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
  • public-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' 'vgapnbtmosicuxejhqyzflkdw' < plaintext > ciphertext

# Decryption
$ tr 'vgapnbtmosicuxejhqyzflkdw' '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:

<table>
<thead>
<tr>
<th>English Bigram</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>TH</td>
<td>2.71</td>
</tr>
<tr>
<td>HE</td>
<td>2.33</td>
</tr>
<tr>
<td>IN</td>
<td>2.03</td>
</tr>
<tr>
<td>ER</td>
<td>1.78</td>
</tr>
<tr>
<td>AN</td>
<td>1.61</td>
</tr>
<tr>
<td>RE</td>
<td>1.41</td>
</tr>
<tr>
<td>ES</td>
<td>1.32</td>
</tr>
<tr>
<td>ON</td>
<td>1.32</td>
</tr>
<tr>
<td>ST</td>
<td>1.25</td>
</tr>
<tr>
<td>NT</td>
<td>1.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ciphertext Bigram</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>tn</td>
<td>77</td>
</tr>
<tr>
<td>np</td>
<td>50</td>
</tr>
<tr>
<td>yt</td>
<td>76</td>
</tr>
<tr>
<td>hn</td>
<td>45</td>
</tr>
<tr>
<td>nh</td>
<td>61</td>
</tr>
<tr>
<td>nu</td>
<td>44</td>
</tr>
<tr>
<td>ng</td>
<td>51</td>
</tr>
<tr>
<td>mu</td>
<td>42</td>
</tr>
<tr>
<td>vu</td>
<td>51</td>
</tr>
<tr>
<td>cv</td>
<td>42</td>
</tr>
</tbody>
</table>
Breaking Monoalphabetic Substitution Cipher

• **Trigram** Frequency analysis results:

<table>
<thead>
<tr>
<th>Trigram frequency in English</th>
<th>Trigram frequency in ciphertext (The top-10 patterns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE : 1.81</td>
<td>ytn : 60</td>
</tr>
<tr>
<td>AND : 0.73</td>
<td>tnh : 13</td>
</tr>
<tr>
<td>ING : 0.72</td>
<td>vup : 26</td>
</tr>
<tr>
<td>ENT : 0.42</td>
<td>pyt : 13</td>
</tr>
<tr>
<td>ION : 0.42</td>
<td>nhc : 16</td>
</tr>
<tr>
<td>HER : 0.36</td>
<td>hcv : 13</td>
</tr>
<tr>
<td>FOR : 0.34</td>
<td>nhn : 15</td>
</tr>
<tr>
<td>THA : 0.33</td>
<td>tne : 13</td>
</tr>
<tr>
<td>NTH : 0.33</td>
<td>nuy : 14</td>
</tr>
<tr>
<td>INT : 0.32</td>
<td>mrc : 13</td>
</tr>
</tbody>
</table>
Breaking Monoalphabetic Substitution Cipher

• Applying the partial mappings...
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 00112233445566778899AABBCDDEEFF
$ openssl enc -aes-128-ecb -d -in cipher.txt -out plain2.txt \
-K 00112233445566778899AABBCDDEEFF
```

- 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 indicates decryption
- The `-K` option is used to specify the encryption/decryption key
Cipher Block Chaining (CBC) Mode

- 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 01122334566778899AABBCCDDEEFF \\
   -iv 0010203040506070890a0b0c0d0e0f
$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher2.txt \\
   -K 01122334566778899AABBCCDDEEFF \\
   -iv 0010203040506070890a0b0c0d0e0f
$ 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 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

- 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

- Similar to CFB
  - Used as stream cipher
  - Does not need padding
  - Decryption can parallelized

- Encryption in the OFB mode can be parallelized

![Diagram of OFB Mode Encryption and Decryption](image-url)
Counter (CTR) Mode

- 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

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

• How does decryption software know where padding starts?

```bash
$ openssl enc -aes-128-cbc -d -in cipher.bin -out plain3.txt \
  -K 0011223344556677889aabbccddeeff \
  -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

• 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

```plaintext
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 $P1_{\text{guessed}}$ and XOR it with $IV_{\text{bob}}$ and $IV_{\text{next}}$, 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
4a6f686e20536d6974682e2e2e2e2e

# XOR P1_guessed with IV_bob
$ xor.py 4a6f686e20536d6974682e2e2e2e2e \n          4ae71336e44bf9bf79d2752e234818a5
       00887b58c41894d60dba5b000d66368b

# XOR the above result with with IV_next
$ xor.py 00887b58c41894d60dba5b000d66368b \n          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”.

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

# Compare C1 and C2
$ xxd -p C1
7380ee1c0f9eb7dae28c1ba6a1a74310114288f771139da8ec99dfb0036e38ce

$ xxd -p C2
7380ee1c0f9eb7dae28c1ba6a1a74310114288f771139da8ec99dfb0036e38ce
```
Programming using Cryptography APIs

```python
#!/usr/bin/python3
from Crypto.Cipher import AES
from Crypto.Util import Padding

key_hex_string = '001123456789ABCDDEEFF'
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)  # Initialize cipher
ctext = cipher.encrypt(data[0:32])  # Encrypts first 32 bytes of data
ctext += cipher.encrypt(Padding.pad(data[32:], 16))  # Encrypts the rest of the data
print("Ciphertext: {0}".format(ctext.hex()))

# Encrypt the entire data
ctext = cipher.encrypt(Padding.pad(data, 16))  # Initialize cipher (start new chain)
print("Ciphertext: {0}".format(ctext.hex()))

# Decrypt the ciphertext
plaintext = cipher.decrypt(ctext)  # Decrypt
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
• 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

```python
# 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)

```python
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] = 0xEE  # 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
Programming using the GCM Mode

```python
#!/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: {}".format(ciphertext.hex()))

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

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 brown fox jumps over the lazy dog'
*** Authentication failed ***
```