# CSE 470 CRYPTOGRAPHY AND INFORMATION SECURITY

1801042609

YAKUP TALHA YOLCU

# TinyJAMBU:

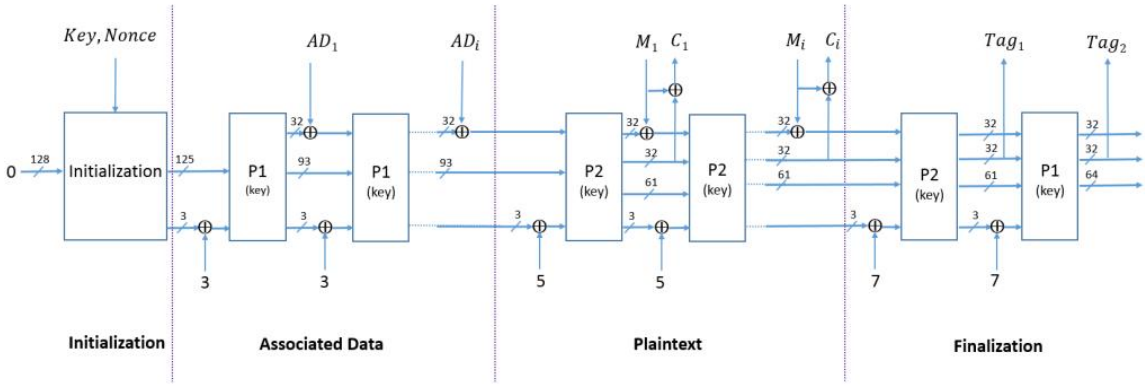
TinyJAMBU, designed by Wu and Huang , is an AEAD scheme that is inspired by the third-round candidate of the CAESAR competition, JAMBU. The main component of TinyJAMBU is a 128-bit keyed permutation without a key schedule that is based on a nonlinear feedback shift register, and the nonlinearity in each round is obtained using a single NAND operation.

| AEAD variants | *Key* | *Nonce* | *Tag* | *State size* |
|---|---|---|---|---|
| **TinyJAMBU-128** | 128 | 96 | 64 | 128 |
| TinyJAMBU-192 | 192 | 96 | 64 | 128 |
| TinyJAMBU-256 | 256 | 96 | 64 | 128 |

Security Analysis. In , the designers provided a security proof for the mode, analysis of the keyed-permutation, and security against forgery and key recovery attacks (including differential, linear, algebraic, and slide attacks). Saha et al. showed that the security margin of TinyJAMBU is around 12 % due to the dependencies between the outputs of multiple AND gates.

The TinyJAMBU mode is a small variant of the JAMBU mode which is a third-round candidate of the CAESAR competition. In the TinyJAMBU mode, a 128-bit keyed permutation is used, the state size is 128 its, and the message block size is 32 bits. When nonce is reused, the TinyJAMBU mode provides better authentication security than the JAMBU mode. When nonce is reused, the TinyJAMBU mode provides better authentication security than the Duplex mode (for the same permutation size and the same message block size). The reason is that the attacker can easily set part of the state to arbitrary value when nonce is reused in the Duplex mode, while it is difficult to do that in the TinyJAMBU mode. The TinyJAMBU mode is shown in Fig. 2.1. If the last block of the associated data (or plaintext) is not a full block , the length of the partial block (the number of bytes) is xored to the state.

The TinyJAMBU mode for 128-bit state state and keyed-permutations



## Operations, Variables and Functions

| | | |
|---|---|---|
| $\oplus$ | : | bit-wise exclusive OR |
| & | : | bit-wise AND |
| $\sim$ | : | bit-wise NOT |
| $\|\|$ | : | concatenation |
| $\lfloor a \rfloor$ | : | floor operator, gives the integer part of $a$ |
| $a_{\{i \cdots j\}}$ | : | the word consists of $a_i \|\| a_{i+1} \|\| \cdots \|\| a_j$, where $a_i$ is the $i$th bit of $a$. |
| $AD$ | : | associated data, a sequence of bytes. |
| $ad_i$ | : | one bit of associated data. |
| $adlen$ | : | the length of associated data in bits. |
| $C$ | : | ciphertext, a sequence of bytes |
| $c_i$ | : | the $i$th ciphertext bit. |
| $FrameBits$ | : | Three-bit FrameBits. |
| | | FrameBits = 1 for nonce |
| | | FrameBits = 3 for associated data |
| | | FrameBits = 5 for plaintext and ciphertext |
| | | FrameBits = 7 for finalization |
| $FrameBits_i$ | : | The $i$th bit of FrameBits. |
| $K$ | : | the key. |
| $k_i$ | : | the $i$th bit of $K$. |
| $klen$ | : | the key length in bits. |
| $M$ | : | the plaintext, a sequence of bytes |
| $m_i$ | : | the $i$th bit of the plaintext. |
| $mlen$ | : | the length of the plaintext in bits. |
| NONCE | : | the 96-bit nonce. |
| $\text{nonce}_i$ | : | the $i$th bit of the 96-bit nonce. |
| $P_n$ | : | the 128-bit permutation with $n$ rounds |
| $S$ | : | the 128-bit state of the permutation. |
| $s_i$ | : | the $i$th bit of the state of the permutation. |
| $T$ | : | the 64-bit authentication tag. |
| $t_i$ | : | the $i$th bit of the authentication tag. |

## The Keyed Permutation P

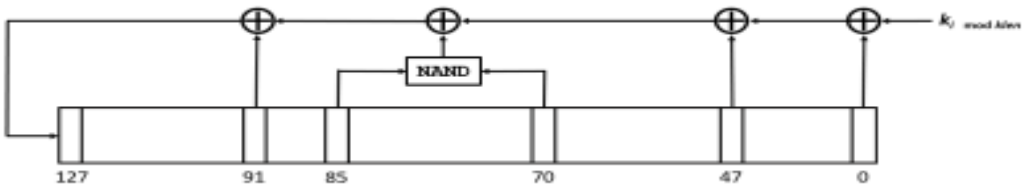
In TinyJAMBU, a 128-bit keyed permutation is used. The permutation Pn consists of n rounds. In the ith round of the permutation, a 128-bit nonlinear feedback shift register is used to update the state as follows

$$\begin{aligned}&\text{StateUpdate}(S, K, i):\\ &\quad \text{feedback} = s_0 \oplus s_{47} \oplus (\sim (s_{70} \& s_{85})) \oplus s_{91} \oplus k_{i \bmod klen}\\ &\quad \text{for } j \text{ from } 0 \text{ to } 126:\ s_j = s_{j+1}\\ &\quad s_{127} = \text{feedback}\\ &\text{end}\end{aligned}$$

**Code:**

```
void stateUpdate(int *state, const char *key, int i)
{
    int j;
    int temp1, temp2, temp3, temp4, feedback;
    for (j = 0; j < (i >> 5); j++)
    {
        temp1 = (state[1] >> 15) | (state[2] << 17);
        temp2 = (state[2] >> 6)  | (state[3] << 26);
        temp3 = (state[2] >> 21) | (state[3] << 11);
        temp4 = (state[2] >> 27) | (state[3] << 5);
        feedback = state[0] ^ temp1 ^ (~(temp2 & temp3)) ^ temp4 ^ ((int*)key)[j & 3];
        state[0] = state[1]; state[1] = state[2]; state[2] = state[3];
        state[3] = feedback ;
    }
}
```

For example, P384 means that the state of the permutation is updated using the function StateUpdate() for 384 times. 32 rounds of the permutation can be computed in parallel on 32-bit CPU




## The initialization

In the keyed permutation of TinyJAMBU-128, the 128-bit key of TinyJAMBU128 is used, and the klen is set to 128. The initialization of TinyJAMBU-128 consists of two stages: key setup and nonce setup.

Key Setup. The key setup is to randomize the state using the keyed permutation P1024.

1. Set the 128-bit state S as 0.
2. Update the state using P1024.

Nonce Setup. The nonce setup consists of three steps. In each step, the Framebits of nonce (the value is 1) are XORed with the state, then we update the state using the keyed permutation P384, then 32 bits of the nonce are XORed with the state.

for $i$ from 0 to 2:
    $s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
    Update the state using $P_{384}$
    $s_{\{96\cdots127\}} = s_{\{96\cdots127\}} \oplus nonce_{\{32i\cdots32i+31\}}$
end for

**Code**:

```
void initialization(const char *key, const char *iv, int *state)
{
        int i;

        for (i = 0; i < 4; i++) state[i] = 0;

        stateUpdate(state, key, 1024);

        for (i = 0;  i < 3; i++)
        {
            state[1] ^= FrameBitsIV;
            stateUpdate(state, key, 384);
            state[3] ^= ((int*)iv)[i];
        }
}
```

## Processing the associated data

After the initialization, we process the associated data AD. In each step, the Framebits of associated data (the value is 3) are XORed with the state, then we update the state using the keyed permutation P384, then 32 bits of the associated data are XORed with the state.

for $i$ from 0 to $\lfloor adlen/32 \rfloor$:
    $s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
    Update the state using $P_{384}$
    $s_{\{96\cdots127\}} = s_{\{96\cdots127\}} \oplus ad_{\{32i\cdots32i+31\}}$
end for

Processing the partial block of associated data. If the last block is not a full block (it is called a partial block), the last block is XORed to the state, and the number of bytes of associated data in the partial block is XORed to the state.

if $(adlen \bmod 32) > 0$:
  $s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
  Update the state using $P_{384}$
  $lenp = adlen \bmod 32$ /* number of bits in the partial block */
  $startp = adlen - lenp$ /* starting position of the partial block */
  $s_{\{96\cdots96+lenp-1\}} = s_{\{96\cdots96+lenp-1\}} \oplus ad_{\{startp\cdots adlen-1\}}$

  /* the number of bytes in the partial block is XORed to the state*/
  $s_{\{32\cdots33\}} = s_{\{32\cdots33\}} \oplus (lenp/8)$
end if

**Code:**

```
void processAssociatedData(const char *k, const char *ad, long long adlen, int *state)
{
    long long i;
    int j;

    for (i = 0; i < (adlen >> 2); i++)
    {
        state[1] ^= FrameBitsAD;
        stateUpdate(state, k, 384);
        state[3] ^= ((int*)ad)[i];
    }

    if ((adlen & 3) > 0)
    {
        state[1] ^= FrameBitsAD;
        stateUpdate(state, k, 384);
        for (j = 0; j < (adlen & 3); j++)  ((char*)state)[12 + j] ^= ad[(i << 2) + j];
        state[1] ^= adlen & 3;
    }
}
```

## The encryption

After processing the associated data, we encrypt the plaintext M. In each step, the Framebits of plaintext (the value is 5) are XORed with the state, then we update the state using the keyed permutation P1024, then 32 bits of the plaintext are XORed with the state, and we obtain 32 bits of ciphertext by XORing the plaintext with another part of the state.

**Processing the full blocks of plaintext:**

for $i$ from 0 to $\lfloor mlen/32 \rfloor$:
$s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
Update the state using $P_{1024}$
$s_{\{96\cdots127\}} = s_{\{96\cdots127\}} \oplus m_{\{32i\cdots32i+31\}}$
$c_{\{32i\cdots32i+31\}} = s_{\{64\cdots95\}} \oplus m_{\{32i\cdots32i+31\}}$
end for

**Processing the partial block of plaintext.**

If the last block is not a full block (it is a partial block), the last block is XORed to the state, and the number of bytes in the partial block is XORed to the state

if $(mlen \bmod 32) > 0$:
$s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
Update the state using $P_{1024}$
$lenp = mlen \bmod 32$ /* number of bits in partial block */
$startp = mlen - lenp$ /* starting position of partial block */
$s_{\{96\cdots96+lenp-1\}} = s_{\{96\cdots96+lenp-1\}} \oplus m_{\{startp\cdots mlen-1\}}$
$c_{\{startp\cdots mlen-1\}} = s_{\{64\cdots64+lenp-1\}} \oplus m_{\{startp\cdots mlen-1\}}$
/* the length (bytes) of the last partial block is XORed to the state*/
$s_{\{32\cdots33\}} = s_{\{32\cdots33\}} \oplus (lenp/8)$
end if

**The finalization**

After encrypting the plaintext, we generate the 64-bit authentication tag T as follows. The Framebits of finalization (the value is 7) are XORed with the state.

$s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
Update the state using $P_{1024}$
$t_{\{0\cdots31\}} = s_{\{64\cdots95\}}$

$s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
Update the state using $P_{384}$
$t_{\{32\cdots63\}} = s_{\{64\cdots95\}}$

**Code (Processing the full blocks of plaintext, Processing the partial block of plaintext, The finalization):**

```
int encryption(char *c,long int *clen, const char *m, long long mlen, const char *ad, long long adlen, const char *npub, const char *k)
{
    long long i;
    int j;
    char mac[8];
    int state[4];

    initialization(k, npub, state);

    processAssociatedData(k, ad, adlen, state);

    for (i = 0; i < (mlen >> 2); i++)
    {
        state[1] ^= FrameBitsPC;
        stateUpdate(state, k, 1024);
        state[3] ^= ((int*)m)[i];
        ((int*)c)[i] = state[2] ^ ((int*)m)[i];
    }
    if ((mlen & 3) > 0)
    {
        state[1] ^= FrameBitsPC;
        stateUpdate(state, k, 1024);
        for (j = 0; j < (mlen & 3); j++)
        {
            ((char*)state)[12 + j] ^= m[(i << 2) + j];
            c[(i << 2) + j] = ((char*)state)[8 + j] ^ m[(i << 2) + j];
        }
        state[1] ^= mlen & 3;
    }

    state[1] ^= FrameBitsFinalization;
    stateUpdate(state, k, 1024);
    ((int*)mac)[0] = state[2];

    state[1] ^= FrameBitsFinalization;
    stateUpdate(state, k, 384);
    ((int*)mac)[1] = state[2];

    *clen = mlen + 8;
    memcpy(c + mlen, mac, 8);

    return 0;
}
```

## The decryption

In a decryption process, the initialization and processing the associate data are the same as the encryption process. After processing the associated data, we decrypt the ciphertext C. In each step, the Framebits of plaintext (the value is 5) are XORed with the state, then we update the state using the keyed permutation P1024. We obtain 32 bits of plaintext by XORing the ciphertext with 32 state bits s{64···95}, then the plaintext is XORed with the state bits s{96···127}.

Processing the full blocks of ciphertext:

for $i$ from 0 to $\lfloor mlen/32 \rfloor$:
$s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
Update the state using $P_{1024}$
$m_{\{32i\cdots32i+31\}} = s_{\{64\cdots95\}} \oplus c_{\{32i\cdots32i+31\}}$
$s_{\{96\cdots127\}} = s_{\{96\cdots127\}} \oplus m_{\{32i\cdots32i+31\}}$
end for

Processing the partial block of ciphertext. If the last block is not a full block (it is a partial block), the number of bytes in the partial block is XORed to the state.

if $(mlen \bmod 32) > 0$:
$s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}}$
Update the state using $P_{1024}$
$lenp = mlen \bmod 32$ /* number of bits in partial block */
$startp = mlen - lenp$ /* starting position of partial block */
$m_{\{startp\cdots mlen-1\}} = s_{\{64\cdots64+lenp-1\}} \oplus c_{\{startp\cdots mlen-1\}}$
$s_{\{96\cdots96+lenp-1\}} = s_{\{96\cdots96+lenp-1\}} \oplus m_{\{startp\cdots mlen-1\}}$
/* the length (bytes) of the last partial block is XORed to the state*/
$s_{\{32\cdots33\}} = s_{\{32\cdots33\}} \oplus (lenp/8)$
end if

**Code:**

```
int decryption(char *m, long long *mlen, const char *c, long long clen, const char *ad, long long adlen, const char *npub, const char *
{
    long long i;
    int j, check = 0;
    char mac[8];
    int state[4];

    *mlen = clen - 8;

    initialization(k, npub, state);

    processAssociatedData(k, ad, adlen, state);

    for (i = 0; i < (*mlen >> 2); i++)
    {
        state[1] ^= FrameBitsPC;
        stateUpdate(state, k, 1024);
        ((int*)m)[i] = state[2] ^ ((int*)c)[i];
        state[3] ^= ((int*)m)[i];
    }
    if ((*mlen & 3) > 0)
    {
        state[1] ^= FrameBitsPC;
        stateUpdate(state, k, 1024);
        for (j = 0; j < (*mlen & 3); j++)
        {
            m[(i << 2) + j] = c[(i << 2) + j] ^ ((char*)state)[8 + j];
            ((char*)state)[12 + j] ^= m[(i << 2) + j];
        }
        state[1] ^= *mlen & 3;
    }

    state[1] ^= FrameBitsFinalization;
    stateUpdate(state, k, 1024);
    ((int*)mac)[0] = state[2];

    state[1] ^= FrameBitsFinalization;
    stateUpdate(state, k, 384);
    ((int*)mac)[1] = state[2];

    for (j = 0; j < 8; j++) { check |= (mac[j] ^ c[clen - 8 + j]); }
    if (check == 0) return 0;
    else return -1;
}
```

## The verification

After decrypting the plaintext, we generate a 64-bit authentication tag T 0 , then compare T 0 with the received tag T. The Framebits of finalization are of value 7

$$
\begin{aligned}
&s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}} \\
&\text{Update the state using } P_{1024} \\
&t'_{\{0\cdots31\}} = s_{\{64\cdots95\}} \\
&s_{\{36\cdots38\}} = s_{\{36\cdots38\}} \oplus FrameBits_{\{0\cdots2\}} \\
&\text{Update the state using } P_{384} \\
&t'_{\{32\cdots63\}} = s_{\{64\cdots95\}}
\end{aligned}
$$

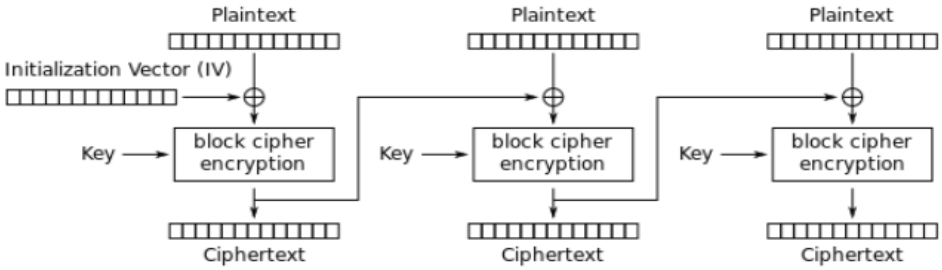
$T' = t'_{\{0\cdots63\}}$. Accept the message if $T' = T$; otherwise, reject.

**Code:**

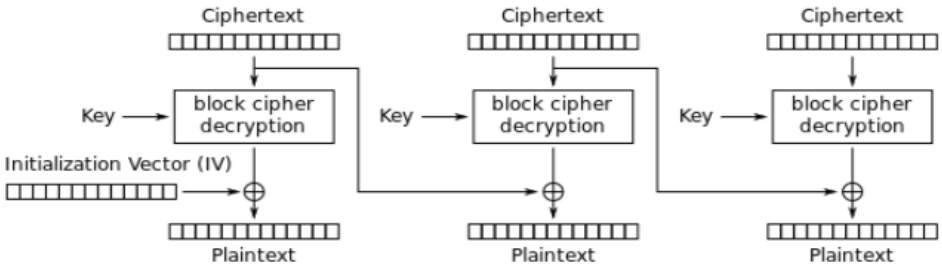
```
state[1] ^= FrameBitsFinalization;
stateUpdate(state, k, 1024);
((int*)mac)[0] = state[2];

state[1] ^= FrameBitsFinalization;
stateUpdate(state, k, 384);
((int*)mac)[1] = state[2];

for (j = 0; j < 8; j++) { check |= (mac[j] ^ c[clen - 8 + j]); }
if (check == 0) return 0;
else return -1;
```

## CBC (Cipher Block Chaining Mode)

Cipher block chaining (CBC) is a mode of operation for a block cipher -- one in which a sequence of bits are encrypted as a single unit, or block, with a cipher key applied to the entire block. Cipher block chaining uses what is known as an initialization vector (IV) of a certain length. By using this along with a single encryption key, organizations and individuals can safely encrypt and decrypt large amounts of plaintext.




Cipher Block Chaining (CBC) mode encryption




Cipher Block Chaining (CBC) mode decryption

CODE :

```
void CBCmode(const char* plain_text, const char* key, const char* IV, long
block_n)
{
    char plaintext[strlen(plain_text)+1];
    strcpy(plaintext, plain_text);
    int BLOCK_SIZE = strlen(plaintext)/block_n;
    long clens[block_n];
    char* associative_data = "assoc_data";
    long assoc_data_len  = sizeof(char)*strlen(associative_data);
    char* npub = "123456789012";

    int keyLen = (int)strlen(key);
    byte keyBytes[keyLen];
    get_byte_array(key, keyBytes);

    const int sourceLen = (int) strlen(plaintext);

    byte plaintextBytes[sourceLen];
    get_byte_array(plaintext, plaintextBytes);

    const int blockCount = sourceLen / BLOCK_SIZE + 1;
    byte byteBlocks[blockCount][BLOCK_SIZE*10];

    for (int i = 0; i < blockCount; ++i) {
        for (int j = 0; j < BLOCK_SIZE*10; ++j) {
            byteBlocks[i][j] = '\0';
        }
    }

    int bytePos = 0;
    for (int i = 0; i < blockCount; ++i) {
        for (int j = 0; j < BLOCK_SIZE; ++j) {
            byteBlocks[i][j] = plaintextBytes[bytePos++];
        }
    }

    int padding = bytePos - sourceLen;
    for (int i = BLOCK_SIZE - padding; i < BLOCK_SIZE*10; ++i) {
        byteBlocks[blockCount - 1][i] = '\0';
    }

    printf("\nCBC MODE :\n");
    printf("Text plaintext : %s\n", plaintext);

    for (int i = 0; i < blockCount; ++i) {
        byte tempStore[BLOCK_SIZE];
```

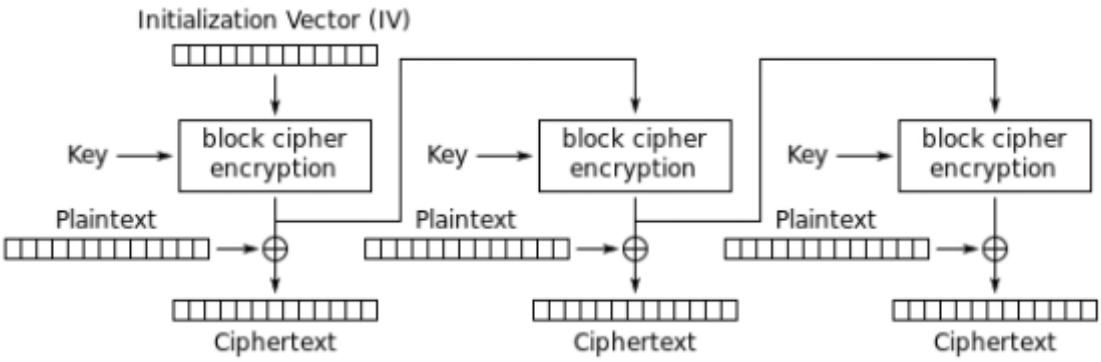
```
        if (i == 0) {
            xor_byte_arrays(IV, byteBlocks[i], tempStore, BLOCK_SIZE);
        } else {
            xor_byte_arrays(byteBlocks[i - 1], byteBlocks[i], tempStore,
BLOCK_SIZE);
        }

        encryption(
                byteBlocks[i], &clens[i],
                tempStore, BLOCK_SIZE,
                associative_data,
                assoc_data_len,
                npub,
                key
            );
    }

    printf("Text Encrypted CBC mode :");
    for (int i = 0; i < blockCount; ++i) {
        for (int j = 0; j < BLOCK_SIZE; ++j) {
            printf("%c", byteBlocks[i][j]);
        }
    }
    printf("\n");

    byte cipherStore[BLOCK_SIZE*10];
    for(int i =0; i<BLOCK_SIZE*10; ++i ) cipherStore[i] = '\0';
    bytecpy(cipherStore, byteBlocks[0], clens[0]);

    for (int i = 0; i < blockCount; ++i) {
        byte tempStore[BLOCK_SIZE], plainStore[BLOCK_SIZE];

        long long m_len;

        decryption(
            tempStore, &m_len,
            byteBlocks[i], clens[i],
            associative_data,assoc_data_len,
            npub,
            key
        );
        if (i == 0) {
            xor_byte_arrays(IV, tempStore, plainStore, BLOCK_SIZE);
        } else {
            xor_byte_arrays(cipherStore, tempStore, plainStore, BLOCK_SIZE);
        }
        plainStore[strlen(plainStore)] = '\0';
        bytecpy(cipherStore, byteBlocks[i], clens[i]);
        bytecpy(byteBlocks[i], plainStore, BLOCK_SIZE);
```

```
    }

    printf("Text Decrypted CBC mode :");
    for (int i = 0; i < blockCount; ++i) {
        if( i == blockCount - 1){
            printf("%s", byteBlocks[i]);
        }else{
        for (int j = 0; j < BLOCK_SIZE; ++j) {
            printf("%c", byteBlocks[i][j]);
        }}
    }
    printf("\n");
}
```

## OFB Output Feedback Mode

The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error-correcting codes to function normally even when applied before encryption.

**OFB** is an AES block cipher mode similar to the CFB mode. What mainly differs from CFB is that the OFB mode relies on XOR-ing plaintext and ciphertext blocks with expanded versions of the initialization vector.




Output Feedback (OFB) mode encryption

Code :

```
void myOFBmode(const char* plain_text, const char* key, const char* IV, long
block_n) {
    printf("\nOFB MODE : \n");
    printf("Text plaintext : %s\n",plain_text);

    int new_plain_text_len=strlen(plain_text)+strlen(plain_text)%block_n;
    char* new_plain_text=(char*)malloc(sizeof(char)*new_plain_text_len);


    for(int i=0;i<new_plain_text_len;i++) {
        new_plain_text[i]='\0';
    }
    for(int i=0;i<strlen(plain_text);i++) {
        new_plain_text[i]=plain_text[i];
    }


    int block_size = ((new_plain_text_len)/block_n);
    byte text_blocks[block_n][block_size];
    for( int i =0; i<block_n; ++i ){
        for(int j = 0; j<block_size; ++j )
            text_blocks[i][j] = (byte)'\0';
    }
    for( int i =0; i<block_n; ++i ){
        for(int j = 0; j<block_size; ++j )
            text_blocks[i][j] = (byte)new_plain_text[i*block_size + j];
    }


    long encryp_result_lengths[block_n];
    byte encryp_results[block_n][block_size*10];
    byte cipher_text[block_n][block_size*10];

    for( int i =0; i<block_n; ++i ){
        for(int j = 0; j<block_size*10; ++j ){
            encryp_results[i][j] = (byte)'\0';
            cipher_text[i][j] = (byte)'\0';
        }
    }

    char* associative_data = "assoc_data";
    long assoc_data_len  = sizeof(char)*strlen(associative_data);
    char* npub = "123456789012";


    //first block
    encryption(encryp_results[0],&encryp_result_lengths[0],
        IV,strlen(IV)*sizeof(byte),
```

```
        associative_data,assoc_data_len,npub,key
    );

    for(int j=0;j<block_size;j++) {
        cipher_text[0][j]=text_blocks[0][j] ^ encryp_results[0][j];
    }

    for(int i=1;i<block_n;i++) {
        encryption(encryp_results[i],&encryp_result_lengths[i],
            encryp_results[i-1],encryp_result_lengths[i-1],
            associative_data,assoc_data_len,npub,key
        );

        for(int j=0;j<block_size;j++) {
            cipher_text[i][j]=text_blocks[i][j] ^ encryp_results[i][j];
        }
    }

    printf("Text Encrypted OFB mode : ");
    for(int i =0; i<block_n; ++i ){
        for(int j=0;j<block_size;j++) {
            printf("%c", cipher_text[i][j]);
        }
    }
    printf("\n");


    byte plain_text_deciphered[block_n][block_size];

    for(int i=0;i<block_n;i++) {
        for(int j=0;j<block_size;j++) {
            plain_text_deciphered[i][j]=(byte)'\0';
        }
    }


    long decryp_result_lengths[block_n];
    byte decryp_results[block_n][block_size*10];
    byte decipher_text[block_n][block_size*10];

    for( int i =0; i<block_n; ++i ){
        for(int j = 0; j<block_size*10; ++j ){
            decryp_results[i][j] = (byte)'\0';
            decipher_text[i][j] = (byte)'\0';
        }
    }


    //first block
```

```
    encryption(decryp_results[0],&decryp_result_lengths[0],
    IV,strlen(IV)*sizeof(byte),
    associative_data,assoc_data_len,npub,key
    );

    for(int j=0;j<block_size;j++) {
        decipher_text[0][j]=cipher_text[0][j] ^ decryp_results[0][j];
    }

    for(int i=1;i<block_n;i++) {
        encryption(decryp_results[i],&decryp_result_lengths[i],
        decryp_results[i-1],decryp_result_lengths[i-1],
        associative_data,assoc_data_len,npub,key
        );

        for(int j=0;j<block_size;j++) {
            decipher_text[i][j]=cipher_text[i][j] ^ decryp_results[i][j];
        }
    }

    printf("Text Decrypted OFB mode : ");
    for(int i =0; i<block_n; ++i ){
        for(int j=0;j<block_size;j++) {
            printf("%c", decipher_text[i][j]);
        }
    }
    printf("\n");

    free(new_plain_text);

}
```

# Romulus

Romulus is an authenticated encryption with associated data (AEAD) technique based on a tweakable block cipher (TBC) Skinny, as described in this introduction. Romulus is divided into two families:

Romulus-N, a nonce-based AE (NAE), and Romulus-M, a nonce-misuse-resistant AE (MRAE).

TBCs have been recognized as a useful primitive because they may be used to build simple and secure NAE/MRAE systems, such as ΘCB3 and SCT. While these techniques are computationally efficient (in terms of the amount of primitive calls) and secure, lightweight applications are not their primary use cases, and they are not well-suited to small devices. With this in mind, Romulus aspires towards TBCbased NAE and MRAE schemes that are lightweight, efficient, and highly secure.

Although Romulus-overall N's structure is comparable to a (TBC-based variant of) block cipher mode COFB, we make significant changes to reach our design aim. Because of the faster MAC computation for associated data, Romulus-N requires fewer TBC calls than ΘCB3, and the hardware implementation is much smaller than ΘCB3 due to the lower state size and inverse-freeness (i.e., TBC inverse is not needed). In reality, the state size of Romulus-N is exactly what is required for computing TBC. It also encrypts an n-bit plaintext block with only one call to the n-bit block TBC, resulting in no efficiency loss. For small messages, Romulus-N is extremely efficient, which is especially important in many lightweight applications, requiring only two TBC calls to handle one associated data block and one message block (in comparison, other designs such as ΘCB3 OCB3, TAE, and CCM require from three to five TBC calls in the same situation). Romulus-N achieves these benefits without sacrificing security, i.e., Romulus-N has complete n-bit security, which is comparable to ΘCB3.

The state size of Romulus-N is comparable to other size-oriented and n-bit secure AE systems, such as typical permutation-based AEs using 3n-bit permutation with n-bit rate (3n to 3.5n bits). Because of the decreased output size, the underlying cryptographic primitive is predicted to be substantially more lightweight and/or faster (3n vs n bits). Furthermore, the n-bit security of Romulus-N is demonstrated using the standard model, which guarantees security not only mathematically but qualitatively. To elaborate, a security proof in the standard model allows one to precisely tie the primitive's security status to the overall security of the mode that employs it. In our situation, the best attack on each member of Romulus is a chosen-plaintext attack (CPA) in the single-key configuration against Skinny, i.e., Romulus retains the stated n-bit security unless Skinny is cracked by CPA adversaries in the singlekey setting. With non-standard models, such a guarantee is impossible, and it's sometimes difficult to determine the influence of a discovered "flaw" in the primitive on the mode's security. In a broader sense, "uninstantiable" Random Oracle-Model schemes best exemplify the gap between the proof and the actual security.

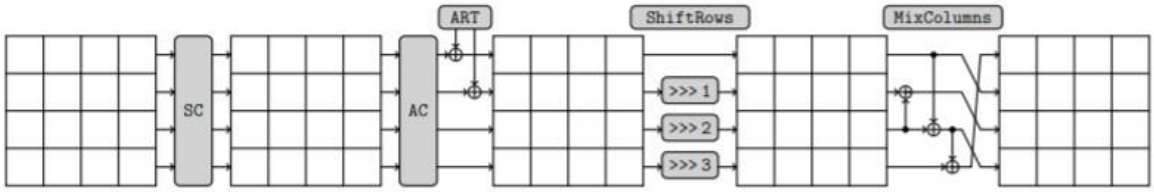
## Parameters:

Romulus has the following parameters:

- Nonce length nl ∈ {96, 128}.
- Key length k = 128.
- Message block length n = 128.
- Counter bit length d ∈ {24, 56, 48}.
- AD block length n + t, where t ∈ {96, 128}.
- Tag length τ = 128.
- A TBC E' : K × T' × M → M, where K = {0, 1}^k , M = {0, 1}^n , and T' = T × B × D. Here, T = {0, 1}^t , D = [[2^d − 1]]0 , and B = [[256]]0 for parameters t and d, and B is also represented as a byte. For tweak T' = (T, B, D) ∈ T' , T is always assumed to be a byte string including ε, and t is a multiple of 8. E' is either Skinny-128-384 or Skinny-128-256 with appropriate tweakey encoding functions.

## The Round Function:

Skinny-128-256 and Skinny-128-384 have 48 and 56 rounds, respectively. SubCells(SC), AddConstants(AC), AddRoundTweakey(ART), ShiftRows(SR), and MixColumns(MC) are the five operations that make up an encryption round.




**SubCells(SC):**

Every cell of the cipher internal state receives an 8-bit Sbox.

**AddConstants(AC):**

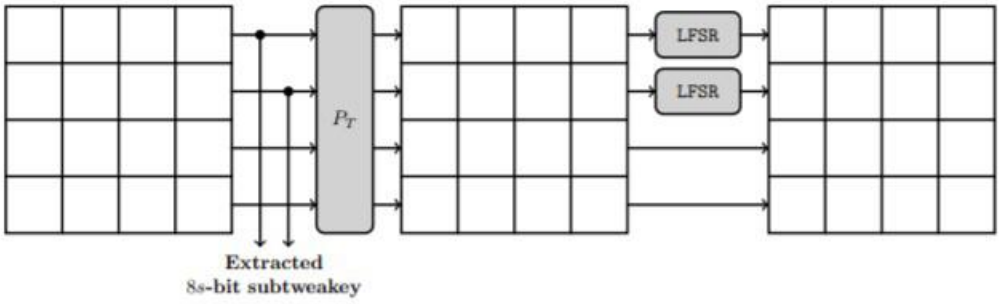
To generate round constants, a 6-bit affine LFSR with the state (rc5, rc4, rc3, rc2, rc1, rc0) (with rc0 being the least significant bit) is utilized. It has the following update function:
(rc5||rc4||rc3||rc2||rc1||rc0) → (rc4||rc3||rc2||rc1||rc0||rc5⊕rc4⊕1).
The six bits are set to zero and then updated before being used in a round. Depending on the size of the internal state, the bits from the LFSR are ordered into a 4 x 4 array (only the first column of the state is influenced by the LFSR bits).

**AddRoundTweakey(ART):**

All tweakey arrays' first and second rows are taken and bitwise exclusive-ored to the cipher internal state while maintaining array placement. For i = {0, 1} and j = {0, 1, 2, 3}, the formal expression is:

- ISi,j = ISi,j ⊕ TK1i,j ⊕ TK2i,j when z = 2,
- ISi,j = ISi,j ⊕ TK1i,j ⊕ TK2i,j ⊕ TK3i,j when z = 3.

The tweakey schedule in Skinny. Each tweakey word TK1, TK2 and TK3 (if any) follows a similar transformation update, except that no LFSR is applied to TK1.




**ShiftRows(SR):**
The rows of the cipher state cell array in this layer are rotated to the right, much like in AES. The second, third, and fourth cell rows are rotated to the right by 1, 2, and 3 positions, respectively. In other
words, the cells positions of the cipher internal state cell array are permuted P: for every $0 \leq i \leq 15$, we
set ISi ← ISP[i] with,
P = [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12].

**MixColumns(MC):**
The following binary matrix M is multiplied by each column of the cipher internal state array:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

**Algorithm** Romulus-N.Enc$_K(N, A, M)$

1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** $n$
4. **if** $|A[a]| < u$ **then** $w_A \leftarrow 26$ **else** 24
5. $A[a] \leftarrow \mathtt{pad}_u(A[a])$
6. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
7. $\quad (S, \eta) \leftarrow \rho(S, A[2i-1])$
8. $\quad S \leftarrow \widetilde{E}_K^{(A[2i], S, \overline{2i-1})}(S)$
9. **end for**
10. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
11. $(S, \eta) \leftarrow \rho(S, V)$
12. $S \leftarrow \widetilde{E}_K^{(N, w_A, \overline{3})}(S)$
13. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
14. **if** $|M[m]| < n$ **then** $w_M \leftarrow 21$ **else** 20
15. **for** $i = 1$ **to** $m - 1$
16. $\quad (S, C[i]) \leftarrow \rho(S, M[i])$
17. $\quad S \leftarrow \widetilde{E}_K^{(N, 4, \overline{i})}(S)$
18. **end for**
19. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
20. $(S, C'[m]) \leftarrow \rho(S, M'[m])$
21. $C[m] \leftarrow \mathtt{lsb}_{|M[m]|}(C'[m])$
22. $S \leftarrow \widetilde{E}_K^{(N, w_M, \overline{m})}(S)$
23. $(\eta, T) \leftarrow \rho(S, 0^n)$
24. $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m-1] \,\|\, C[m]$
25. **return** $(C, T)$

**Algorithm** Romulus-N.Dec$_K(N, A, C, T)$

1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** $n$
4. **if** $|A[a]| < u$ **then** $w_A \leftarrow 26$ **else** 24
5. $A[a] \leftarrow \mathtt{pad}_u(A[a])$
6. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
7. $\quad (S, \eta) \leftarrow \rho(S, A[2i-1])$
8. $\quad S \leftarrow \widetilde{E}_K^{(A[2i], S, \overline{2i-1})}(S)$
9. **end for**
10. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
11. $(S, \eta) \leftarrow \rho(S, V)$
12. $S \leftarrow \widetilde{E}_K^{(N, w_A, \overline{3})}(S)$
13. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
14. **if** $|C[m]| < n$ **then** $w_C \leftarrow 21$ **else** 20
15. **for** $i = 1$ **to** $m - 1$
16. $\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
17. $\quad S \leftarrow \widetilde{E}_K^{(N, 4, \overline{i})}(S)$
18. **end for**
19. $\widetilde{S} \leftarrow (0^{|C[m]|} \,\|\, \mathtt{msb}_{n-|C[m]|}(G(S)))$
20. $C'[m] \leftarrow \mathtt{pad}_n(C[m]) \oplus \widetilde{S}$
21. $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$
22. $M[m] \leftarrow \mathtt{lsb}_{|C[m]|}(M'[m])$
23. $S \leftarrow \widetilde{E}_K^{(N, w_C, \overline{m})}(S)$
24. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
25. $M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m-1] \,\|\, M[m]$
26. **if** $T^* = T$ **then return** $M$ **else** $\perp$

**Algorithm** $\rho(S, M)$

1. $C \leftarrow M \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** $(S', C)$

**Algorithm** $\rho^{-1}(S, C)$

1. $M \leftarrow C \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** $(S', M)$

## Conclusion:

The NIST lightweight cryptography competition has reached the final round, with a shortlist of ten contenders. Each differs in their approach, but they aim to create a cryptography method that is secure,
has a low footprint, and is robust against attacks. So while many of the contenders, such as ASCON, GIFT and Isap, use the sponge method derived from the SHA-3 standard (Keccak), Romulus takes a more traditional approach and looks towards a more traditional light-weight crypto approach. Overall it
is defined as a tweakable block cipher (TBC) and which supports authenticated encryption with associated data (AEAD). For its more traditional approach, it uses the Skinny lightweight tweakable block cipher.

CBC Cipher Block Chaining Mode

```
void CBCMode(unsigned char*c,unsigned long long* clen,
       const unsigned char* m,const unsigned char* ad,
             unsigned char* nsec,
       const unsigned char* npub,
       const unsigned char* k) {

    //c -> ciphertext
    //clen -> ciphertext len
    //m -> plaintext
    //mlen-> plaintext len
    //ad -> associative data
    //nsec - >
    //npub -> IV
    //k -> key

    int block_n=5;
  int new_plain_text_len=strlen(m);
  char* new_plain_text=(char*)malloc(sizeof(char)*new_plain_text_len);

  for(int i=0;i<new_plain_text_len;i++) {
    new_plain_text[i]='\0';
  }
  for(int i=0;i<strlen(m);i++) {
    new_plain_text[i]=m[i];
  }

  int block_size = ((new_plain_text_len)/block_n);
  byte text_blocks[block_n][block_size];
  for( int i =0; i<block_n; ++i ){
    for(int j = 0; j<block_size; ++j )
      text_blocks[i][j] = (byte)'\0';
  }
  for( int i =0; i<block_n; ++i ){
    for(int j = 0; j<block_size; ++j )
      text_blocks[i][j] = (byte)new_plain_text[i*block_size + j];
  }

    byte xor_results[block_n][block_size*10];
  byte cipher_text[block_n][block_size*10];
    unsigned long long cipher_text_lens[block_n];
  byte second_plain_text[block_n][block_size*10];


    for(int i=0;i<block_n;i++) {
        for(int j=0;j<block_size*10;j++) {
```

```
            xor_results[i][j]=(byte)'\0';
            cipher_text[i][j]=(byte)'\0';
            second_plain_text[i][j]=(byte)'\0';
        }
    }

    //first block - xor
    for(int j=0;j<block_size;j++) {
        xor_results[0][j]=text_blocks[0][j] ^ npub[j];
    }

    int* keep_array_enc=(int*)(malloc(sizeof(int)*block_n));
    int* keep_array_dec=(int*)(malloc(sizeof(int)*block_n));

    keep_array_enc[0] = romulusEncryption(cipher_text[0],&cipher_text_lens[0],
            xor_results[0],strlen(xor_results[0]),
            ad,strlen(ad),nsec,npub,k
        );

    for(int i=1;i<block_n;i++) {
        //xor
        for(int j=0;j<block_size;j++) {
            xor_results[i][j]=text_blocks[i][j] ^ cipher_text[i-1][j];
        }

        keep_array_enc[i]=romulusEncryption(cipher_text[i],&cipher_text_lens[i
],
            xor_results[i],strlen(xor_results[i]),
            ad,strlen(ad),nsec,npub,k
        );
    }

    printf("Text Encrypted CBC Mode : ");
  for(int i =0; i<block_n; ++i ){
    for(int j=0;j<block_size;j++) {
      printf("%c", cipher_text[i][j]);
    }
  }
  printf("\n");

  byte decryp_result[block_n+1][block_size*10];
    unsigned long long decryp_result_lens[block_n+1];


    for(int i=0;i<block_n;i++) {
        for(int j=0;j<block_size*100;j++) {
            decryp_result[i][j]='\0';
        }
    }
```

```
    //first block decrpytion

    keep_array_dec[0]=romulusDecryption(decryp_result[0],&decryp_result_lens[0
],
                                nsec,cipher_text[0],strlen(cipher_text[0]),
                               ad,strlen(ad),npub,k
    );


    //first block xor
    for(int j=0;j<block_size;j++) {
        second_plain_text[0][j]=decryp_result[0][j] ^ npub[j];
    }


    for(int i=1;i<block_n;i++) {
            keep_array_dec[i]=romulusDecryption(decryp_result[i],&decryp_resul
t_lens[i],
                                nsec,cipher_text[i],strlen(cipher_text[i]),
                               ad,strlen(ad),npub,k);


        for(int j=0;j<block_size;j++) {
            second_plain_text[i][j]=decryp_result[i][j] ^ cipher_text[i-1][j];
        }
    }

    printf("Text Decrypted CBC Mode : ");
    for(int i =0; i<block_n; ++i ){
      for(int j=0;j<block_size;j++) {
        printf("%c", second_plain_text[i][j]);
      }
    }
    printf("\n");

    int flag=0;
    for(int i=0;i<block_n;i++) {
      for(int j=0;j<block_size;j++) {
        if(second_plain_text[i][j]!=text_blocks[i][j]) {
          flag=1;
          break;
        }
      }
      if(flag) {
        break;
      }
    }
    if(!flag) {
```

```
      printf("Success, test passed\n");
    }

    free(keep_array_enc);
    free(keep_array_dec);
    free(new_plain_text);

}
```

OFB Output Feedback Mode

```
void OFBMode(unsigned char*c,unsigned long long* clen,
       const unsigned char* m,const unsigned char* ad,
             unsigned char* nsec,
       const unsigned char* npub,
       const unsigned char* k) {

  printf("\nOFB MODE : \n");
  printf("Text plaintext : %s\n",m);

  int block_n=5;

  int new_plain_text_len=strlen(m)+strlen(m)%block_n;
  char* new_plain_text=(char*)malloc(sizeof(char)*new_plain_text_len);


  for(int i=0;i<new_plain_text_len;i++) {
    new_plain_text[i]='\0';
  }
  for(int i=0;i<strlen(m);i++) {
    new_plain_text[i]=m[i];
  }


  int block_size = ((new_plain_text_len)/block_n);
  byte text_blocks[block_n][block_size];
  for( int i =0; i<block_n; ++i ){
    for(int j = 0; j<block_size; ++j )
      text_blocks[i][j] = (byte)'\0';
  }
  for( int i =0; i<block_n; ++i ){
    for(int j = 0; j<block_size; ++j )
      text_blocks[i][j] = (byte)new_plain_text[i*block_size + j];
  }


  unsigned long long encryp_result_lengths[block_n];
```

```
  byte encryp_results[block_n][block_size*10];
  byte cipher_text[block_n][block_size*10];

  for( int i =0; i<block_n; ++i ){
    for(int j = 0; j<block_size*10; ++j ){
      encryp_results[i][j] = (byte)'\0';
      cipher_text[i][j] = (byte)'\0';
    }
  }

    int* keep_array_enc=(int*)(malloc(sizeof(int)*block_n));
    int* keep_array_dec=(int*)(malloc(sizeof(int)*block_n));


  //first block

    keep_array_enc[0]=romulusEncryption(encryp_results[0],&encryp_result_lengt
hs[0],
      npub,strlen(npub),ad,strlen(ad),nsec,npub,k
    );

  //first xor
  for(int j=0;j<block_size;j++) {
    cipher_text[0][j]=text_blocks[0][j] ^ encryp_results[0][j];
  }

  for(int i=1;i<block_n;i++) {
    keep_array_enc[i]=romulusEncryption(encryp_results[i],&encryp_result_lengt
hs[i],
      encryp_results[i-1],strlen(encryp_results[i-
1]),ad,strlen(ad),nsec,npub,k
    );


    for(int j=0;j<block_size;j++) {
      cipher_text[i][j]=text_blocks[i][j] ^ encryp_results[i][j];
    }
  }

  printf("Text Encrypted OFB mode : ");
  for(int i =0; i<block_n; ++i ){
    for(int j=0;j<block_size;j++) {
      printf("%c", cipher_text[i][j]);
    }
  }
  printf("\n");


  byte plain_text_deciphered[block_n][block_size];
```

```
  for(int i=0;i<block_n;i++) {
    for(int j=0;j<block_size;j++) {
      plain_text_deciphered[i][j]=(byte)'\0';
    }
  }


  unsigned long long decryp_result_lengths[block_n];
  byte decryp_results[block_n][block_size*10];
  byte decipher_text[block_n][block_size*10];

  for( int i =0; i<block_n; ++i ){
    for(int j = 0; j<block_size*10; ++j ){
      decryp_results[i][j] = (byte)'\0';
      decipher_text[i][j] = (byte)'\0';
    }
  }


  //first block

  keep_array_dec[0]=romulusEncryption(decryp_results[0],&decryp_result_lengths
[0],
    npub,strlen(npub),ad,strlen(ad),nsec,npub,k
  );

  for(int j=0;j<block_size;j++) {
    decipher_text[0][j]=cipher_text[0][j] ^ decryp_results[0][j];
  }

  for(int i=1;i<block_n;i++) {
    keep_array_dec[i]=romulusEncryption(decryp_results[i],&decryp_result_lengt
hs[i],
      decryp_results[i-1],strlen(decryp_results[i-
1]),ad,strlen(ad),nsec,npub,k
    );

    for(int j=0;j<block_size;j++) {
      decipher_text[i][j]=cipher_text[i][j] ^ decryp_results[i][j];
    }
  }

  printf("Text Decrypted OFB mode : ");
  for(int i =0; i<block_n; ++i ){
    for(int j=0;j<block_size;j++) {
      printf("%c", decipher_text[i][j]);
    }
  }
```

```
  int flag=0;
  for(int i =0; i<block_n; ++i ){
    for(int j=0;j<block_size;j++) {
      if(decipher_text[i][j]!=text_blocks[i][j]) {
        flag=1;
        break;
      }
    }
    if(flag) {
      break;
    }
  }
  printf("\n");
  if(!flag) {
    printf("Test passed\n");
  }

  printf("\n");

  free(new_plain_text);

  free(keep_array_dec);
  free(keep_array_enc);


}
```

Output Results of both TinyJambu and Romulus Algorithms :

```
PART A

File 'plaintext.txt' reading
First plaintext : This is TinyJambu algorithm.
First plaintext Encrypted : [?iIz&MoLUĚM7O
First plaintext Encrypted Length : 37
First plaintext Decrypted: This is TinyJambu algorithm.

Second plaintext : Yakup Talha Yolcu
Second plaintext Encrypted : Yq+*Doadhrq*
Second plaintext Encrypted Length : 25
Second plaintext Decrypted: Yakup Talha Yolcu

PART B

CBC MODE :
Text plaintext : GEBZETECHNICALUNIVERSITY
Text Encrypted CBC mode :C3#4-O:!( _ >9+
Text Decrypted CBC mode :GEBZETECHNICALUNIVERSITY

OFB MODE :
Text plaintext : GEBZETECHNICALUNIVERSITY
Text Encrypted OFB mode : Cewp|@A{eqgU
Text Decrypted OFB mode : GEBZETECHNICALUNIVERSITY
talha@LAPTOP-TH6JIR3U:/mnt/d/GTU/4.SINIF/CSE470 Cryptography
```

```
talha@LAPTOP-TH6JIR3U:/mnt/d/GTU/4.SINIF/CSE470 Cryptography and Comput
Test of the Romulus light-weight cipher
Plaintext: YakupTalhaYolcu
Key: 0123456789ABCDEF0123456789ABCDEF
Nonce: 000000000000111111111111
Additional Information: YakupTalhaYolcu

Plaintext: YakupTalhaYolcu
Cipher: B65FBDF41A657B140D3CE096C65BCF59, Len: 31
Plaintext: YakupTalhaYolcu, Len: 15
Romulus Cipher passed from the test successfully!
Test of the Romulus light-weight cipher
Plaintext: GebzeTechnicalUniversity
Key: 0123456789ABCDEF0123456789ABCDEF
Nonce: 000000000000111111111111
Additional Information: YakupTalhaYolcu

Plaintext: GebzeTechnicalUniversity
Cipher: 22965233683639E2C7E65142C80D5C967FDF5A3C, Len: 40
Plaintext: GebzeTechnicalUniversity, Len: 24
Romulus Cipher passed from the test successfully!
Test of the Romulus Algorithm with Cipher Block Chaining (CBC) mode
Plaintext: YakupTalha
Key: 0123456789ABCDEF0123456789ABCDEF
Nonce: 000000000000111111111111
Additional Information: YakupTalhaYolcu

Plaintext: YakupTalha
Text Encrypted CBC Mode : nW[3R"
Text Decrypted CBC Mode : YakupTalha
Success, test passed
Test of the Romulus Algorithm with Cipher Block Chaining (OFB) mode
Plaintext: YakupTalhaYolcu
Key: 0123456789ABCDEF0123456789ABCDEF
Nonce: 000000000000111111111111
Additional Information: YakupTalhaYolcu

Plaintext: YakupTalhaYolcu

OFB MODE :
Text plaintext : YakupTalhaYolcu
Text Encrypted OFB mode : zp6□(□□E
Text Decrypted OFB mode : YakupTalhaYolcu
Test passed

talha@LAPTOP-TH6JIR3U:/mnt/d/GTU/4.SINIF/CSE470 Cryptography and Comput
```