

## Block Cipher Modes of operation

1. **Introduction to modes of operation**
2. **The Initialization Vector (IV)**
3. **Message padding**
4. **Electronic codebook (ECB)**
5. **Cipher feedback (CFB)**
6. **Output feedback (OFB)**
7. **Putting it all together**

### Introduction to modes of operation

If you followed this tutorial up to this point, you should know by now that block ciphers operate on blocks of fixed size, in this case 128 bits. Although it might happen that we need to encode a message of exactly 128 bits, we want our plaintext to be of any length. As you can see, encrypting the same plaintext under the same key always produces the same output, we need a mode of operation to provide confidentiality for messages of arbitrary length.

In this final chapter, we will learn about three modes of operation, namely CBC, OFB and CFB.

### The Initialization Vector (IV)

Every block cipher mode of operation requires an initialization vector (commonly called IV), which serves as basis block to start the process and at the same time introduces randomization in the process. There is no need for the IV to be secret (in most cases), but as you will see soon enough, it might cause a leak of information if it is reused with the same key. Just remember that the IV is a initial block of the same size than our block cipher block (128 bits for AES).

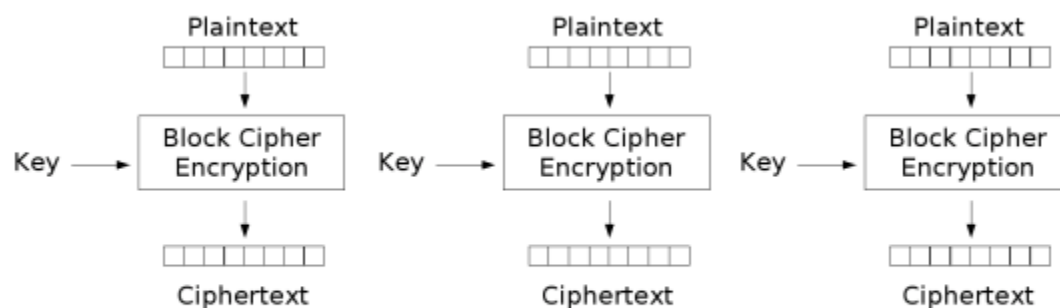
### Message padding

Because a block cipher works on blocks of a fixed size, but messages come in a variety of lengths, some modes (mainly CBC) require that the final block be padded before encryption. As you can read in **[this article](#)**, many padding schemes exist. The simplest

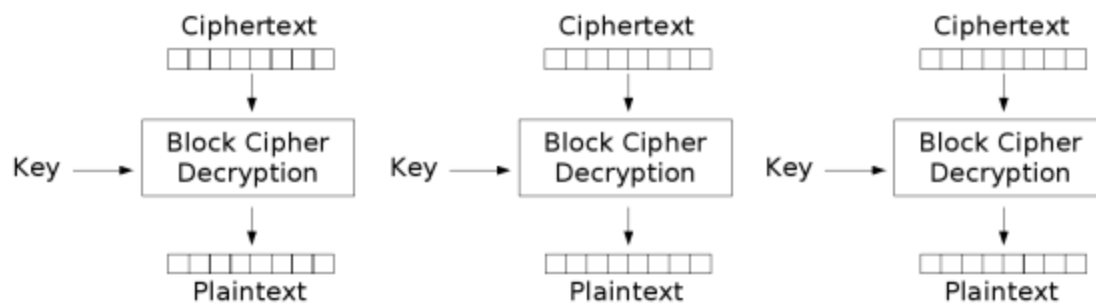
(which we will implement) is to add null bytes to the plaintext to bring its length to a multiple of the block size. Slightly more complex is the original DES method, which is to add a single one bit, followed by enough zero bits to fill out the block. If the message ends on a block boundary, a whole padding block will be added.

## Electronic codebook (ECB)

The simplest (and also the least secure) encryption mode is the electronic codebook (also called ECB). The input message is simply divided into blocks of the required size and each block is encrypted separately. The disadvantage is evident: identical plaintext blocks are encrypted into identical ciphertext blocks (since the secret key stays the same during the entire process). I strongly recommend to not use this mode at all, I'm only showing this for information purpose only.



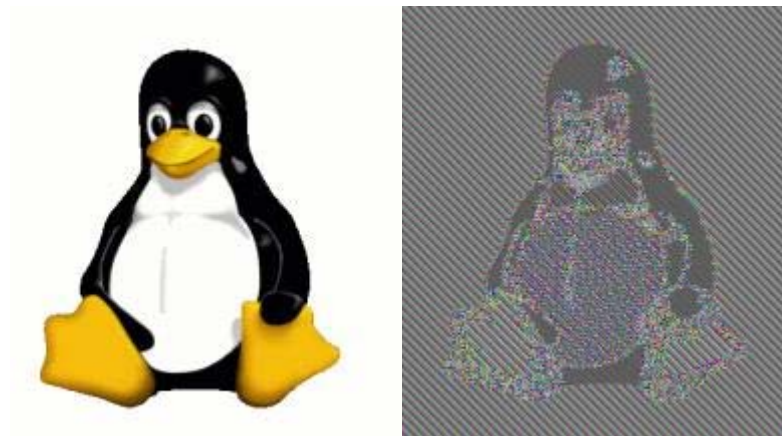
Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

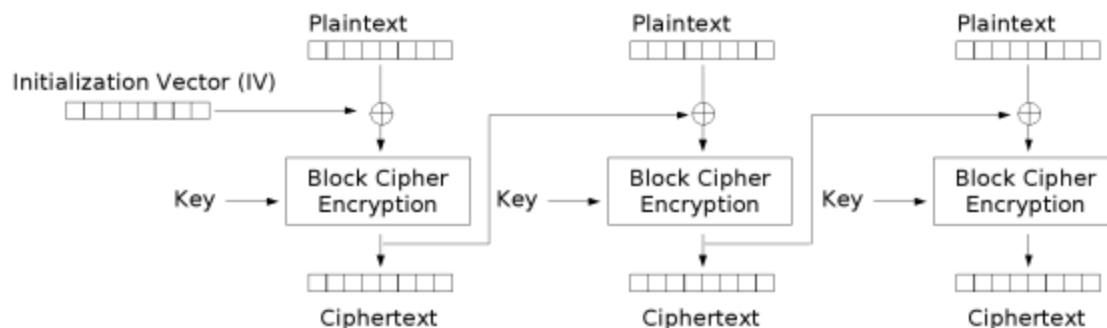
As you can see on these images (taken from this [Wikipedia article](#)), you can see that ECB is fairly simple to implement and even more simple to misuse for malicious purpose. An even more stunning example is the encryption of this image under ECB. Since identical input pattern result in the same output pattern, it is possible to recognize the source

image, even under its encrypted version:

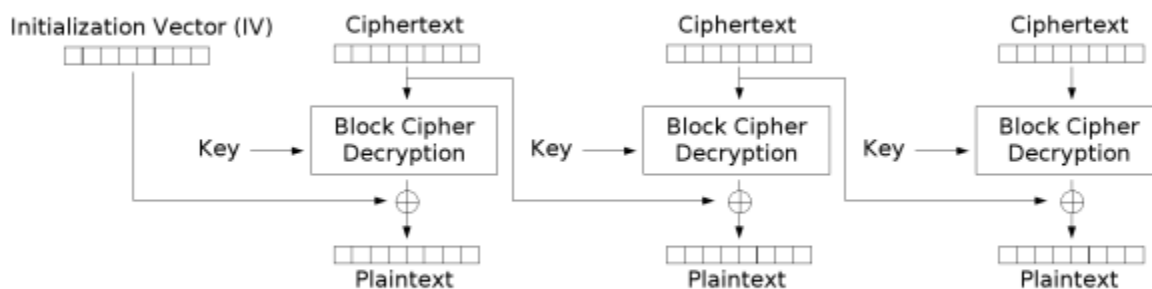


## Cipher-block chaining (CBC)

In CBC mode, the input message is once again divided into blocks and each block is XORed to the previous ciphertext block (the output of the previous encryption) before being encrypted. As you can see, we require a ciphertext block to XOR the first message block to, which will be our IV. In CBC mode, each ciphertext block is dependent on all plaintext blocks processed up to that point.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

CBC is the most commonly used mode of operation but its main drawback is that the

message must be padded to a multiple of the cipher block size, and therefore is being replaced by synchronous stream-ciphers such as CFB or OFB.

Before we start implementing the CBC mode, let us first define a structure that we'll use to select the appropriate mode as well as the prototype of our encryption function:

```
enum modeOfOperation{
    OFB,
    CFB,
    CBC
};

void encrypt(FILE *in, FILE *out, enum modeOfOperation);
```

If at a later stage, you have several block ciphers implemented, you could change this prototype to have a pointer to a block cipher, which would then be used internally by the mode of operation.

I decided to use several blocks of 128 bits, that each serve its very own purpose:

- **plaintext:** this corresponds to the 128 bit plaintext block that we encrypt during each iteration
- **input:** input corresponds to what is being used as input for the block cipher (the AES input)
- **output:** this corresponds to the output of our block cipher
- **ciphertext:** this corresponds to the output of our mode of operation for this iteration

As you will see, we could manage with a less blocks but since each mode of operation has its own special structure, we can keep our code clean and structured by introducing a little overhead. I'm also using an additional variable that defines if we are currently in the first round of the mode (where we use the IV) or at a later stage.

In order to recover the exact plaintext, even using our padding scheme, I decided to first write the mode of operation in use as well as the exact input file size into the output file:

```
fseek(in, 0, SEEK_END);
fileSize = ftell(in);
fseek(in, 0, SEEK_SET);

/* add the file header */
fwrite(&mode, sizeof(mode), 1, out);
fwrite(&fileSize, sizeof(fileSize), 1, out);
```

Since we are working with binary files, I use **fread()** and **fwrite()** to process the two files. Our algorithm reads blocks of 16 bytes (saving the return value of `fread()` in case we read less than 16 bytes and we need to pad), we perform the padding if we need to, XOR the plaintext to either the IV or the output (in CBC, the output is equal to the ciphertext),

apply our AES encryption and write the result to the file.

```

/* the non-expanded keySize */
enum keySize size = SIZE_32;

/* the AES input/output */
unsigned char plaintext[16] = {0};
unsigned char input[16] = {0};
unsigned char output[16] = {0};
unsigned char ciphertext[16] = {0};
unsigned char IV[16] = {0};

/* the AES key */
unsigned char key[32] = {0x0};

/* char firstRound */
char firstRound = 1;

size_t read;
int i;

while ((read = fread(plaintext, sizeof(unsigned char), 16, in)) > 0)
{
    /* padd with 0 bytes */
    if (read < 16)
    {
        for (i = read; i < 16; i++)
            plaintext[i] = 0;
    }

    for (i = 0; i < 16; i++)
    {
        input[i] = plaintext[i] ^ ((firstRound) ? IV[i] : ciphertext[i]);
    }
    firstRound = 0;
    aes_encrypt(input, ciphertext, key, size);
    /* always 16 bytes output because of the padding for CBC */
    fwrite(ciphertext, sizeof(unsigned char), 16, out);
}

```

As you can see, we don't need the output block this time, as the ciphertext is identical to the output. The CBC decryption is similar, except for the fact that we pay attention to the originalFileSize we recovered from the input file and in case we have less than 16 bytes left to decrypt, we write the remaining bytes instead of the 16 bytes we read from the file:

```

while ((read = fread(ciphertext, sizeof(unsigned char), 16, in)) > 0)
{
    aes_decrypt(ciphertext, output, key, size);
    for (i = 0; i < 16; i++)
    {
        plaintext[i] = ((firstRound) ? IV[i] : input[i]) ^ output[i];
    }
    firstRound = 0;
    if (originalFileSize < 16)
    {
        fwrite(plaintext, sizeof(unsigned char), originalFileSize, out);
    }
    else
    {
        fwrite(plaintext, sizeof(unsigned char), read, out);
        originalFileSize -= 16;
    }
    memcpy(input, ciphertext, 16*sizeof(unsigned char));
}

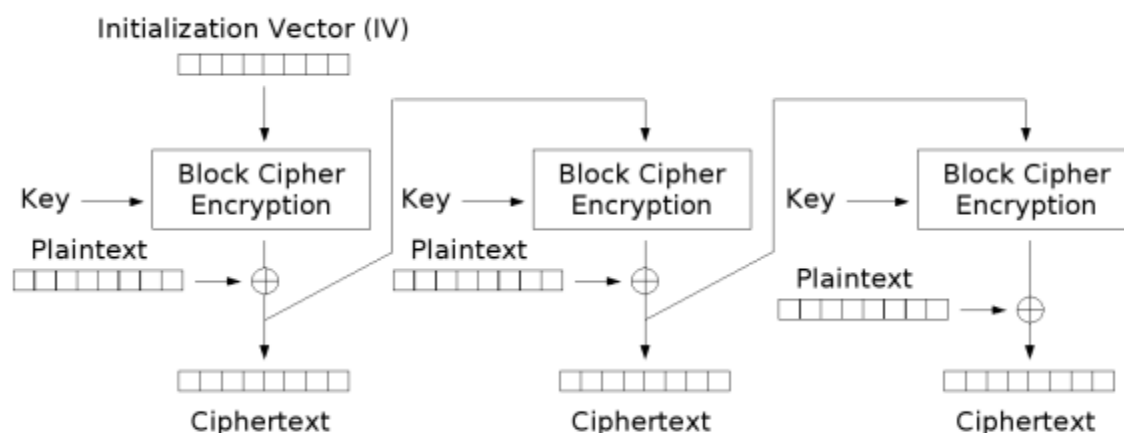
```

You probably noticed that I copied the content of the ciphertext into the input block. This is because ciphertext will be overwritten during the next iteration by the new ciphertext block, but we still need the ciphertext from the previous iteration to perform the XOR

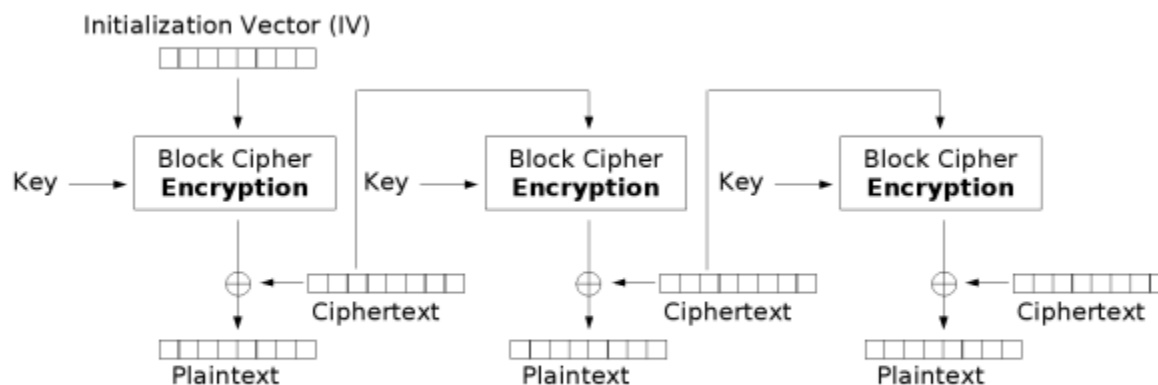
operation. In this case, I use input not as the input for the AES block cipher but for the XOR operation. Don't worry if you can't implement this code into a working example, I'll post the complete code at the end of this chapter.

## Cipher feedback (CFB)

The cipher feedback (CFB) mode, a close relative of CBC, makes a block cipher into a self-synchronizing stream cipher. Operation is very similar; in particular, CFB decryption is almost identical to CBC decryption performed in reverse.



Cipher Feedback (CFB) mode encryption



Cipher Feedback (CFB) mode decryption

CFB shares two advantages over CBC mode with the stream cipher modes OFB and CTR: the block cipher is only ever used in the encrypting direction, and the message does not need to be padded to a multiple of the cipher block size. I personally think this mode of fairly easy to implement, so without any hesitation I show you the final code for encryption:

```
while ((read = fread(plaintext, sizeof(unsigned char), 16, in)) > 0)
```

```
{
    if (firstRound)
    {
        aes_encrypt(IV, output, key, size);
        firstRound = 0;
    }
    else
    {
        aes_encrypt(input, output, key, size);
    }
    for (i = 0; i < 16; i++)
    {
        ciphertext[i] = plaintext[i] ^ output[i];
    }
    fwrite(ciphertext, sizeof(unsigned char), read, out);
    memcpy(input, ciphertext, 16*sizeof(unsigned char));
}
```

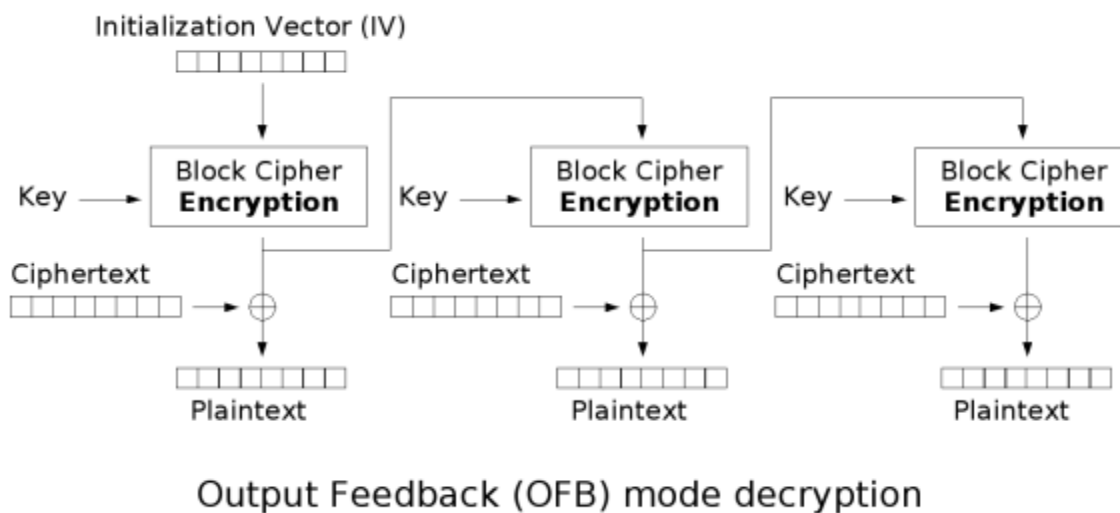
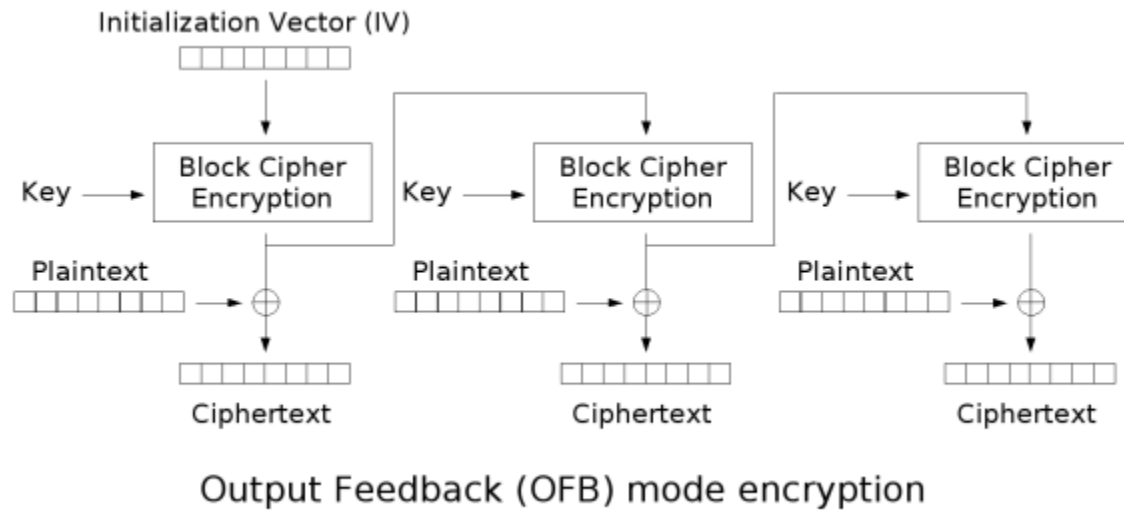
In this case, you'll probably notice that I could have used the ciphertext immediately as input for the AES encryption, something that I found illogical for naming convention, so I simply copied the content from the ciphertext into the input. In case you're looking for optimal performance, feel free to change this.

The decryption is very similar to the encryption (it even uses the AES **encryption**) and here's the code:

```
while ((read = fread(ciphertext, sizeof(unsigned char), 16, in)) > 0)
{
    if (firstRound)
    {
        aes_encrypt(IV, output, key, size);
        firstRound = 0;
    }
    else
    {
        aes_encrypt(input, output, key, size);
    }
    for (i = 0; i < 16; i++)
    {
        plaintext[i] = output[i] ^ ciphertext[i];
    }
    fwrite(plaintext, sizeof(unsigned char), read, out);
    memcpy(input, ciphertext, 16*sizeof(unsigned char));
}
```

## Output feedback (OFB)

If you read and understood the CFB mode, you shouldn't have any problems implementing the OFB mode. The only difference is that it uses the AES output before the XOR operation (and not like CFB after the XOR operation) as input for the next iteration.



Here's the code for the encryption:

```
while ((read = fread(plaintext, sizeof(unsigned char), 16, in)) > 0)
{
    if (firstRound)
    {
        aes_encrypt(IV, output, key, size);
        firstRound = 0;
    }
    else
    {
        aes_encrypt(input, output, key, size);
    }
    for (i = 0; i < 16; i++)
    {
        ciphertext[i] = plaintext[i] ^ output[i];
    }
    fwrite(ciphertext, sizeof(unsigned char), read, out);
    memcpy(input, output, 16*sizeof(unsigned char));
}
```

As you can see, the only difference to CFB mode is that I copy the content of output into input and not the content of ciphertext. Here's the code for the decryption:

```
while ((read = fread(ciphertext, sizeof(unsigned char), 16, in)) > 0)
{
    if (firstRound)
```

```

{
    aes_encrypt(IV, output, key, size);
    firstRound = 0;
}
else
{
    aes_encrypt(input, output, key, size);
}
for (i = 0; i < 16; i++)
{
    plaintext[i] = output[i] ^ ciphertext[i];
}
fwrite(plaintext, sizeof(unsigned char), read, out);
memcpy(input, output, 16*sizeof(unsigned char));
}

```

## Putting it all together

Now that we have explained and implementing our three modes of operation, all that is left is to put them all together into one piece of code.

### Encryption:

```

void encrypt(FILE *in, FILE *out, enum modeOfOperation mode)
{
    /* the non-expanded key size */
    enum keySize size = SIZE_32;

    /* the AES input/output */
    unsigned char plaintext[16] = {0};
    unsigned char input[16] = {0};
    unsigned char output[16] = {0};
    unsigned char ciphertext[16] = {0};
    unsigned char IV[16] = {0};

    /* the AES key */
    unsigned char key[32] = {0x0};

    /* char firstRound */
    char firstRound = 1;

    long int fileSize;
    size_t read;
    int i;

    if (in != NULL)
    {
        fseek(in, 0, SEEK_END);
        fileSize = ftell(in);
        fseek(in, 0, SEEK_SET);

        /* add the file header */
        fwrite(&mode, sizeof(mode), 1, out);
        fwrite(&fileSize, sizeof(fileSize), 1, out);

        while ((read = fread(plaintext, sizeof(unsigned char), 16, in)) > 0)
        {
            if (mode == CFB)
            {
                if (firstRound)
                {
                    aes_encrypt(IV, output, key, size);
                    firstRound = 0;
                }
                else
                {
                    aes_encrypt(input, output, key, size);
                }
                for (i = 0; i < 16; i++)
                {
                    ciphertext[i] = plaintext[i] ^ output[i];
                }
            }
        }
    }
}

```



```
while ((read = fread(ciphertext, sizeof(unsigned char), 16, in)) > 0)
{
    if (mode == CFB)
    {
        if (firstRound)
        {
            aes_encrypt(IV, output, key, size);
            firstRound = 0;
        }
        else
        {
            aes_encrypt(input, output, key, size);
        }
        for (i = 0; i < 16; i++)
        {
            plaintext[i] = output[i] ^ ciphertext[i];
        }
        fwrite(plaintext, sizeof(unsigned char), read, out);
        memcpy(input, ciphertext, 16*sizeof(unsigned char));
    }
    else if (mode == OFB)
    {
        if (firstRound)
        {
            aes_encrypt(IV, output, key, size);
            firstRound = 0;
        }
        else
        {
            aes_encrypt(input, output, key, size);
        }
        for (i = 0; i < 16; i++)
        {
            plaintext[i] = output[i] ^ ciphertext[i];
        }
        fwrite(plaintext, sizeof(unsigned char), read, out);
        memcpy(input, output, 16*sizeof(unsigned char));
    }
    else if (mode == CBC)
    {
        aes_decrypt(ciphertext, output, key, size);
        for (i = 0; i < 16; i++)
        {
            plaintext[i] = ((firstRound) ? IV[i] : input[i]) ^ output[i];
        }
        firstRound = 0;
        if (originalFileSize < 16)
        {
            fwrite(plaintext, sizeof(unsigned char), originalFileSize, out);
        }
        else
        {
            fwrite(plaintext, sizeof(unsigned char), read, out);
            originalFileSize -= 16;
        }
        memcpy(input, ciphertext, 16*sizeof(unsigned char));
    }
}
}
```