Đại học Quốc gia Thành phố Hồ Chí Minh
Trường Đại học Công nghệ Thông tin (UIT)

BÁO CÁO BÀI TẬP

Môn học: Mật mã học

Kỳ báo cáo: Buổi 05 (Session 05)

Tên chủ đề: Ôn tập

Ngày báo cáo: 22/05/2023

1. THÔNG TIN CHUNG:

Lớp: NT219.N22.ATCL.1

STT	Họ và tên	MSSV	Email
1	Đoàn Hải Đăng	21520679	21520679@gm.uit.edu.vn
2	Lê Thanh Tuấn	21520518	21520518@gm.uit.edu.vn
3	Phan Thị Hồng Nhung	21521250	21521250@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:

STT	Công việc	Kết quả tự đánh giá	Người đóng góp
1	Bài tập 1	100%	Hồng Nhung
2	Bài tập 2	100%	Hồng Nhung
3	Bài tập 3	100%	Hải Đăng
4	Bài tập 4	100%	Hải Đăng
5	Bài tập 5	100%	Thanh Tuấn
6	Bài tập 6	100%	Thanh Tuấn
7	Bài tập 7	100%	Cả nhóm
8	Bài tập 8	100%	Cả nhóm

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

BÁO CÁO CHI TIẾT

1. Bài tập 1

Weak keys and semi-weak keys

- A weak key in DES is a key that results in an encryption algorithm that is less secure than anticipated. It makes the encryption process predictable, which facilitates easier decryption and retrieval of the original plaintext by an attacker.
- A semi-weak key is a key that generates encryption algorithms with a limited number of weak encryption rounds. Although the encryption process is not entirely predictable, it still has vulnerabilities that can be exploited by an attacker.

Cases that can lead to weak key in DES

- Using easily guessable passwords: If users utilize weak or easily guessable passwords as the key for DES, it can be easily attacked.
- Short key length: If the password used is too short (less than 56 bits), it can be brute-forced to find the key within a reasonable time.
- Key repetition: If two or more systems use the same key, attacking one system may reveal the key to the other systems.

2. Bài tập 2

Find the plaintext:

In the previous section, weak keys were mentioned, so I found 8 possible weak keys on Wikipedia and tried each one of them.

- Alternating ones + zeros (0x0101010101010101)
- Alternating 'F' + 'E' (0xFEFEFEFEFEFEFEFE)
- '0xE0E0E0E0F1F1F1F1'
- '0x1F1F1F1F0E0E0E0E'

If an implementation does not consider the parity bits, t

- all zeros (0x0000000000000000)
- all 'F' (0xFFFFFFFFFFFFFFFF)
- '0xE1E1E1E1F0F0F0F0'
- '0x1E1E1E1E0F0F0F0F'

Define the **decrypt_des** function:

- This function takes three parameters: key (DES key), iv (initialization vector), and ciphertext (the encrypted message).

- The cipher object is set to use the CBC mode (Cipher Block Chaining) for decryption.
- The ciphertext is decrypted using the decrypt method of the cipher object.

Define the **main** function:

- The base64 encoded ciphertext is decoded using b64decode and stored in the ciphertext variable.
- The decrypt_des function is called with the key, IV, and ciphertext to obtain the decrypted plaintext.
- The decrypted plaintext is printed to the console.
- Finally, I obtained the correct result as shown in the output.

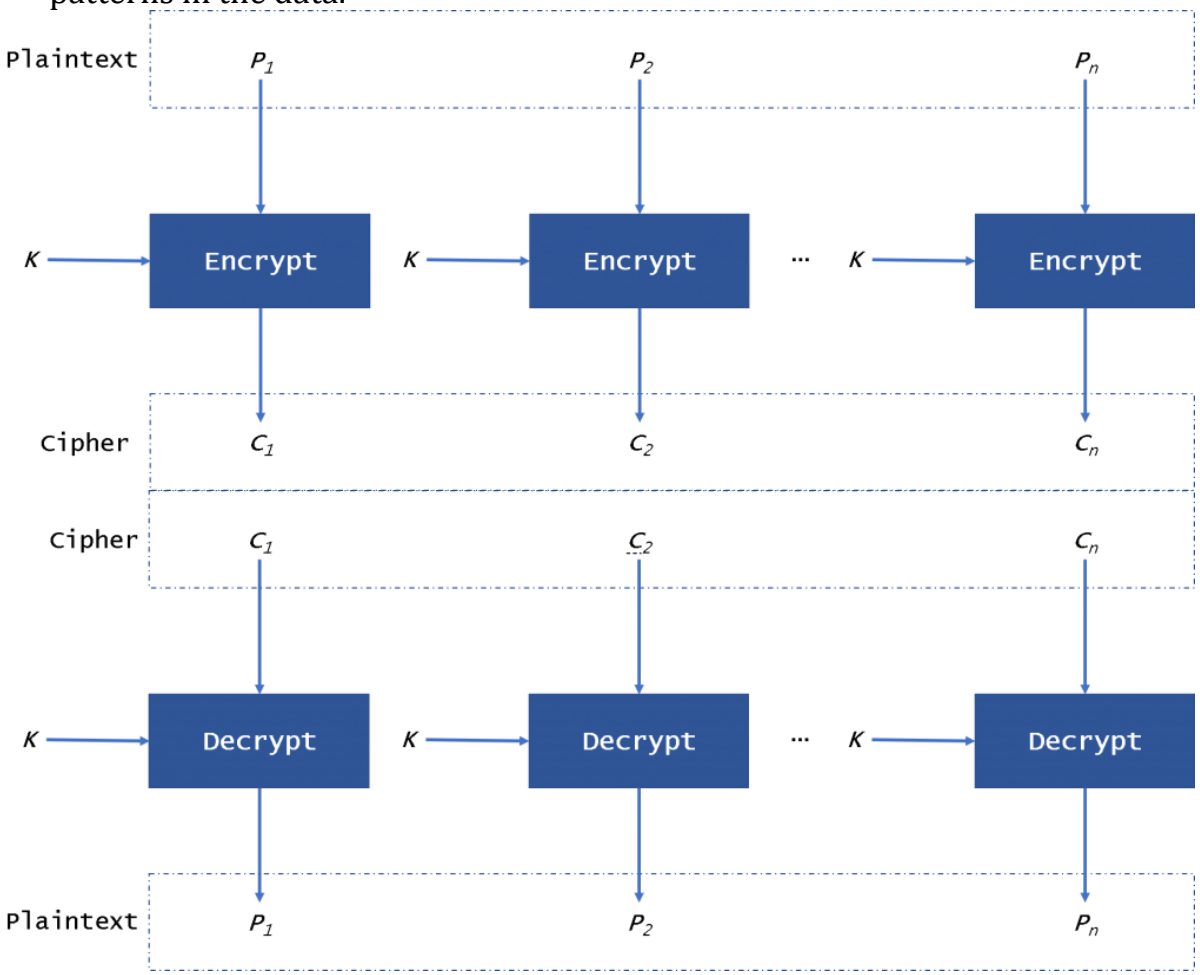
```
Cau2.py > ...
1   from base64 import b64decode
2   from Crypto.Cipher import DES
3
4   def decrypt_des(key, iv, ciphertext):
5       cipher = DES.new(key, DES.MODE_CBC, iv)
6       plaintext = cipher.decrypt(ciphertext)
7       return plaintext.rstrip(b'\0')
8
9   def main():
10      ciphertext = b64decode("jtEl85W3Riqjk56bj+7J5YcYhHvzHc6d")
11      iv = b64decode("VyUR14UQP/0=")
12      key = b"\xE0\xE0\xE0\xE0\xF1\xF1\xF1\xF1"
13
14      plaintext = decrypt_des(key, iv, ciphertext)
15      print("Plaintext:", plaintext.decode('utf-8'))
16
17  if __name__ == '__main__':
18      main()

PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL

PS D:\Nam2\MMH\Cryptography_Lab\Lab_5> python -u "d:\Nam2\MMH\Cryptography_Lab\Lab_5\Cau2.py"
Plaintext: review DES - weak KEY♥♥♥
```

Plaintext: **review DES - weak KEY♥♥♥**

3. Bài tập 3

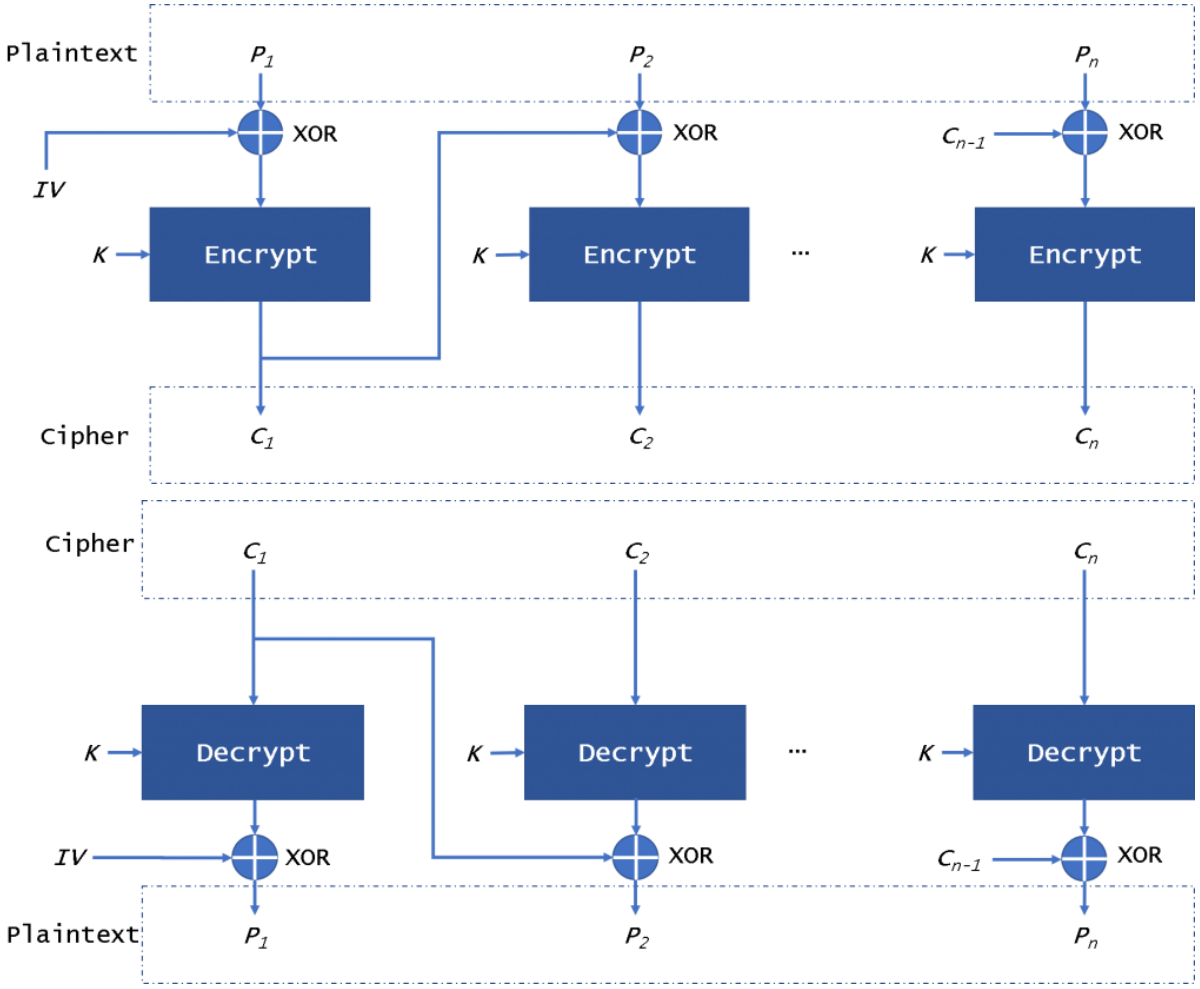
Differences between the modes on AES (ECB, CBC, OFB, CFB, CTR, XTS, CCM, GCM)

- *ECB:* Divides the plaintext into blocks and encrypts each block independently using the same key. Lacks diffusion and is vulnerable to patterns in the data.



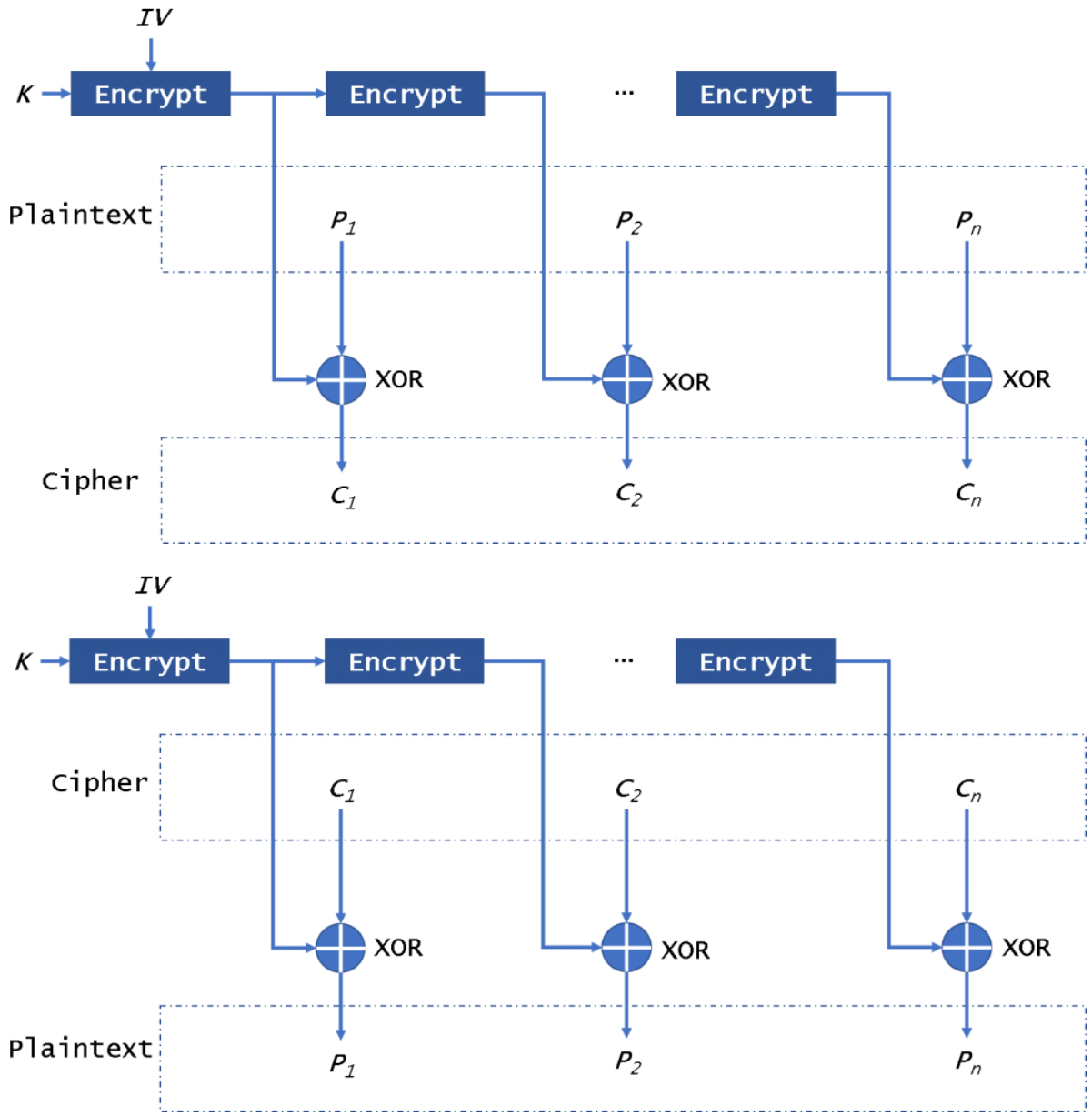
If we encrypt the same plaintext, we will get the same ciphertext. So there is a high risk in this mode. And the plaintext and ciphertext blocks are a one-to-one correspondence.

- *CBC:* XORs each plaintext block with the previous ciphertext block before encryption. Requires an initialization vector (IV) and provides better security than ECB.



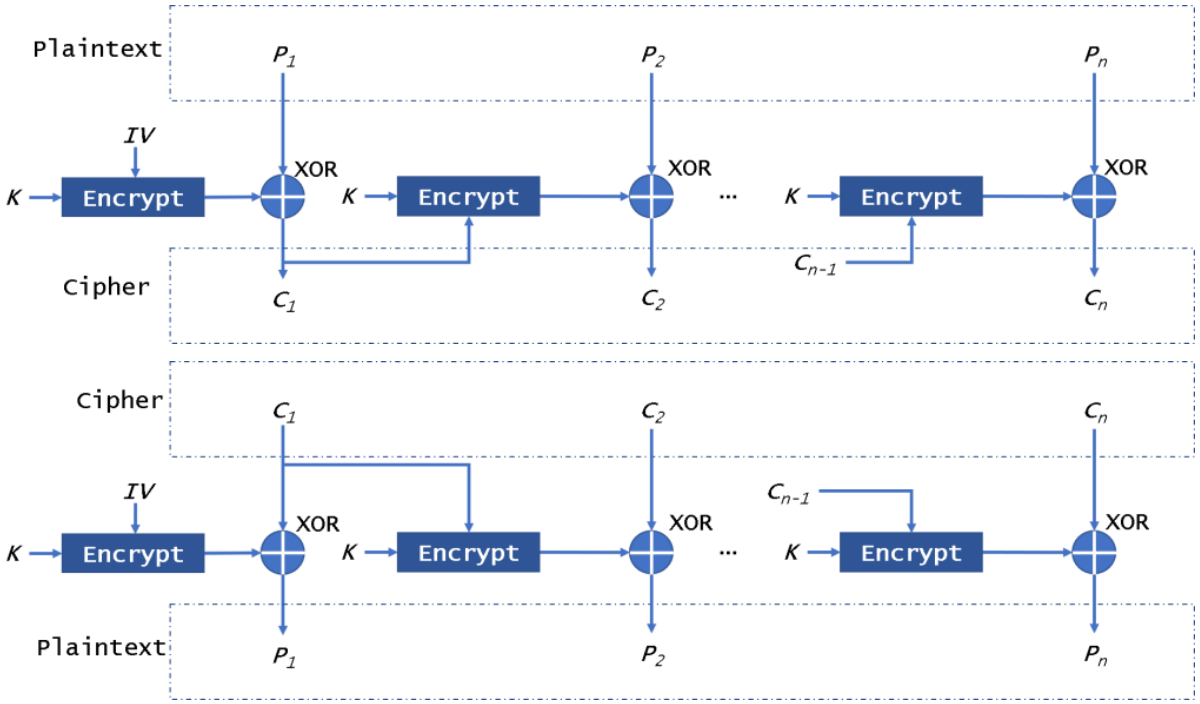
In this mode, even if we encrypt the same plaintext block, we will get a different ciphertext block. We can decrypt the data in parallel, but it is not possible when encrypting data. If a plaintext or ciphertext block is broken, it will affect all following block.

- *OFB:* Encrypts a randomly generated IV using the key to create a keystream, which is then XORed with the plaintext to produce the ciphertext. Allows for random access and error propagation.



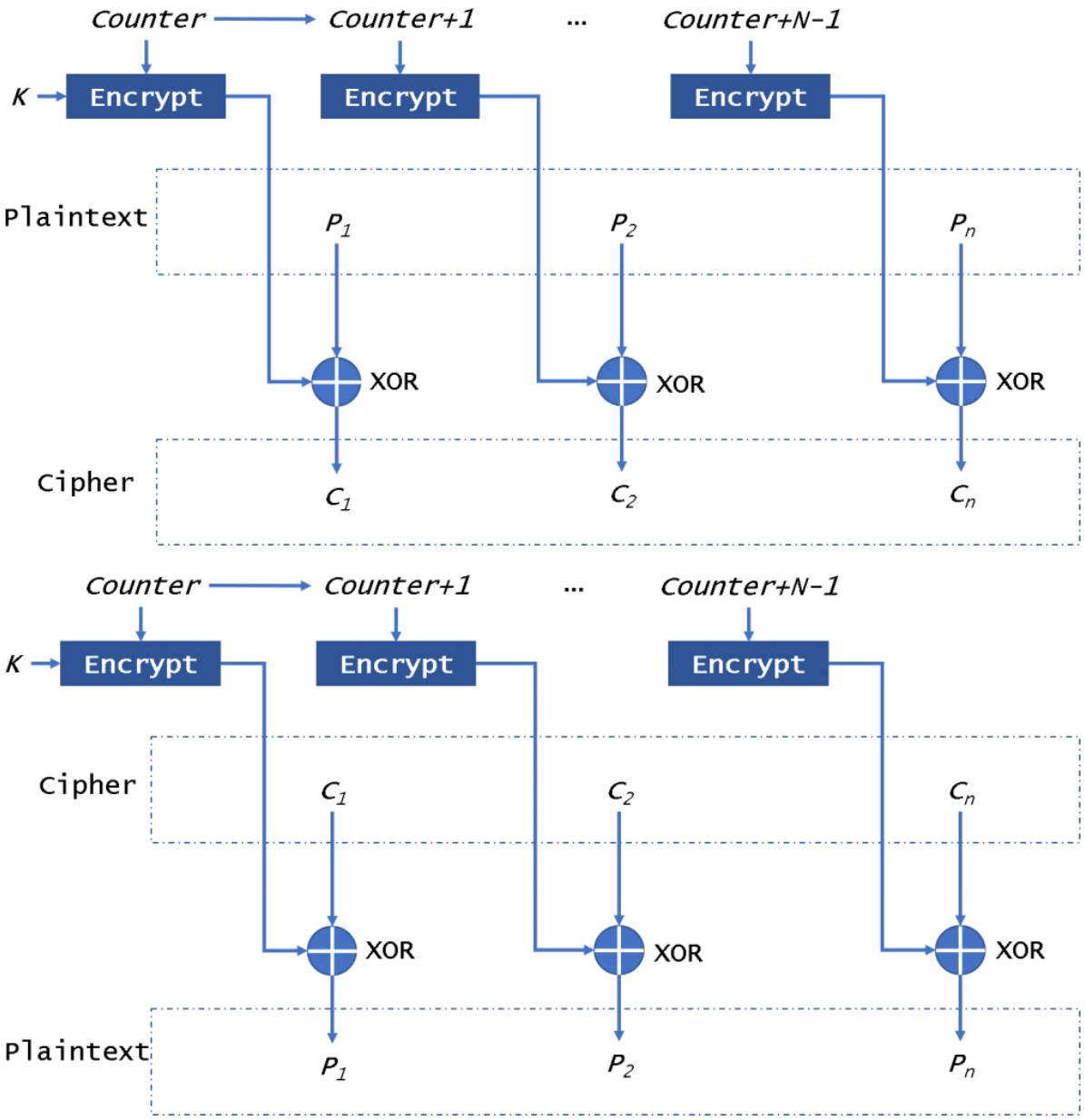
In this mode, it will encrypt the IV in the first time and encrypt the per-result. Then it will use the encryption results to xor the plaintext to get ciphertext. It will not be affected by the broken block.

- *CFB:* Similar to OFB, but the ciphertext of the previous block is fed back into the encryption process. Offers random access and error propagation but may suffer from a short delay.



This mode will not encrypt plaintext directly, it just uses the ciphertext to xor with the plaintext to get the ciphertext. So in this mode, it doesn't need to pad data. If there is a broken block, it will affect all following block.

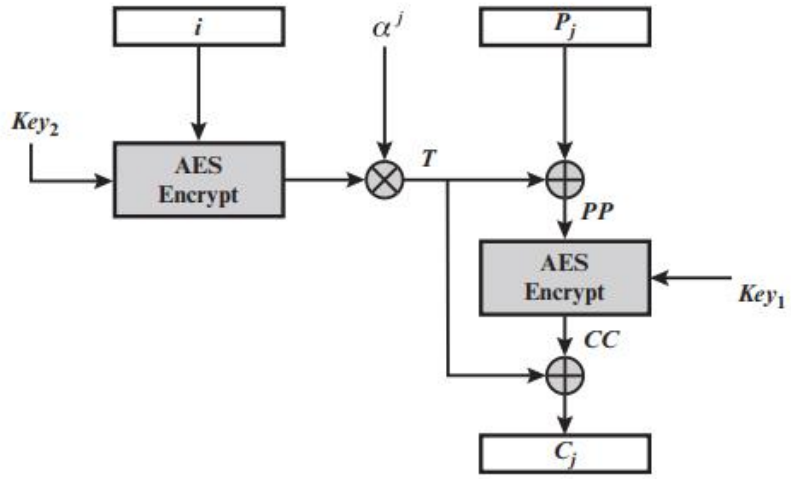
- *CTR:* Converts the block cipher into a stream cipher. Uses a counter value and an IV to generate a keystream, which is then XORed with the plaintext. Provides parallel encryption and random access.



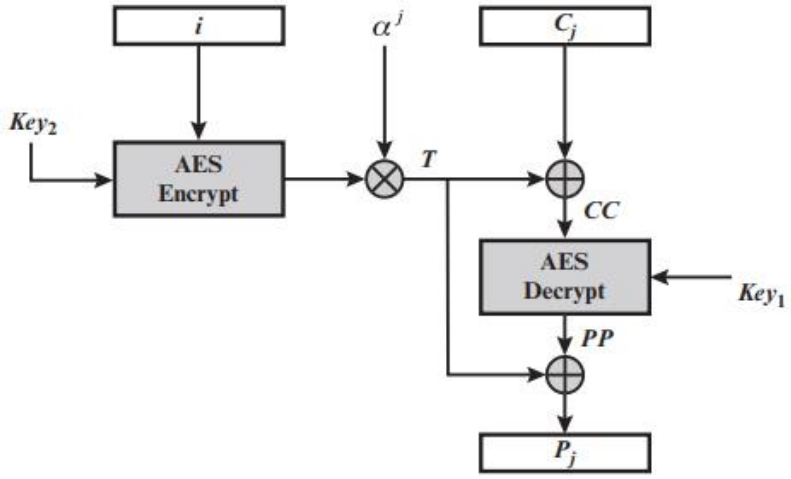
The counter has the same size as the used block. All encryption blocks use the same encryption key. As this mode, It will not be affected by the broken block.

- *XTS:* Specifically designed for disk encryption. Applies two independent AES operations to each data unit and provides protection against data manipulation.

XTS uses two AES keys. One key is used to perform the AES block encryption; the other is used to encrypt what is known as a "Tweak Value". This achieves the goal of each block producing unique cipher text given identical plain text without the use of initialization vectors and chaining. In effect, the text is almost double-encrypted using two independent keys. Decryption of the data is accomplished by reversing this process. Since each block is independent and there is no chaining, if the stored cipher data is damaged and becomes corrupted, only the data for that particular block will be unrecoverable.



(a) Encryption

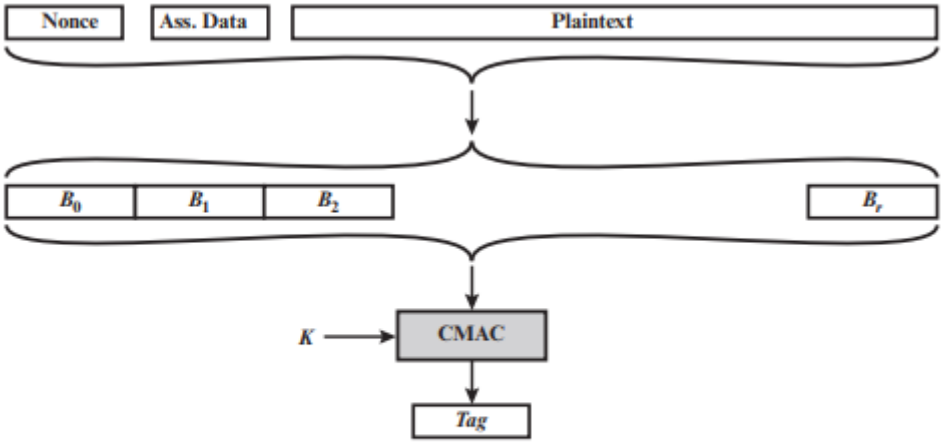


(b) Decryption

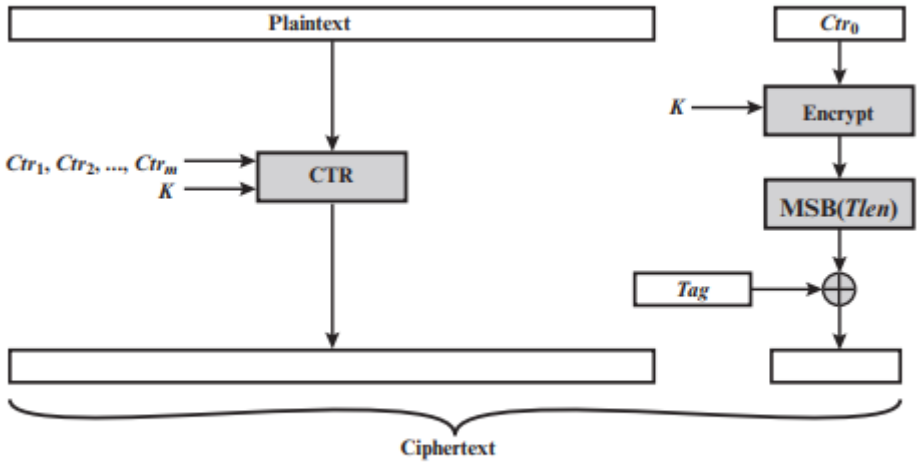
Figure 7.10 XTS-AES Operation on Single Block

- *CCM:* Combines CTR mode for encryption and CBC-MAC for message authentication. Suitable for low-resource devices and provides encryption and authentication within a single pass.

In CCM, the plaintext is divided into blocks, and each block is encrypted using the counter mode, similar to AES-CTR. A unique counter value and an initialization vector (IV) are used to generate a keystream, which is then XORed with the plaintext to produce the ciphertext.



(a) Authentication



(b) Encryption

Figure 12.9 Counter with Cipher Block Chaining-Message Authentication Code (CCM)

- *GCM:* Combines CTR mode with Galois field multiplication for authentication. Provides strong security, parallel encryption, and authentication within a single pass.

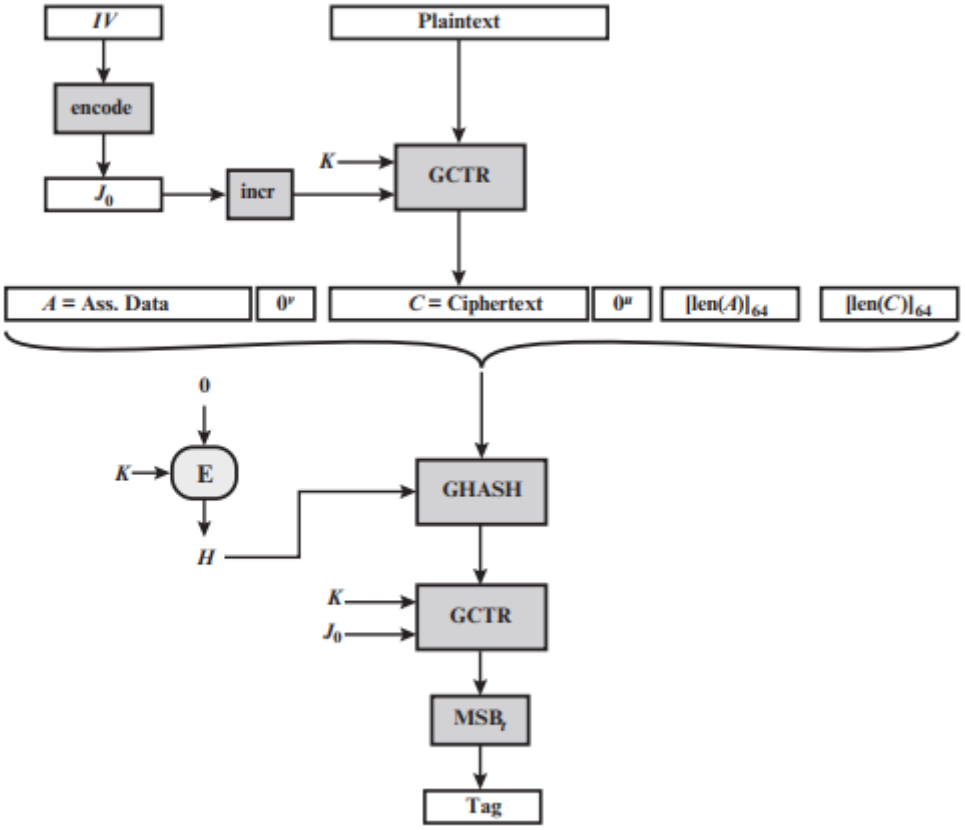


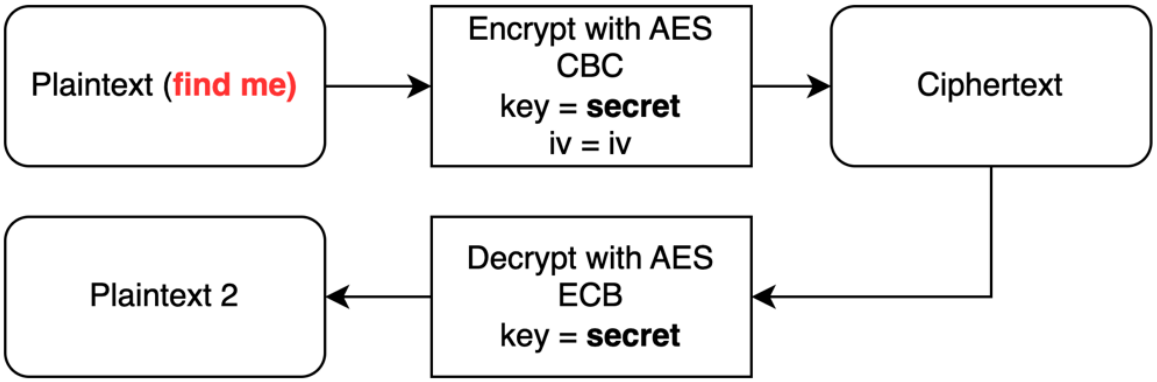
Figure 12.11 Galois Counter—Message Authentication Code (GCM)

In essence, the message is encrypted in variant of CTR mode. The resulting ciphertext is multiplied with key material and message length information over GF(2^{128}) to generate the authenticator tag. The standard also specifies a mode of operation that supplies the MAC only, known as GMAC.

The GCM mode makes use of two functions: GHASH, which is a keyed hash function, and GCTR, which is essentially the CTR mode with the counters determined by a simple increment by one operation.

4. Bài tập 4

Find the plaintext of the program below:




```
CryptoPP::StringSource(ivBase64, true,
    new CryptoPP::Base64Decoder(
        new CryptoPP::StringSink(iv)
    )
);

CryptoPP::StringSource(ciphertextBase64, true,
    new CryptoPP::Base64Decoder(
        new CryptoPP::StringSink(ciphertext)
    )
);

CryptoPP::StringSource(plaintext2Base64, true,
    new CryptoPP::Base64Decoder(
        new CryptoPP::StringSink(plaintext2)
    )
);
```

These lines initialize and decode the Base64-encoded strings representing the IV (Initialization Vector), ciphertext, and plaintext2. Decode the Base64-encoded data into the corresponding byte arrays.

```
for (size_t i = 0; i < ciphertext.size(); i += 16)
{
        std::string block = plaintext2.substr(i, 16);
        CryptoPP::xorbuf(reinterpret_cast<CryptoPP::byte*>(&block[0]),
            reinterpret_cast<const CryptoPP::byte*>(&t[0]),
            reinterpret_cast<const CryptoPP::byte*>(&block[0]), 16);
        res += block;
        t = ciphertext.substr(i, 16);
}
```

This loop performs the XOR decryption operation. It iterates over the ciphertext in blocks of 16 bytes. For each block, it takes the corresponding 16-byte

block from **plaintext2** and XORs it with the previous block's value (initialized as the IV). The result is appended to the **res** string, and the current ciphertext block becomes the new "previous block" (**t**) for the next iteration.

```
PS D:\Selfwork\Crypto> .\ex4_AES.exe
review AES --- xor with ECB♣♣♣♣♣
```

Plaintext: **review AES --- xor with ECB♣♣♣♣♣**

5. Bài tập 5

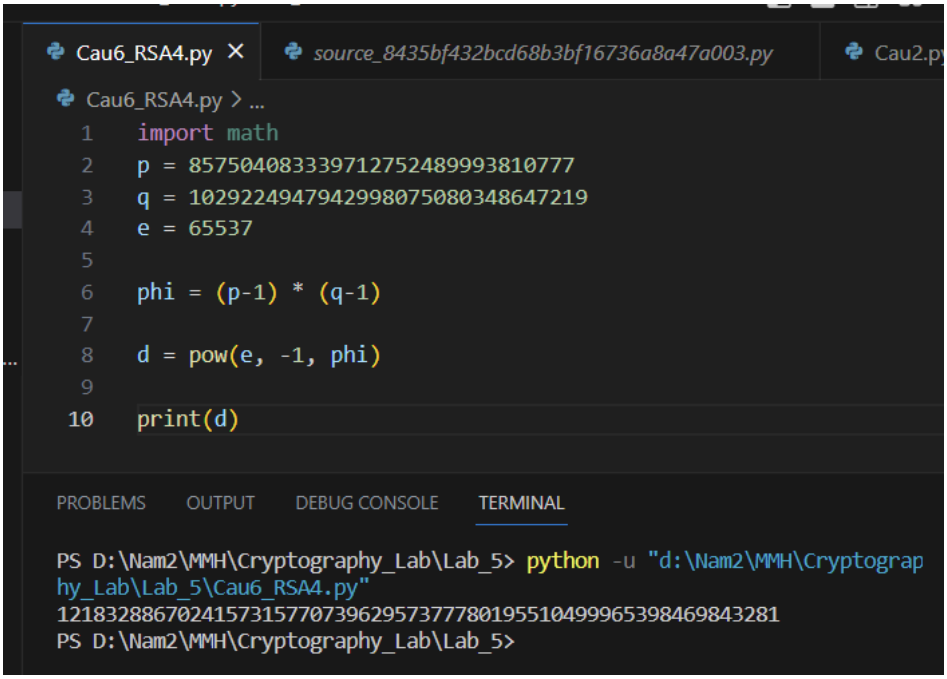
☞ In the case of using the RSA algorithm, if a small number of repeated encryption operations result in plaintext, the possible reasons could be:

- Software execution errors: Errors in the software implementation can lead to unexpected outcomes, including returning plaintext instead of encrypted data.
- Incorrect public key: If the public key used for encryption is not correct, it could lead to successful decryption without requiring the private key.
- Error in the encryption/decryption process: Mistakes in the encryption or decryption process, such as applying the wrong formulas or procedures, could result in incorrect outcomes.

6. Bài tập 6

RSA Starter 5

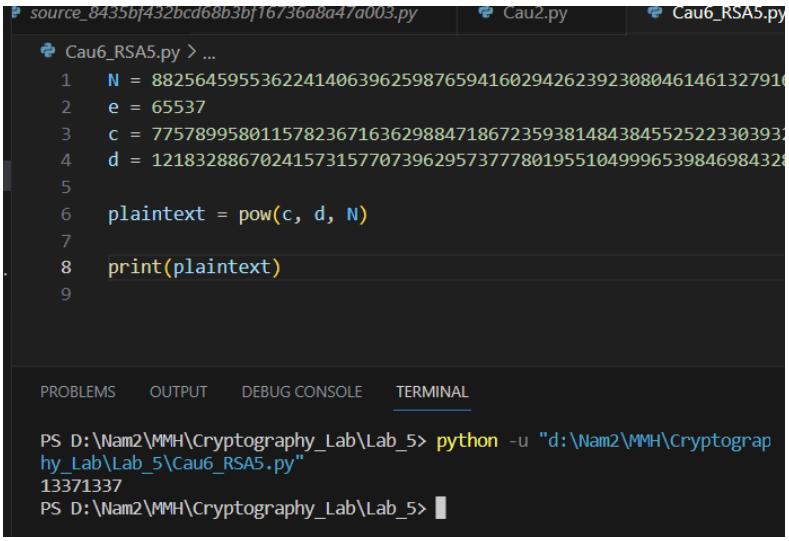
To solve this challenge, we must have the private key (d) in the previous challenge (RSA Starter 4).



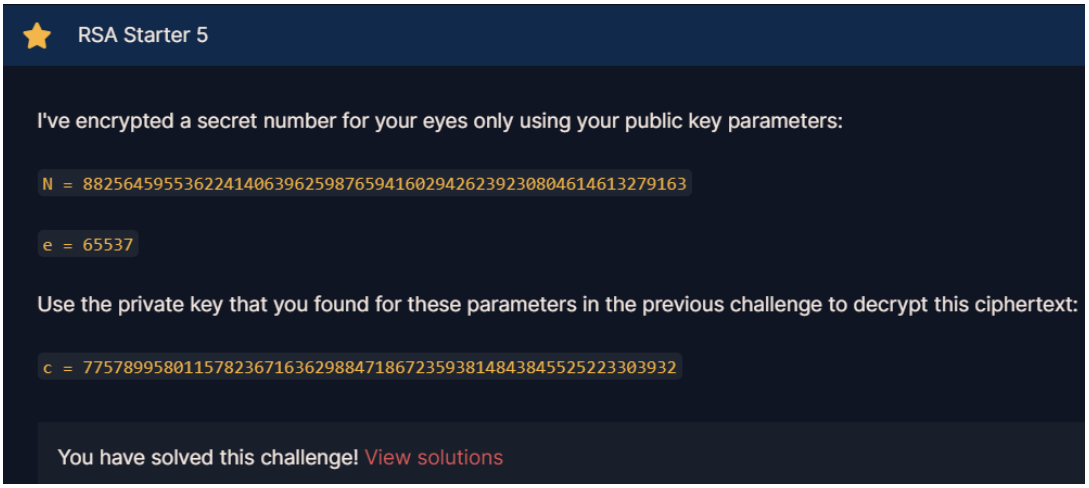
Plaintext = c^d mod N

In the code, we use the pow() function with three arguments to perform modular exponentiation. The first argument is the ciphertext "c", the second argument is the private key "d", and the third argument is the modulus "N". The result is assigned to the variable plaintext, which represents the decrypted secret number.

After running this code, we have the flag = 13371337

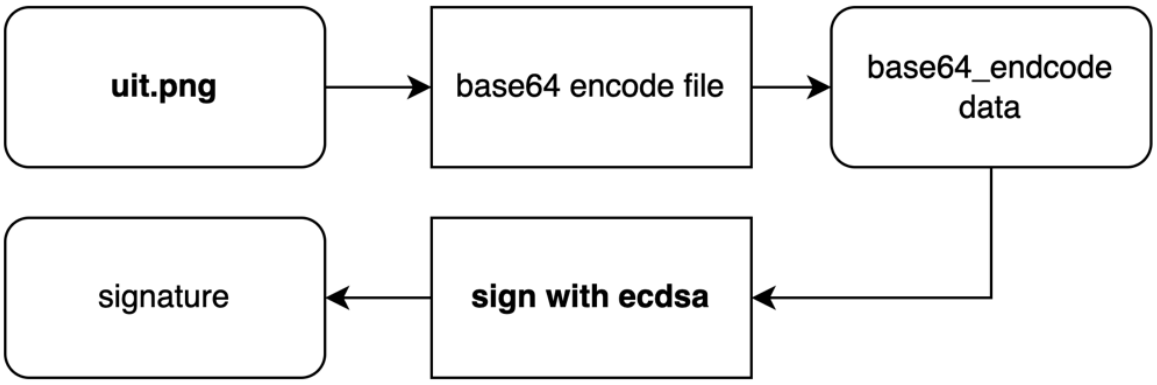


Result:



7. Bài tập 7

Check which signature is correct for the file "uit.png":



Load publicKey and message from file given in Courses:

```
//Load public key
ECDSA<ECP, SHA1>::PublicKey publicKey;
LoadPublicKey("ec.public.key", publicKey);

//Load message
std::ifstream inputFile("uit.png", std::ios::binary);
std::vector<char> msbytes(
     (std::istreambuf_iterator<char>(inputFile)),
     (std::istreambuf_iterator<char>()));
inputFile.close();
string message(msbytes.begin(),msbytes.end());
```

Encode message loaded from file:

```
//Encode message
```

```
    string encoded;
    CryptoPP::StringSource(message, true,
    new CryptoPP::Base64Encoder(new CryptoPP::StringSink(encoded)));
```

The code below is used to check whether the message we encode is similar to 10 signatures given:

- The **lsdir** variable is a vector of strings that contains the filenames or signatures to be verified.
- The **sigbytes** vector is converted into a string format (**signature**) using its constructor, which takes iterators pointing to the beginning and end of the **sigbytes** vector.
- The **VerifyMessage** function is called with three arguments: **publicKey**, **encoded**, and **signature**. This function is responsible for performing the RSA signature verification process. The function returns a boolean value indicating whether the verification was successful or not.

```
  //Verify message
  std::vector<string> lsdir = {"0eccaf9b4a71a84fc1f36733d47c5147",
"6dc251932a97988e3a420f2bbe9143aa", "8b61860d79c2a96ec81829a68d8060ef",
"0256f2c4a2666f0e795c216e6c90f9f3", "369e5feed49bfa0de17b40f9f939d566",
"566e9c1e95f786c06570a556e83abdc7", "a7f6df12eeb641b179b9c885b5ba262b",
"ab5f656849484cef6b157b5ededf4cf9", "ccd26dcb9c0a9196d708fa3cfa5c25eb",
"d20f6fa3834061f881ec816ca7afb35a"};
  for (int i = 0; i < 10; i++){
      bool result = false;
      string filename = lsdir[i];

      std::ifstream siginp(filename, std::ios::binary);
      std::vector<char> sigbytes(
          (std::istreambuf_iterator<char>(siginp)),
          (std::istreambuf_iterator<char>()));
      siginp.close();

      string signature(sigbytes.begin(),sigbytes.end());

      result = VerifyMessage( publicKey, encoded, signature );
      string result_str = (result) ? "true" : "false";
      cout << filename << " " << result_str << endl;
}
```

Result:

```
PS D:\Selfwork\Crypto> .\ex7_ECC.exe
0eccaf9b4a71a84fc1f36733d47c5147 false
6dc251932a97988e3a420f2bbe9143aa true
8b61860d79c2a96ec81829a68d8060ef true
0256f2c4a2666f0e795c216e6c90f9f3 true
369e5feed49bfa0de17b40f9f939d566 false
566e9c1e95f786c06570a556e83abdc7 true
a7f6df12eeb641b179b9c885b5ba262b false
ab5f656849484cef6b157b5ededf4cf9 false
ccd26dcb9c0a9196d708fa3cfa5c25eb false
d20f6fa3834061f881ec816ca7afb35a true
```

⇨ The image has 5 signature

8. Bài tập 8

In this challenge, we get a hash function, and we need to find a collision.

```python
def pad(data):
    padding_len = (BLOCK_SIZE - len(data)) % BLOCK_SIZE
    return data + bytes([padding_len]*padding_len)
```

This function adds padding to the current block until it reaches the full block size. It does this by appending the required number of missing bytes, similar to the padding scheme used in PKCS#7. However, the flaw is that if the block is already the correct size, no additional bytes are added.

If we find 2 messages which have identical outputs after the padding, then the rest of the function will be completed in the same way for both messages, thus generating the same output.

For example, if we send the hex "1f" and the hex "1f1f1f...1f1f1f" which is of length 32 then the output will be the same,

"1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f".

Finally, the JSON is
{"m1":"1f","m2":"1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f"}

Result:

```
┌──(rosy㉿Rosy)-[~]
└─$ nc socket.cryptohack.org 13405
Can you help beta test our new CryptoHash? If you find a collision, we'll give you a flag!

Please send two hex encoded messages m1, m2 formatted in JSON: {"m1":"1f","m2":"1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f"}
{"flag": "Oh no! Looks like we have some more work to do... As promised, here's your flag: crypto{Always_add_padding_even_if_its_a_whole_block!!!}"}
```


Hash Stuffing
50 pts • 523 Solves • 11 Solutions
With all the attacks on MD5 and SHA1 floating around, we thought it was time to start rolling our own hash algorithm. We've set the block size to 256 bits, so I doubt anyone will find a collision.
Connect at nc socket.cryptohack.org 13405
Challenge files:
- source.py
You have solved this challenge! View solutions