Cracking AES

Cracking AES encryption is generally infeasible due to its robust security. However, if parts of the key are known or predictable, it becomes possible to perform a targeted brute-force attack on the unknown portions. This tutorial guides you through using CrypTool 2 to decrypt AES-encrypted ciphertexts with partially known keys.

Original Project source: https://samsclass.info/141/proj/C201.htm

Prerequisites:
• A Windows machine (real or virtual).
• CrypTool 2 installed. Download it from [CrypTool 2 Downloads](#).

Steps:
1. **Launch CrypTool 2:**
• Open CrypTool 2 on your Windows machine.
2. **Access the Wizard:**
• In the "Main Functions" section on the left, click the wand icon labeled "Use the wizard…".
3. **Select Cryptanalysis:**
• In the "TASK SELECTION" screen, select "Cryptanalysis" and click "Next".
4. **Choose Modern Encryption:**
• In the "AGE SELECTION" screen, select "Modern Encryption" and click "Next".
5. **Select Symmetric Encryption:**
• In the "TYPE SELECTION" screen, choose "Symmetric Encryption" and click "Next".
6. **Choose AES Algorithm:**
• In the "ALGORITHM SELECTION" screen, select "AES" and click "Next".
7. **Opt for Ciphertext-Only Analysis:**
• In the next screen, ensure "Ciphertext-Only" is selected and click "Next".
8. **Configure the Attack:**
• In the "AES - CIPHERTEXT-ONLY ANALYSIS" screen:
• **Ciphertext:** Input the provided ciphertext.
• **Keypattern:** Enter the known parts of the key, using asterisks (*) for unknown hexadecimal characters. For example, if the key is known to be all zeroes except for the last 24 bits, you would enter 00000000000000000000000000******.
• **Chaining Mode:** Select the appropriate mode (e.g., ECB).
9. **Execute the Attack:**
• Click "Next" to start the analysis. CrypTool 2 will attempt all possible combinations for the unknown key portions, evaluating each decryption's entropy to identify the most probable plaintext.
10. **Review the Results:**
• Once the process completes, the decrypted plaintext will be displayed.

Challenges:

C 201.1: Shakespeare

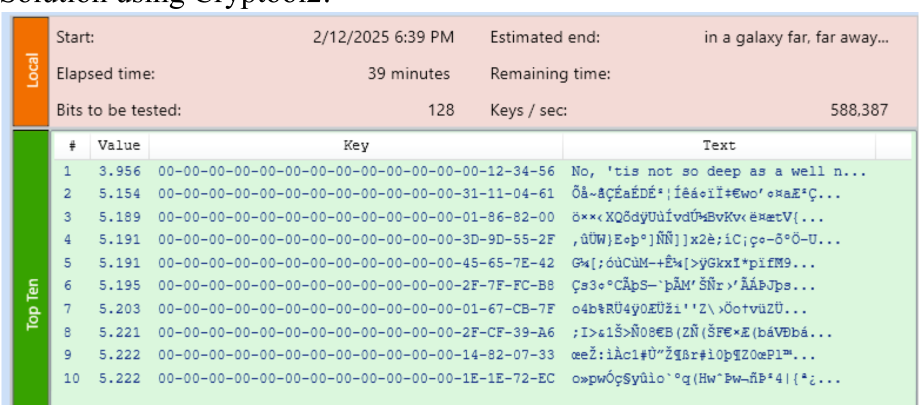
Ciphertext:
EF490B9CD4F92E3CA945DF24DAAFF8A0FF7FA784A50C57E296CE69C62C4F7FE6B4D1
D2D144D98C8D871D4615BF2533C76DD518FAC35729D45E772B6365E5A457AE92E33922
E1A5FE9E1DC5F1AEDEDD7EB32B1630AF4C5F10A453EEFF1E9C9CE0A3F9FCD0DDB2
A36C016B70B76E2CFE5B4A0377AA11521F7032E308FC4954D9BB3495CE8E07F4800B7A
A53B8BCD6291C7167B52F0F5F921C78CDCAB9606666543D4DEF1917CC389C20FB3E10
99F0FBBCA

Key Pattern: 00000000000000000000000000****** (all zeroes except for the last 24 bits).

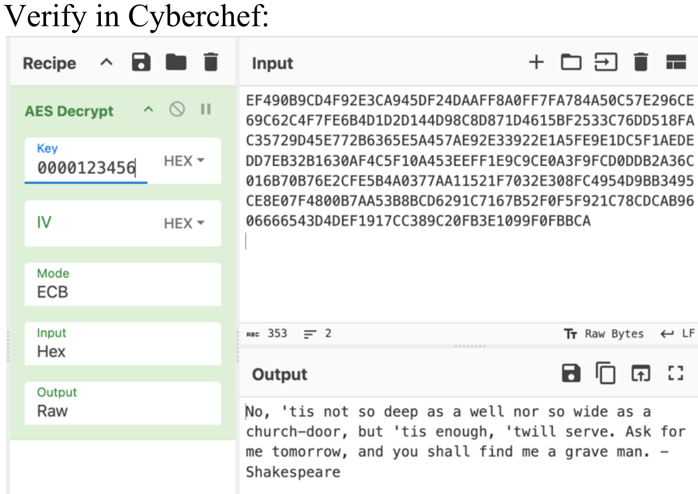
Chaining Mode: ECB

Objective: Decrypt the ciphertext to find the plaintext. The flag is the author of the decrypted quote.

Solution using Cryptool2:



Verify in Cyberchef:



C 201.2: DO NOT GO GENTLE

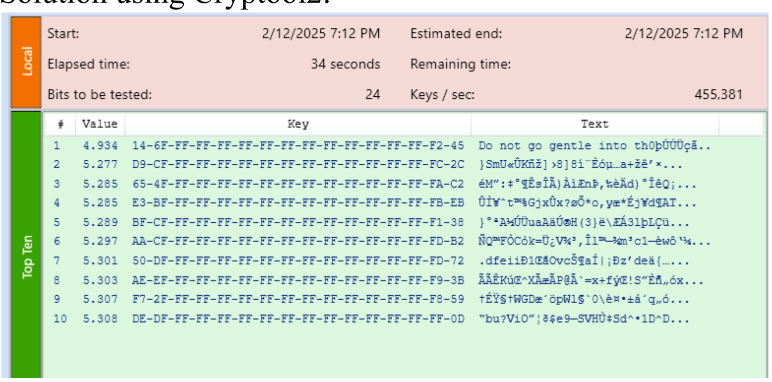
• **Ciphertext:**
E34ADF1C54B33E19518AFABDB36183C37665EE1A234093395325612E97CB773D3975E69D3665C0F65E30BF2D5B9CCFC8B96290954EF6EF3B813A3E50AB637DCFD55D25D775D6360904740BCCEC800C8D5CE03ABAAE18F7D47A6C1AB32C134E3A80DDC77EB2ED1B4FADB2585020B832399B228703B1654BA09645E8ACC7A6B9DC87DAAE6AB576763D62A9EB7585AAFA16

• **Key Pattern:** ***FFFFFFFFFFFFFFFFFFFFFFFFFF*** (all 'F's except for the first and last 12 bits).

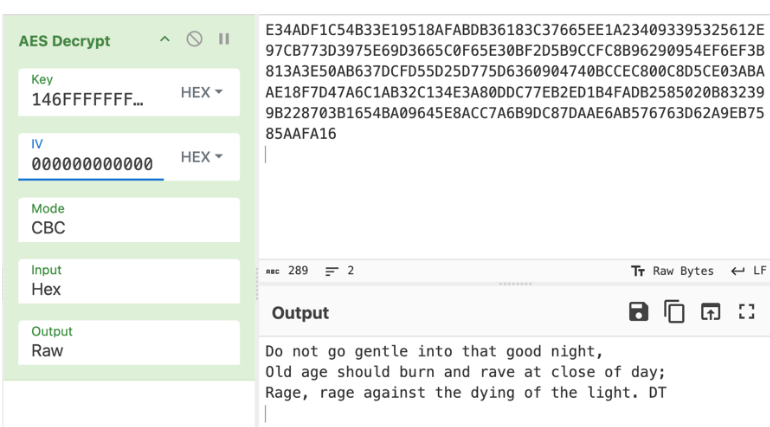
• **Hint:** Don't assume it's in ECB mode.

• **Objective:** Decrypt the ciphertext. The flag is the first four words of the decrypted text.

Solution using Cryptool2:



Verify in Cyberchef:



While CrypTool 2 is a powerful tool for cryptanalysis, it has limitations in certain scenarios. For instance, in Challenge C 201.2, where the Initialization Vector (IV) plays a crucial role, CrypTool 2 may not provide the flexibility to modify the IV as needed. In such cases, alternative tools like CyberChef can be beneficial.

Exploring AES Encryption: ECB vs. CBC Modes in Python

In this tutorial, we'll explore the differences between **AES encryption modes**, specifically **Electronic Codebook (ECB)** and **Cipher Block Chaining (CBC)**, using **Python 2**. We'll demonstrate how **ECB mode fails to obscure patterns** in encrypted data, whereas **CBC mode provides stronger security** by eliminating these patterns.

Original Project source: https://samsclass.info/141/proj/C202.htm

1. **Generating tux_ecb.bmp (ECB Mode)**

The following Python 2 script encrypts tux.bmp using **AES in ECB mode**, creating tux_ecb.bmp:

```
#!/usr/bin/env python2
from Crypto.Cipher import AES

# Ensure key is 16 bytes
key = "aaaabbbbccccdddd"

# Initialize AES cipher in ECB mode
cipher = AES.new(key, AES.MODE_ECB)

# Read the original BMP file
try:
    with open("tux.bmp", "rb") as f:
        clear = f.read()
except IOError:
    print("[ERROR] Could not find 'tux.bmp'. Make sure it's in the same directory.")
    exit(1)

# Ensure input length is a multiple of 16
clear_trimmed = clear[64:-2]  # Remove BMP header and last 2 bytes

# Verify that the length is now a multiple of 16
if len(clear_trimmed) % 16 != 0:
    print("[ERROR] Trimmed data is not a multiple of 16 bytes.")
    exit(1)

# Encrypt the trimmed BMP data
ciphertext_trimmed = cipher.encrypt(clear_trimmed)

# Reconstruct the encrypted BMP file (keep header and last 2 bytes)
ciphertext = clear[:64] + ciphertext_trimmed + clear[-2:]

# Save the encrypted image
output_file = "tux_ecb.bmp"
with open(output_file, "wb") as f:
    f.write(ciphertext)

print("[SUCCESS] Encrypted image saved as '{}'".format(output_file))
```

2. Verification

After image was created, verify both the original and encrypted image:

```
└─$ python2 aes_tux.py
[SUCCESS] Encrypted image saved as 'tux_ecb.bmp'

┌──(kali2㉿kali2)-[~/Desktop]
└─$ xxd -l 64 tux_ecb.bmp
00000000: 424d b266 0e00 0000 0000 3600 0000 2800  BM.f......6...(.
00000010: 0000 b901 0000 e9fd ffff 0100 2000 0000  ............ ...
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 cccc ccff cccc ccff cccc  ................

┌──(kali2㉿kali2)-[~/Desktop]
└─$ xxd -l 64 tux.bmp
00000000: 424d b266 0e00 0000 0000 3600 0000 2800  BM.f......6...(.
00000010: 0000 b901 0000 e9fd ffff 0100 2000 0000  ............ ...
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 cccc ccff cccc ccff cccc  ................
```

3. Viewing the Last 4 Bytes in Hex

Run the following code to extract the last **4 bytes** from tux_ecb.bmp:

```
>>> with open("tux_ecb.bmp", "rb") as f:
    bytes = f.read()
...
>>>
>>> for c in bytes[-4:]:
  print hex(ord(c)),
...
0x3e 0xc9 0xcc 0xff
```

Flag for C202.1: 3EC9CCFF

4. Generating tux_cbc.bmp (CBC Mode)

This script encrypts tux.bmp using **AES in CBC mode**, creating tux_cbc.bmp:

```
#!/usr/bin/env python2

from Crypto.Cipher import AES

# Ensure key and IV are 16 bytes (AES block size)
key = "aaaabbbbccccdddd"  # AES requires a 16-byte key
iv = "0000111122223333"   # 16-byte IV for CBC mode

# Initialize AES cipher in CBC mode
cipher = AES.new(key, AES.MODE_CBC, iv)

# Read the original BMP file
try:
    with open("tux.bmp", "rb") as f:
        clear = f.read()
except IOError:
    print("[ERROR] Could not find 'tux.bmp'. Make sure it's in the same directory.")
    exit(1)

# Trim image data to ensure it is a multiple of 16 bytes
clear_trimmed = clear[64:-2]  # Remove BMP header and last 2 bytes

# Verify that the length is now a multiple of 16
if len(clear_trimmed) % 16 != 0:
    print("[ERROR] Trimmed data is not a multiple of 16 bytes.")
    exit(1)

# Encrypt the trimmed BMP data using CBC mode
ciphertext_trimmed = cipher.encrypt(clear_trimmed)

# Reconstruct the encrypted BMP file (keep header and last 2 bytes)
ciphertext = clear[:64] + ciphertext_trimmed + clear[-2:]



# Save the encrypted image

output_file = "tux_cbc.bmp"

with open(output_file, "wb") as f:

    f.write(ciphertext)

print("[SUCCESS] Encrypted image saved as '{}'".format(output_file))

# Extract the last 4 bytes in hex format

last_4_bytes = ciphertext[-4:]

hex_values = " ".join(hex(ord(c))[2:].upper().zfill(2) for c in last_4_bytes)



print("[FLAG] Last 4 bytes in hex: {}".format(hex_values))
```

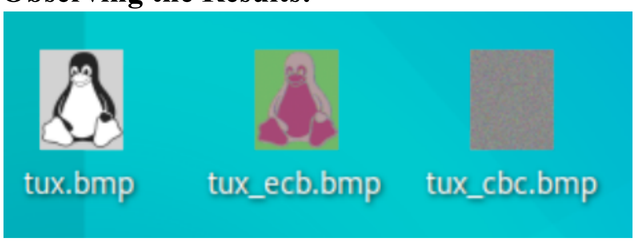
5. **Viewing the Last 4 Bytes in Hex**

Run the following code to extract the last **4 bytes** from tux_cbc.bmp:

```
>>> with open("tux_cbc.bmp", "rb") as f:
  bytes = f.read()
...
>>> for c in bytes[-4:]:
    print hex(ord(c)),
...
0xcc 0xe 0xcc 0xff
```

Flag for C202.2 CC0ECCFF

Observing the Results:



After generating both images, open them with an **image viewer** and compare:

✅ **tux_ecb.bmp (ECB Mode)**

• The encrypted image still reveals **patterns resembling the original image**.
• This demonstrates **ECB mode's weakness**, as it **encrypts identical blocks the same way**.

✅ **tux_cbc.bmp (CBC Mode)**

• The encrypted image **appears as random noise**.
• **CBC mode improves security** by **chaining blocks together**, ensuring identical plaintext blocks encrypt differently.

Conclusion:

This exercise highlights the importance of choosing the appropriate **AES encryption mode**:

• **ECB mode is insecure** because **identical plaintext blocks produce identical ciphertext blocks**, leading to pattern retention.
• **CBC mode eliminates patterns** by **XORing each block with the previous one**, improving security.

For real-world encryption, **CBC (or stronger modes like GCM) should always be used instead of ECB** to prevent data leaks.