CSE 470 CRYPTOGRAPHY AND INFORMATION SECURITY

1801042609

YAKUP TALHA YOLCU

**TinyJAMBU:**

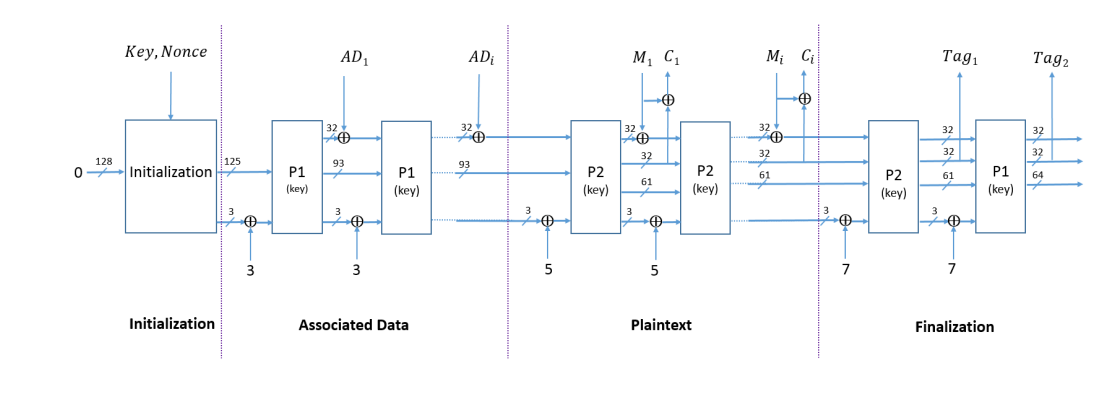
TinyJAMBU, designed by Wu and Huang , is an AEAD scheme that is inspired by the third-round candidate of the CAESAR competition, JAMBU. The main component of TinyJAMBU is a 128-bit keyed permutation without a key schedule that is based on a nonlinear feedback shift register, and the nonlinearity in each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

tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu

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.

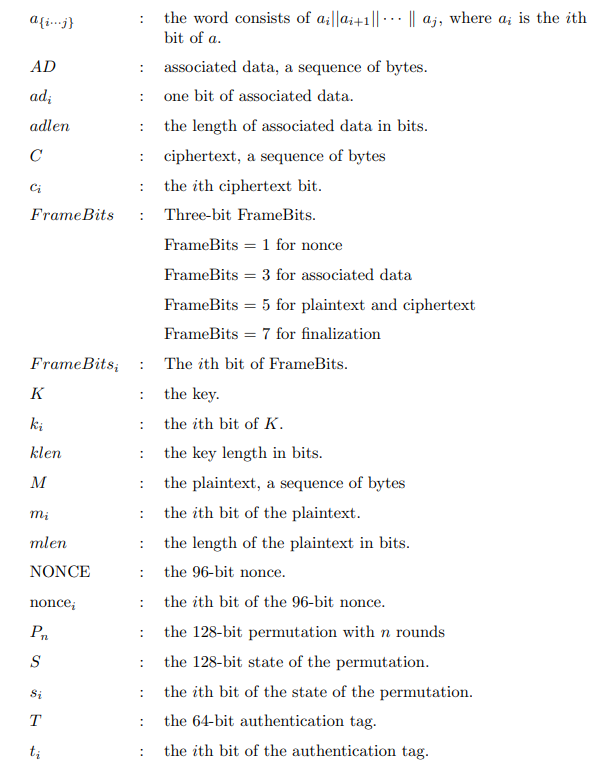
The TinyJAMBU mode is a small variant of the JAMBU mode which is a third-round candidate of the CAESAR competition. In the TinyJAMBU mode, a 128-bit keyed permutation is used, the state size is 128 its, and the message block size is 32 bits. When nonce is reused, the TinyJAMBU mode provides better authentication 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

*metin içeren bir resim

Açıklama otomatik olarak oluşturuldu*

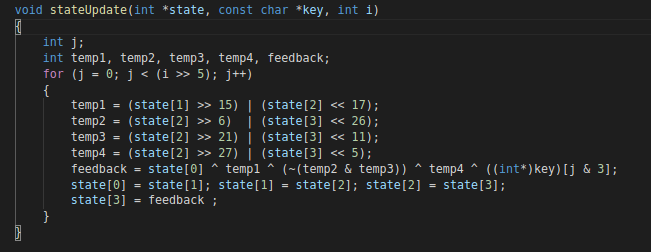
**

**The Keyed Permutation P**

metin içeren bir resim

Açıklama otomatik olarak oluşturulduIn 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

**Code:**



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

saat içeren bir resim

Açıklama otomatik olarak oluşturuldu

**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.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Code**:

metin, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**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.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

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.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Code:**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**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:**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**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

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**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.

metin, kişi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

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

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**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}.

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**Processing the full blocks of ciphertext:

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.

metin, kişi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Code:**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**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

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

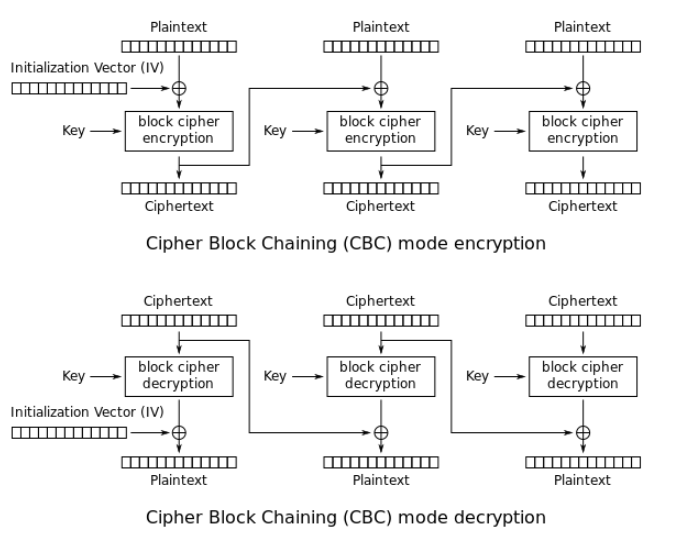
**Code:**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**CBC (Cipher Block Chaining Mode)**

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



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]);

        }}

    }

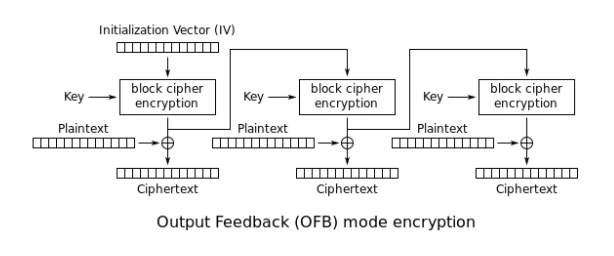
    printf("\n");

}

**OFB Output Feedback Mode**

The output feedback (OFB) mode makes a block cipher into a synchronous [stream cipher](https://en.wikipedia.org/wiki/Stream_cipher). It generates [keystream](https://en.wikipedia.org/wiki/Keystream) blocks, which are then [XORed](https://en.wikipedia.org/wiki/XOR) 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](https://en.wikipedia.org/wiki/Error-correcting_code) to function normally even when applied before encryption.

**OFB** is an [AES](https://www.educative.io/edpresso/what-is-the-aes-algorithm) block cipher mode similar to the [CFB](https://www.educative.io/edpresso/what-is-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.



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 ≤ i ≤ 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:

tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu

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\_result\_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\_lengths[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\_lengths[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\_lengths[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);

}

metin içeren bir resim

Açıklama otomatik olarak oluşturulduOutput Results of both TinyJambu and Romulus Algorithms :

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu