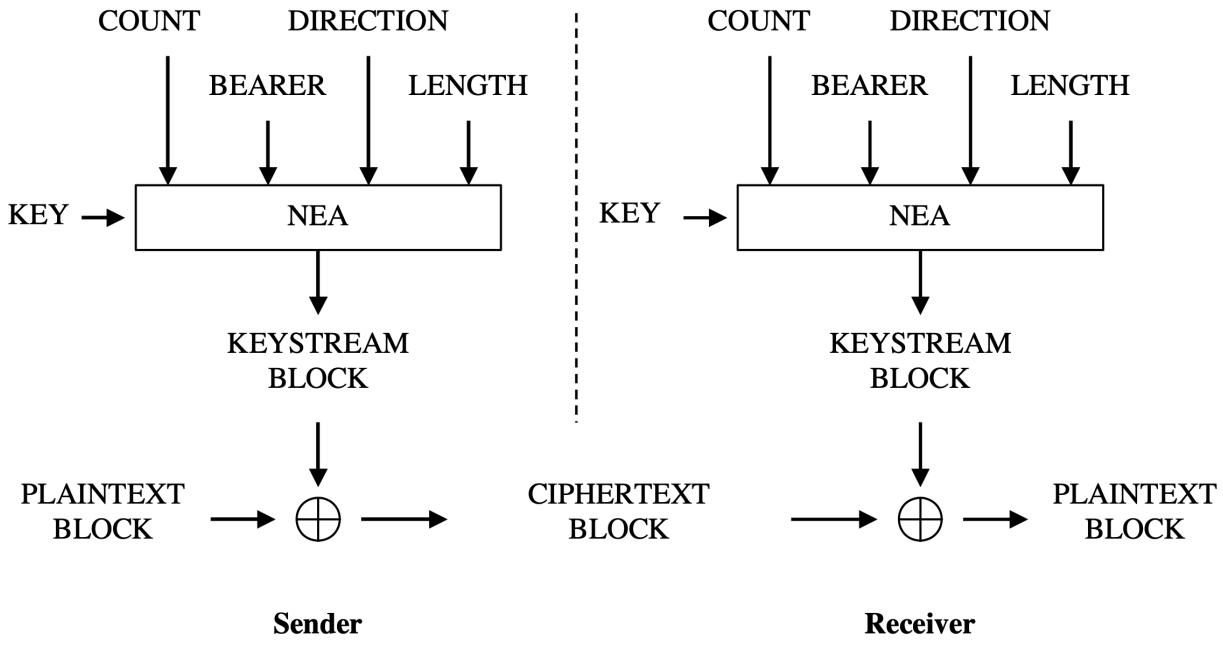
Project 2: Use of open source crypto library for data encryption and integrity check

In the second project, you will learn how to use an open source crypto library, openssl, for data encryption and integrity check. You should submit two separate programs, one for ciphering and the other for integrity check. This document includes four parts. The last two include sample code for you to get familiar with openssl. To finish this project you may want to install openssl from its source code. Please feel free to search online tutorials on how to install openssl. For example, this link provides basic steps about openssl installation: https://www.tecmint.com/install-openssl-from-source-in-centos-ubuntu/.

Part 1: Ciphering

Please implement the ciphering algorithm described in this part whose input and output are given as follows:

- Input: Plaintext, Key, Count, Bearer, Direction, Length
- Output: Ciphertext



The input parameters to the ciphering algorithm are a 128-bit cipher key named KEY, a 32-bit COUNT, a 5-bit bearer identity BEARER, the 1-bit direction of the transmission i.e. DIRECTION, and the length of the keystream required i.e. LENGTH. The DIRECTION bit shall be 0 for uplink and 1 for downlink.

The above figure illustrates the use of the ciphering algorithm NEA to encrypt plaintext by applying a keystream using a bit per bit binary addition of the plaintext and the keystream. The

plaintext may be recovered by generating the same keystream using the same input parameters and applying a bit per bit binary addition with the ciphertext.

The input parameter LENGTH shall affect only the length of the KEYSTREAM BLOCK, not the actual bits in it.

The NEA algorithm is based on 128-bit AES in CTR mode. An example program using the 128-bit AES in CTR mode based on openssl can be found in Part 3.

The sequence of 128-bit counter blocks needed for CTR mode T1, T2, ..., Ti, ... shall be constructed as follows:

The most significant 64 bits of T1 consist of COUNT[0] .. COUNT[31] | BEARER[0] .. BEARER[4] | DIRECTION | 0^26 (i.e., 26 zero bits). These are written from most significant on the left to least significant on the right, so for example COUNT[0] is the most significant bit of T1. The least significant 64 bits of T1 are all 0.

Subsequent counter blocks are then obtained by applying the standard integer incrementing function mod 2^64 to the least significant 64 bits of the previous counter block.

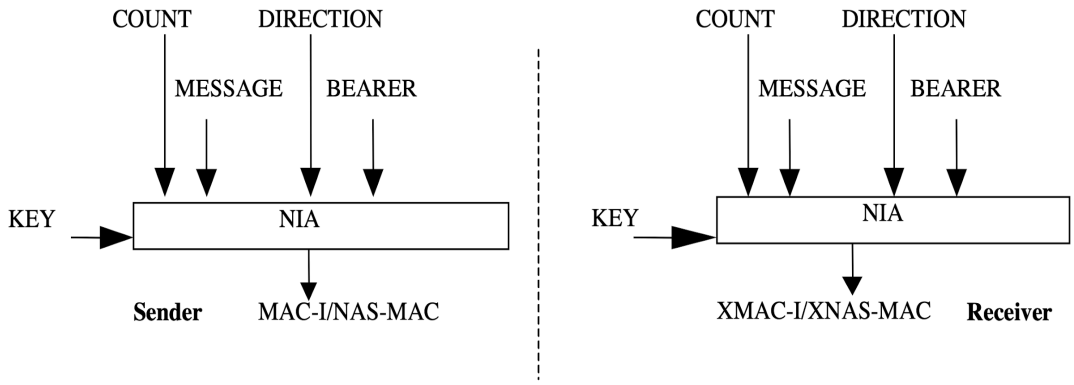
Part 2: Integrity check

Please implement the integrity check algorithm described in this part whose input and output are given as follows:

- Input: Message, Key, Count, Direction, Bearer
- Output: MAC (Message Authentication Code)

The input parameters to the integrity algorithm are a 128-bit integrity key named KEY, a 32-bit COUNT, a 5-bit bearer identity called BEARER, the 1-bit direction of the transmission i.e. DIRECTION, and the message itself i.e MESSAGE. The DIRECTION bit shall be 0 for uplink and 1 for downlink. The bit length of the MESSAGE is LENGTH.

The following figure illustrates the use of the integrity algorithm EIA to authenticate the integrity of messages.



Based on these input parameters the sender computes a 32-bit message authentication code (MAC-I/NAS-MAC) using the integrity algorithm NIA. The message authentication code is then appended to the message when sent. For integrity protection algorithm, the receiver computes the expected message authentication code (XMAC- I/XNAS-MAC) on the message received in the same way as the sender computed its message authentication code on the message sent and verifies the data integrity of the message by comparing it to the received message authentication code, i.e., MAC-I/NAS-MAC.

The bit length of MESSAGE is BLENGTH.

The input to CMAC mode is a bit string M of length Mlen (see [17, section 5.5]). M is constructed as follows: M0 .. M31 = COUNT[0] .. COUNT[31]

M32 .. M36 = BEARER[0] .. BEARER[4]

M37 = DIRECTION

M38 .. M63 = 0^26 (i.e. 26 zero bits)

M64 .. $M_{BLENGTH+63}$ = MESSAGE[0] .. MESSAGE[BLENGTH-1]

and so Mlen = BLENGTH + 64.

AES in CMAC mode (see Part 4) is used with these inputs to produce a Message Authentication Code T (MACT) of length Tlen = 32.

T is used directly as the NIA output MACT[0] .. MACT[31], with MACT[0] being the most significant bit of T.

Part 3: 128-bit AES in CTR mode

This can be implemented with function CRYPTO_ctr128_encrypt of the openssl library. The signature of this function can be found at this link: https://docs.huihoo.com/doxygen/openssl/1.0.1c/ctr128_8c.html#af30a43dd23d0d25cc94a01f1a41f0746. The following gives an example program that uses 128-bit AES in CTR mode.

```
//++++++++++++++++++++++++++++++++++++
#include <openssl/aes.h>
#include <openssl/modes.h>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/buffer.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define u8 unsigned char

struct ctr_state
{
    unsigned char ivec[AES_BLOCK_SIZE];
    unsigned int num;
    unsigned char ecount[AES_BLOCK_SIZE];
};

void init_ctr(struct ctr_state *state, const unsigned char iv[16])
{
    state->num = 0;
    memset(state->ecount, 0, 16);
    memcpy(state->ivec, iv, 16);
}

void crypt_message(const u8* src, u8* dst, unsigned int src_len, const
AES_KEY* key, const u8* iv)
{
   struct ctr_state state;
   init_ctr(&state, iv);
   CRYPTO_ctr128_encrypt(src, dst, src_len, key, state.ivec, state.ecount,
&state.num, (block128_f)AES_encrypt);
}

int main()
{
   int len;
   char source[128];
   char cipher[128];
   char recovered[128];
   unsigned char iv[AES_BLOCK_SIZE];

   const unsigned char* enc_key = (const unsigned char*)"123456789abcdef0";

   if(!RAND_bytes(iv, AES_BLOCK_SIZE))
   {
```

```
        fprintf(stderr, "Could not create random bytes.");
        exit(1);
    }

    AES_KEY key;
    AES_set_encrypt_key(enc_key, 128, &key);

    strcpy(source, "quick brown fox jumped over the lazy dog what.");
    len = strlen(source);
    memset(recovered, 0, sizeof(recovered));
    crypt_message((const u8*)source, (u8*)cipher, len, &key, iv);
    crypt_message((const u8*)cipher, (u8*)recovered, len, &key, iv);
    printf("Recovered text:%s\n", recovered);

    strcpy(source, "a dog he is idiot who is the genius.");
    len = strlen(source);
    memset(recovered, 0, sizeof(recovered));
    crypt_message((const u8*)source, (u8*)cipher, len, &key, iv);
    crypt_message((const u8*)cipher, (u8*)recovered, len, &key, iv);
    printf("Recovered text:%s\n", recovered);

    return 0;
}
//++++++++++++++++++++++++++++++++
```

Part 4: AES-CMAC Algorithm

The description of the AES-CMAC algorithm can be found here: https://datatracker.ietf.org/doc/rfc4493/. At the end of this document, you can find the reference code for the AES-CMAC algorithm. Please feel free to include this code in your project. For the AES_128 function used in the code, you should use the openssl implementation. An example code snippet can be found as follows.

```
/*
 * An example of using the AES block cipher,
 * with key (in hex) 01000000000000000000000000000000
 * and input (in hex) 01000000000000000000000000000000.
 * The result should be the output (in hex) FAEB01888D2E92AEE70ECC1C638BF6D6
 * (AES_encrypt corresponds to computing AES in the forward direction.)
 * We then compute the inverse, and check that we recovered the original
input.
 * (AES_decrypt corresponds to computing the inverse of AES.)
 */

#include "openssl/aes.h"
#include <stdio.h>

main(){
  AES_KEY AESkey;
  unsigned char MBlock[16];
  unsigned char MBlock2[16];
  unsigned char CBlock[16];
  unsigned char Key[16];
  int i;

  /*
   * Key contains the actual 128-bit AES key. AESkey is a data structure
   * holding a transformed version of the key, for efficiency.
   */

  Key[0]=1;

  for (i=1; i<=15; i++) {
    Key[i] = 0;
    }

  AES_set_encrypt_key((const unsigned char *) Key, 128, &AESkey);

  MBlock[0] = 1;

  for (i=1; i<16; i++)
    MBlock[i] = 0;

  AES_encrypt((const unsigned char *) MBlock, CBlock, (const AES_KEY *)
&AESkey);

  for (i=0; i<16; i++)
    printf("%X", CBlock[i]/16), printf("%X", CBlock[i]%16);
  printf("\n");
```

```
  /*
   * We need to set AESkey appropriately before inverting AES.
   * Note that the underlying key Key is the same; just the data structure
   * AESkey is changing (for reasons of efficiency).
   */
  AES_set_decrypt_key((const unsigned char *) Key, 128, &AESkey);

  AES_decrypt((const unsigned char *) CBlock, MBlock2, (const AES_KEY *)
&AESkey);

  for (i=0; i<16; i++)
    printf("%X", MBlock2[i]/16), printf("%X", MBlock2[i]%16);
  printf("\n");
}
```