$
$IV || 1$
E_k
PT_1
CT_1
$GHASH_{m+1}$
$IV || n$
E_k
PT_n
CT_n
$GHASH_{m+n}$
len(PT)
GHASH
$IV || 0^{32}$
E_k
TAG

Programming using the GCM Mode

```
#!/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’

# Encrypt the data
cipher = AES.new(key, AES.MODE_GCM, iv)                    ①
cipher.update(b’header’)                                   ②
ciphertext  = bytearray(cipher.encrypt(data))
print("Ciphertext: {0}".format(ciphertext.hex()))

# Get the MAC tag
tag = cipher.digest()                                      ③
print("Tag: {0}".format(tag.hex()))
```

The unique part of the above code is the tag generation and verification.
In Line 3 , we use the **digest()** to get the authentication tag, which is generated from the ciphertext.

Programming using the GCM Mode

```
# Corrupt the ciphertext
ciphertext[10] = 0x00                                        ④

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_GCM, iv)
cipher.update(b'header')                                     ⑤
plaintext = cipher.decrypt(ciphertext)
print("Plaintext: {0}".format(plaintext))

# Verify the MAC tag
try:
   cipher.verify(tag)                                        ⑥
except:
   print("*** Authentication failed ***")
else:
   print("*** Authentication is successful ***")
```

In Line 6 , after feeding the ciphertext to the cipher, we invoke **verify()** to verify whether the tag is still valid.

Experiment - GCM Mode

- We modify the ciphertext by changing the 10th byte to (0x00)
- Decrypt the modified ciphertext and verify tag

```
$ enc_gcm.py
Ciphertext: ed1759cf244fa97f87de552c1...a11d
Tag: 701f3c84e2da10aae4b76c89e9ea8427
Plaintext: b'The quick 7rown fox jumps over the lazy dog'
*** Authentication failed ***
```