CSE 470 CRYPTOGRAPHY AND COMPUTER SECURITY

PROJECT

Merve Horuz

1801042651

ISAP: ISAP, or the Internet Secure Association Protocol, is a proprietary encryption algorithm developed by Check Point Software Technologies for use in their VPN products. It is a symmetric key algorithm, meaning that the same key is used for both encryption and decryption.

ISAP uses a combination of block cipher and stream cipher techniques to provide both confidentiality and integrity protection for data transmitted over the internet. It uses a block cipher, such as DES or 3DES, to encrypt large blocks of data, and a stream cipher, such as RC4, to encrypt smaller blocks of data.

Here is an example of how ISAP encryption might work:

The sender generates a random key to use for the current session.

The sender encrypts the data to be transmitted using the block cipher and the key.

The encrypted data is then divided into smaller blocks, and each block is encrypted using the stream cipher and the key.

The encrypted blocks are transmitted over the internet to the receiver.

The receiver decrypts the blocks using the stream cipher and the key.

The receiver then combines the decrypted blocks and decrypts the combined data using the block cipher and the key.

ISAP is no longer in widespread use and has been replaced by more modern and secure encryption algorithms.

Imagine that Alice wants to send a message to Bob, but she wants to ensure that the message is secure and can only be read by Bob. She decides to use a symmetric key algorithm to encrypt the message.

First, Alice and Bob agree on a secret key to use for the current session. Let's say they choose the key "secretkey123".

Alice writes the message she wants to send to Bob, which is: "Meet me at the park at noon."

Alice then uses the symmetric key algorithm and the secret key to encrypt the message. The encrypted message might look something like this: "kfjdslkfjwoieur098234098234098234098234"

Alice sends the encrypted message to Bob over the internet.

When Bob receives the message, he uses the same symmetric key algorithm and the secret key to decrypt the message. The decrypted message will be the original message that Alice sent: "Meet me at the park at noon."

```
Message: Meet me at the park at noon.
Encrypted message: kfjdslkfjwoieur098234098234098234098234
Decrypted message: Meet me at the park at noon.
```

GIFT-COFB:

We started with generating nonce:

```
def generate_nonce():
    while True:
        nonce = random_bits(128)
        if nonce not in list:
            list.append(nonce)
            break

    return nonce
```

A nonce, short for "number used once," is a random number that is generated and used in the encryption process. In the GIFT-COFB cipher, the nonce generator is responsible for generating a unique nonce for each block of data that is encrypted. The nonce is used as part of the key for the block cipher, along with other secret information, to ensure that each block of data is encrypted differently.

Here is an example of how the nonce generator might work in the GIFT-COFB cipher:

The sender generates a random nonce for the current block of data.

The sender combines the nonce with other secret information to create the key for the block cipher.

The sender uses the key to encrypt the current block of data using the block cipher.

The encrypted block of data, along with the nonce and other necessary information, is transmitted to the receiver.

The receiver uses the nonce and the other secret information to recreate the key and decrypt the block of data.

The use of a nonce in the GIFT-COFB cipher helps to ensure that the same message encrypted with the same key will produce a different encrypted message each time, which is important for the security of the cipher.

Encryption of message:

```
def encrypt_gift_cofb(n, a, m, k):
    a_temp = pad(a)

    associated_size = int(len(a_temp) / 128)
    message_size= 0

    A = [bitarray] * (associated_size)
```

n: This appears to be a nonce, which is a random number that is used as part of the key for the cipher.

a: This is the associated data, which is a block of data that is not encrypted but is included in the output of the encryption process.

m: This is the message to be encrypted, which is the data that the sender wants to keep confidential.

k: This is the secret key that is used for the encryption process.

The function first pads the associated data and message (if one is provided) so that they are both a multiple of 128 bits in length. It then divides the data into 128-bit blocks and stores them in two lists called A and M.

The function then initializes a number of other lists and variables that will be used in the encryption process. It begins by generating the first block of ciphertext, Y [0], using the GIFT_box function and the nonce and key. It also generates a value called L, which is used in the encryption process.

The function then enters a loop that iterates over the blocks of associated data. For each block, it calculates a new value of L using the helper function, and then uses this value to calculate the current block of ciphertext, Y[i], using the GIFT_box function and the key.

If the associated data is a multiple of 128 bits in length, the function generates two additional blocks of ciphertext using the helper function and the GIFT_box function.

If a message is provided, the function enters another loop that iterates over the blocks of the message. It calculates a new value of L for each block using the helper function, and then uses this value to calculate the current block of ciphertext, C[i], and the corresponding block of ciphertext for the associated data, Y[i + associated_size].

Finally, the function constructs the output of the encryption process, which consists of the encrypted message (c) and a block of ciphertext (t) derived from the final block of associated data ciphertext. It returns these values as a list.

```
def GIFT_box(plaintext, key):
    cipherstate = initialise_pt(plaintext)
    keystate = initialise_ks(key)

    for i in range(1, 41):
        cipherstate = subcells(cipherstate)
        cipherstate = permBits(cipherstate)
        AddRoundKey(cipherstate, keystate, i)
        keystateupdate(keystate)


    y = finalise(cipherstate)
    return y
```

The function begins by initializing the cipher state and the key state using the initialise_pt and initialise_ks functions, respectively. It then enters a loop that iterates 40 times.

On each iteration of the loop, the function performs a number of operations on the cipher state and key state:

It applies the subcells function to the cipher state. This function appears to perform a substitution operation on each of the 32-bit cells of the cipher state.

It applies the permBits function to the cipher state. This function appears to permute the bits within each cell of the cipher state.

It applies the AddRoundKey function to the cipher state and the key state, using the current iteration number as an argument. This function appears to perform a key addition operation on the cipher state using the key state.

It updates the key state using the keystateupdate function.

After the loop completes, the function applies the finalise function to the cipher state and returns the result as the encrypted data (y).

Decryption of message:

```
def decrypt_gift_cofb(n, a, c, t, k):
    a_temp = pad(a)

    associated_size = int(len(a_temp) / 128)
    c_size = 0

    A = [bitarray] * (associated_size)
```

The decrypt_gift_cofb function is designed to perform the decryption process of the GIFT-COFB encryption algorithm. It takes five input parameters:

n: the nonce, a bitarray of length 128 bits.

a: the associated data, a bitarray of any length.

c: the encrypted message, a bitarray of any length.

t: the authentication tag, a bitarray of length 128 bits.

k: the key, a bitarray of length 128 bits.

The function first pads the associated data and encrypted message to a length that is a multiple of 128 bits, if necessary, using the pad function. It then divides the padded associated data and encrypted message into blocks of 128 bits each and stores these blocks in the A and C lists, respectively.

The function then initializes three empty lists: X, Y, and M, which will be used to store intermediate values during the decryption process. It also initializes the first block of ciphertext, Y [0], using the GIFT_box function and the nonce and key. The first value of L, a 64-bit bitarray, is then initialized as the first 64 bits of Y [0].

Next, the function enters a loop that iterates over the blocks of associated data. On each iteration, it performs the following steps:

It updates L using the helper function with the value "two".

It computes the value of X[i] using the previous block of associated data, A[i-1], the intermediate value Y[i-1], and the updated value of L and a padding sequence.

It computes the value of Y[i] using the GIFT_box function and the value of X[i].

After the loop is completed, the function checks if the length of the associated data is a multiple of 128 bits. If it is, it updates L using the helper function with the value "three". If it is not, it updates L using the helper function with the value "three" twice.

Next, the function computes the value of X[associated_size] using the last block of associated data, A[associated_size-1], the intermediate value Y[associated_size-1], and the updated value of L and a padding sequence. It then computes the value of Y[associated_size] using the GIFT_box function and the value of X[associated_size].

If the encrypted message is not empty, the function enters a loop that iterates over the blocks of encrypted message. On each iteration, it performs the following steps:

It updates L using the helper function with the value "two".

It computes the value of M[i] using the previous block of encrypted message, C[i-1], and the intermediate value Y[i+associated_size-1].

It computes the value of X[i+associated_size] using the value of M[i], the intermediate value Y[i+associated_size-1], and the updated value of L and a padding sequence.

It computes the value of Y[i+associated_size] using the `GIFT_box

TEST

Zoom in:

I received the message as a bit string and processed the bits while encrypting. As you can see, the message and the decrypted message have the same bit string, so the message was decrypted with the correct key.

```
Message:  00001101101110101100101010111101110100011110111
10011110100111011101110101000010110011101001101100100111

Encryted message:  bitarray('10110011001010000001111011111
11011100000100110110110111011100001110000110010111110010

Decrypted message:  bitarray('00001101101110101100101010101
11000000001000010001100111101001110111011101010000101100

Is equal message and decrypted message:  True
```

With wrong associated data, it gives error:

```
#passing wrong associated data gives error proving that authenticity is ensured
aa = "10101110110000001011000"
c  = decrypt_gift_cofb(bitarray(n), bitarray(aa), t[0], t[1], bitarray(k))
```

```
Is equal message and decrypted message(with wrong associated data):  Error
zeroday@zeroday-Lenovo-V330-15IKB:~/Desktop/crypto/hw1/gift-cofb$
```

Gift-cofb encryption algorithm with CBC mode

The Cipher Block Chaining (CBC) mode of encryption is a block cipher mode of operation that allows the encryption of multiple blocks of data using a block cipher, such as AES. It works by XORing (exclusive-ORing) each block of plaintext with the previous ciphertext block before encrypting it. This means that each block of ciphertext is dependent on all the previous blocks of plaintext, which helps to provide confidentiality. In order to ensure the integrity of the data, an initialization vector (IV) is used to XOR the first block of plaintext before it is encrypted. The IV is typically a random number that is generated and sent along with encrypted data. The same key and IV must be used to decrypt the data, so they must be communicated securely between the sender and receiver.

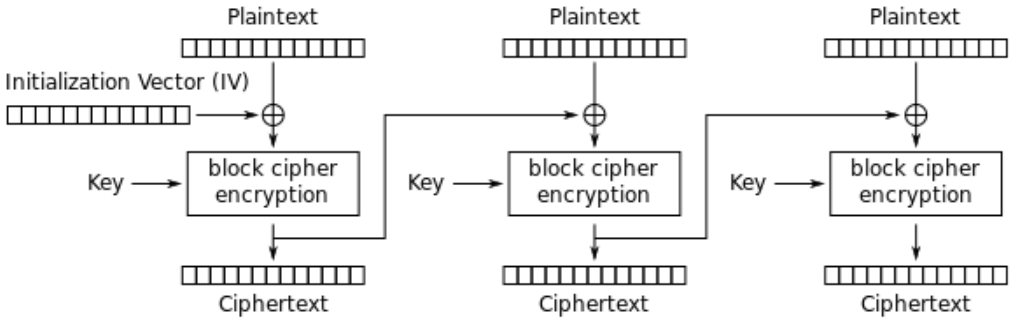
First with random_bits function initialization vector is generated (IV):

```
def random_bits(n):
    iv = ""
    for h in range (0, n):
        iv += str((random.randint(0,1)))

    return iv
```

Then, cbc_encryption function encrypts the message with cbc mode:

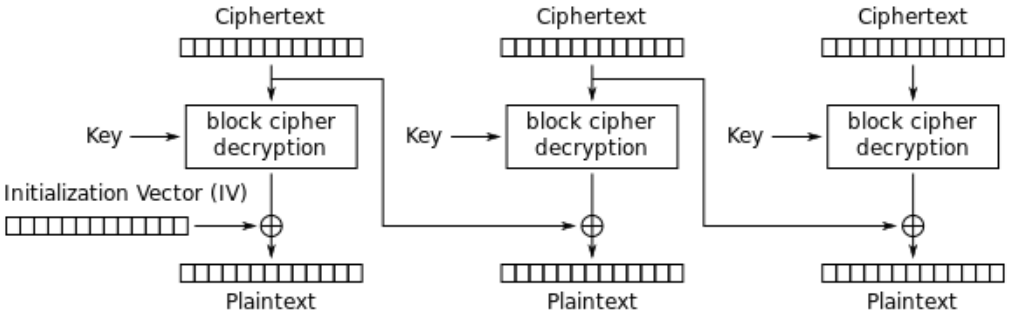
```
def cbc_encryption(m, n):
```



Cipher Block Chaining (CBC) mode encryption

Then, cbc_decryption decrypts the ciphertext with cbc mode:

```
def cbc_decryption(e, tag, n, iv):
```



Cipher Block Chaining (CBC) mode decryption

TEST

```
Message:  00001101101110101100101010111101110100011101110011101011111000000101000000101010111010100110011011100111011000011011100010011100001100000000100001000110011101001101110110111010100001011001110100110111001001111110100100000100000100001010110101000111111

Encrypted message with CBC mode 01101111001010000010110110100000000101111000110001000000110110000010110000111000000111011110110011011011010000111001010110110010001100000110100110110011000110000101111001010110100011011110000101001110101110000101000111011011

Decrypted message with CBC mode 00001101101110101100101010111101110100011101110011101011111000000101000000101010111010100110011011100111011000011011100010011100001100100010000100011001111010011101101110101000010110011101001101110010011111110100100000100000100001010110101000111111

Message equal to decrypted message: True
```

Zoom in

```
zeroday@zeroday-Lenovo-V330-15IKB:~/Desktop/crypto/hw1/gift-cc
Message:  00001101101110101100101010111101110100011110111001110
10011110100111011101110101000010110011101001101100100111111010

Encrypted message with CBC mode 01101111001010000010110110100
00001101001101100110001100001011110010101101000110111100001010

Decrypted message with CBC mode 00001101101110101100101010111
00110000000010000100011001111010011101110111010100001011001110

Message equal to decrypted message: True
```

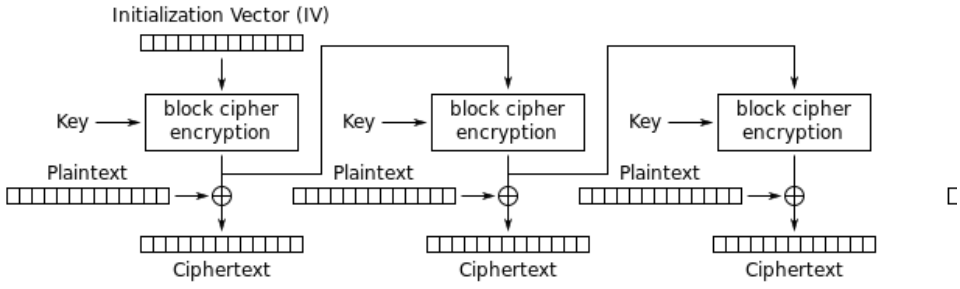
Gift-cofb encryption algorithm with OFB mode

The Output Feedback (OFB) mode of operation is a type of block cipher mode that converts a block cipher into a stream cipher. It is a confidentiality mode, meaning that it is used to protect the confidentiality of the message.

In OFB mode, a block of plaintext is encrypted using the block cipher to produce a block of ciphertext, which is then XORed with a block of plaintext to produce a block of output. This output block is then fed back into the block cipher to produce the next block of ciphertext, which is XORed with the next block of plaintext, and so on.

The ofb_encryption encrypts message with ofb mode:

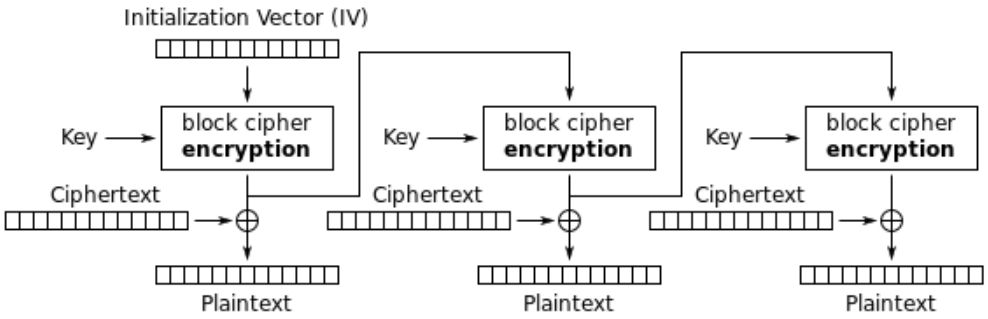
```
def ofb_encryption(m, n):
```



Output Feedback (OFB) mode encryption

The ofb_decryption decrypts message with ofb mode:

```
def ofb_decryption(e, n, iv2):
```



Output Feedback (OFB) mode decryption

CHECK FILE CHANGE

First, add_hash_value function adds the sign end of file.

```
def add_hash_value(file_name,key, n, a):
    if is_there_hash_value(file_name):
        removve_hash_value(file_name)

    document= open(file_name, "rb+")
    value = bitarray()
    value.fromfile(document)
    encrypted = encrypt_gift_cofb(bitarray(n), bitarray(a), value, bitarray(key))
    hash_value = encrypted[0][-128:]
    document.write(seperator.encode() + hash_value)
    document.close()
```

Before adding hash value,

```
CSE 470 cryptography
```

After adding hash value,

```
CSE 470 cryptography~..A....?.Zvh.'.I
```

Check_file_change function checks whether the file is changed or is not.

```
def check_file_change(file_name,key, n, a):
    if is_there_hash_value(file_name):
        document= open(file_name, "rb")
        document.seek(-16, io.SEEK_END)
        offset = document.tell() - 1
        hash_value = bitarray()
        hash_value.fromfile(document, 16)
        document.seek(0, io.SEEK_SET)
        value = bitarray()
        value.fromfile(document, offset)

        encrypted = encrypt_gift_cofb(bitarray(n), bitarray(a), value, bitarray(key))
        current_hash_value = encrypted[0][-128:]
        if current_hash_value != hash_value:
            print("hash values does not match..file has been changed.")
        else:
            print("hash values are same..file has not been changed")
        document.close()
    else:
        print("hash value is not in the file!")
```

Is_there_hash_value function checks that the file has or not the hash value.

```
def is_there_hash_value(file_name):
    size = os.stat(file_name).st_size
    doc = open(file_name, "rb")
    doc.seek(size - append_size, io.SEEK_SET)
    data = doc.read(1)
    if not (data.decode('utf-8') == seperator):
        return False
    else:
        return True
```

Removve_hash_value function removes the hash value from the file.

```
def removve_hash_value(file_name):
    if is_there_hash_value(file_name):
        file_size = os.stat(file_name).st_size
        document= open(file_name, "rb+")
        document.truncate(file_size - append_size)
    else:
        print("hash value is not in the file!")
    document.close()
```

TEST

You have to use hex editor editing the file!!!

Before the file is not changed,

```
hash values are same..file has not been changed
```

After the file is changed,

```
cryptography~..A....?.Zvh.'.I
```

```
hash values does not match..file has been changed.
```