Crypto Collection - v0.1

HW3 - CNS Sapienza

Luigi Russo 1699981

13/11/2018

For updated version visit https://www.gitlab.com/lrusso96

1 Introduction

I've decided to list C and C++ cryptographic libraries. For each library I will give a brief overview (license, language, etc.). Then I will give some details about the following cryptographic fields:

- Key generation
- PKCS
- Hash
- MAC
- Block Cipher
- Cipher Modes

, specifying for each of these what kind of algorithms / protocols they actually support. Then (except for PCKS and MAC) I will give some basic API and in some cases I will present some simplified examples: here basic API expression means a very simple schema to do basic cryptographic operations.

Note: this report is not somehow *The Ultimate Cryptographic Guide*; for full API reference I will provide the links to the actual manuals. This report is just a simple collection that aims to provide a very basic idea of how API of each library has been projected and how you can implement simple snippets of code. The examples are often grabbed from the official manuals, and where possible, have been further simplified.

1.1 Changelog

v0.1 (13/11/2018)

- OpenSSL, Nettle, Botan and Crypto++ support.
- basic API and examples for: Key generation, HASH, Block Cipher and Cipher Modes.
- List of algorithms and protocols supported for the above categories plus MAC and PKCS.

2 Current Libraries

C

- OpenSSL
- Nettle

C++

- Botan
- Crypto++

3 Notable Libraries not (yet) included

- Themis - C++ (Apache 2.0)
- NaCl - C++ (Public domain)
- Libgcrypt - C (GPL)
- libtomcrypt - C (Public domain)
- HElib - C++ (Apache 2.0)

4 Beyond C and C++

I've focused on C and C++ libraries, because of the great effort made to develop robust, efficient cryptographic algorithms and protocols in these two languages. However in near future it would be nice to add some other languages (e.g. Java).

Infobox
License: Apache 1.0 / 4-BSD
Language: C
Company: The OpenSSL Project
Last version: 1.1.1 (**2018**)
Website: https://www.openssl.org
Source code: https://www.github.com/openssl/openssl

4.1 Key generation and exchange

DH	EDH	DSA	RSA
YES	YES	YES	YES

API basic reference

Must use *evp* interface: first choose the proper context (e.g *DH*). Then initialize a public key algorithm context using key pkey for shared secret derivation. The return value is 1 for success, 0 or a negative value for failure: in particular -2 indicates the operation is not supported by the public key algorithm.

```
int EVP_PKEY_derive_init(EVP_PKEY_CTX *ctx);
```

To set the peer key, that is normally a public key, you need to call:

```
int EVP_PKEY_derive_set_peer(EVP_PKEY_CTX *ctx, EVP_PKEY *peer);
```

Finally you can derive a shared secret using the context. If key is NULL then the maximum size of the output buffer is written to the keylen parameter, otherwise the keylen parameter should contain the length of the key buffer: at the end the output shared key is written to key and its length is written to keylen.

```
int EVP_PKEY_derive(EVP_PKEY_CTX *ctx, unsigned char *key, size_t *keylen);
```

Example - Key agreement

The following code sample shows how a private/public key pair and a public key of some peer can be combined to derive the shared secret.

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

  unsigned char *skey;  //shared key
  size_t skeylen;
  EVP_PKEY *peerkey;  //public key of some peer
  EVP_PKEY *pkey;  //private/public key
  EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey);
  if (!ctx)
```

```
  // error occurred
if (EVP_PKEY_derive_init(ctx) <= 0)
  // error
if (EVP_PKEY_derive_set_peer(ctx, peerkey) <= 0)
  // error
// determine buffer length
if (EVP_PKEY_derive(ctx, NULL, &skeylen) <= 0)
  // error
skey = OPENSSL_malloc(skeylen);
if (!skey)
  // malloc failure
if (EVP_PKEY_derive(ctx, skey, &skeylen) <= 0)
  //error

/* Shared secret is skey bytes written to buffer skey */
```

4.2 Public key cryptography standard

PKCS #1	PKCS #5	PKCS #8	PKCS #11	PKCS #12
YES	YES	YES	YES	YES

4.3 Hash

SHA-1	SHA-2	SHA-3
YES	YES	YES

API basic reference

All is managed by EVP_MD class. First of all you need to create a *MessageDigest* and initialize it:

```
EVP_MD_CTX* EVP_MD_CTX_new();
int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
```

where the default impl is SHA-1 (to use it, impl can be left as NULL). To process messages and compute the partial digests it is possible to call:

```
int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, size_t cnt);
```

The last step is to produce the final digest with the function:

```
int EVP_DigestFinal_ex(EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s);
```

without forgetting to release memory to prevent leaks, calling simply:

```
void EVP_MD_CTX_free(EVP_MD_CTX *ctx);
```

Example - SHA-1

```
#include <stdio.h>
#include <openssl/evp.h>

int main(){
  EVP_MD_CTX *mdctx;
  const EVP_MD *md;
  char mess1[] = "Test Message\n";
```

```
  char mess2[] = "Hello World\n";
  unsigned char md_value[EVP_MAX_MD_SIZE];
  int md_len, i;

  // ckeck for support of SHA-1
  OpenSSL_add_all_digests();
  md = EVP_get_digestbyname("SHA-1");
  if(!md)
    // error

  // create and initialize md context
  mdctx = EVP_MD_CTX_new();
  EVP_DigestInit_ex(mdctx, md, NULL);

  // compute digests (partial results are concatenated)
  EVP_DigestUpdate(mdctx, mess1, strlen(mess1));
  EVP_DigestUpdate(mdctx, mess2, strlen(mess2));

  // get final digest and clean resource
  EVP_DigestFinal_ex(mdctx, md_value, &md_len);
  EVP_MD_CTX_free(mdctx);

  // print digest
  printf("Digest is: ");
  for(i = 0; i < md_len; i++)
    printf("%02x", md_value[i]);
  printf("\n");

  // Final cleanup
  EVP_cleanup();
  exit(0);
}
```

4.4 MAC

HMAC-MD5	HMAC-SHA-1	HMAC-SHA-2	Poly1305-AES	BLAKE2-MAC
YES	YES	YES	YES	YES

4.5 Block ciphers

AES	Camellia	3DES	Blowfish	Twofish
YES	YES	YES	YES	-

4.6 Modes of operations

ECB	CBC	OFB	CFB	CTR	CCM	GCM	OCB
YES	YES	YES	YES	YES	YES	YES	YES

API basic reference

First of all it is necessary to set up a context:

```
EVP_CIPHER_CTX_new())
```

After that you have to initialize the encryption operation, with something like this:

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
   ENGINE *impl, const unsigned char *key, const unsigned char *iv);
```

This function sets up cipher context ctx, created tih the function seen before, for encryption with cipher type from ENGINE impl. type is normally supplied by a function such as EVP_aes_256_cbc(). If impl is NULL then the default implementation is used. key is the symmetric key to use and iv is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher. To start computing the encryption of the plaintext blocks it is necessary to invoke the update function:

```
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                  int *outl, const unsigned char *in, int inl);
```

that encrypts inl bytes from the buffer in and writes the encrypted version to out. This function can be called multiple times to encrypt successive blocks of data. To encrypt the last block:

```
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);
```

that returns the computed ciphertext. To prevent memory leaks remember to clean the context, with the counterpart function of the initial EVP_CIPHER_CTX_new()):

```
EVP_CIPHER_CTX_free())
```

The decryption uses the same syntax except for the keyword Decrypt instead of Encrypt: for this reason the decrypted functions are not presented here, but only in the example below.

Example - AES256-CBC

Simple encryption and decryption functions that wrap the functions described above

```
int encrypt(unsigned char *plaintext, int plaintext_len,
 unsigned char *key, unsigned char *iv, unsigned char *ciphertext){

  EVP_CIPHER_CTX *ctx;
  int len;
  int ciphertext_len;

  //Create and initialise the context */
  if(!(ctx = EVP_CIPHER_CTX_new()))
    //error
  // Init the encryption operation
    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv))
      //error

  // provide the message to be encrypted, and obtain the encrypted output.
  if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
    //error

  // Final encryption
  ciphertext_len = len;
  if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
    //error
```

```
  ciphertext_len += len;

  /* Clean up */
  EVP_CIPHER_CTX_free(ctx);

  return ciphertext_len;
}
```

Nettle

Infobox
License: GNU LGPLv3
Language: C
Company: -
Last version: 3.4 (**2017**)
Website: http://www.lysator.liu.se/ nisse/nettle/
Source code: https://git.lysator.liu.se/nettle/nettle

4.7 Key generation and exchange

DH	EDH	DSA	RSA
-	-	YES	YES

API basic reference

Nettle supports only RSA and DSA and in the next lines RSA basic API will be discussed. First of all a pair of key is needed, one public and another private. They can be generated with:

```
void rsa_public_key_init (struct rsa_public_key *pub)
void rsa_private_key_init (struct rsa_private_key *key)
```

that store the keys to ad-hoc objects. Then the following:

```
int rsa_public_key_prepare (struct rsa_public_key *pub)
int rsa_private_key_prepare (struct rsa_private_key *key)
```

are used to compute the octet size of the key and moreover to check is the keys can be used (PKCS #1 standard). After that it is possible both to encrypt and decrypt messages with the following:

```
int rsa_encrypt (const struct rsa_public_key *key, void *random_ctx,
                         nettle_random_func *random, size_t length,
                         const uint8_t *cleartext, mpz_t ciphertext)

 int rsa_decrypt (const struct rsa_private_key *key,
           size_t *length, uint8_t *cleartext, const mpz_t ciphertext)
```

4.8 Public key cryptography standard

PKCS #1	PKCS #5	PKCS #8	PKCS #11	PKCS #12
YES	YES	-	-	-

4.9 Hash

SHA-1	SHA-2	SHA-3
YES	YES	YES

API basic reference

Nettle uses 3 different functions to handle hashes: init, update and digest. Let's see the functions for SHA-1: the others can be obtained simply replacing the prefix of the following functions. To create a hash state it is necessary to call

```
void sha1_init (struct sha1_ctx *ctx)
```

After that we can feed data to the hash state and concatenate them all with:

```
void sha1_update (struct sha1_ctx *ctx, size_t length, const uint8_t *data)
```

and finally, to get the result digest we can call:

```
void sha1_digest (struct sha1_ctx *ctx, size_t length, uint8_t *digest)
```

that also resets the state, ready to be reused for later digests.

4.10 MAC

HMAC-MD5	HMAC-SHA-1	HMAC-SHA-2	Poly1305-AES	BLAKE2-MAC
YES	YES	YES	YES	-

4.11 Block ciphers

AES	Camellia	3DES	Blowfish	Twofish
YES	YES	YES	YES	-

API basic reference

In this section AES-256 encryption algorithm is considered: the other supported algorithms can be used simply replacing the prefix of the following functions. To encrypt a block of text with a private key you have to call simply:

```
void aes256_encrypt (struct aes256_ctx *ctx, size_t length,
                             uint8_t *dst, const uint8_t *src)
```

The private key can be set with the following:

```
void aes256_set_encrypt_key (struct aes256_ctx *ctx, const uint8_t *key)
```

The context object must be initialized for encryption of course. The symmetric operation, i.e. the decryption, can be performed in the same way, simply replacing the kyword decrypt to the encrypt in the above functions. In this case the context must be initialized for decryption

4.12 Modes of operations

ECB	CBC	OFB	CFB	CTR	CCM	GCM	OCB
YES	YES	-	-	YES	YES	YES	-

API basic reference

We consider the CBC Mode of Operation. To encrypt a block it is necessary to call:

```
void cbc_encrypt (const void *ctx, nettle_cipher_func *f,
             size_t block_size, uint8_t *iv, size_t length,
                           uint8_t *dst, const uint8_t *src)
```

The decrypt function is the same, except for the suffix name (replace decrypt with encrypt). However Nettle suggests to not use directly these functions and make use, instead, of already defined MACROS. They are:

```
CBC_CTX (context_type, block_size)
```

that creates the context. Then, to setup the IV it is possible to use the following:

```
CBC_SET_IV (ctx, iv)
```

and finally start the encrypt/decrypt process with:

```
CBC_ENCRYPT (ctx, f, length, dst, src)
CBC_DECRYPT (ctx, f, length, dst, src)
```

Botan

Infobox
License: Simplified BSD
Language: C++ (Python support)
Company: Jack Lloyd
Last version: 2.8.0 (**2018**)
Website:https://botan.randombit.net/
Source code: https://github.com/randombit/botan

Description
Botan (Japanese for peony flower) is a C++ cryptography library taht offers the tools necessary to implement a range of practical systems, such as TLS protocol, X.509 certificates, modern AEAD ciphers, PKCS#11 and TPM hardware support, password hashing, and post quantum crypto schemes.

4.13 Key generation and exchange

DH	EDH	DSA	RSA
YES	YES	YES	YES

API basic reference

In order to achieve a key agreement it is necessary to create a Group (i.e. a domain) object (e.g. Elliptic Curve over prime fields). To generate the keys you can simply initialize two objects of class CTX_PrivateKey, where CTX is a supported context (e.g. ECDH for Elliptic Curves DH), passing a Random Number Generator object and the domain:

```
ECDH_PrivateKey(RandomNumberGenerator &rng,
    constEC_Group &domain, constBigInt &x = 0)
```

Since the key agreement aims to provide the same output to both parties, a function must wrap the computation that allows the two keys to be equal; this function in Botan is derive_key:

```
SymmetricKey derive_key(size_t key_len, const uint8_t in[],
                    size_t in_len, string &params="") const
```

where the uint8_t vector can be replaced by a std::vector as well. That's all: the SimmetricKey object returned is ready to be used.

Example - Key agreement

The code below performs an unauthenticated ECDH key agreement using the secp521r elliptic curve and applies the key derivation function KDF2(SHA-256) with 256 bit output length to the computed shared secret:

```
#include <botan/auto_rng.h>
#include <botan/ecdh.h>
#include <botan/ec_group.h>
#include <botan/pubkey.h>
#include <botan/hex.h>

using namespace Botan;
int main(){
  AutoSeeded_RNG rng;
```

```
  // ec domain and
  EC_Group domain("secp521r1");
  std::string kdf = "KDF2(SHA-256)";

  // generate ECDH keys
  ECDH_PrivateKey keyA(rng, domain);
  ECDH_PrivateKey keyB(rng, domain);

  // Construct key agreements
  PK_Key_Agreement ecdhA(keyA,rng,kdf);
  PK_Key_Agreement ecdhB(keyB,rng,kdf);

  // Agree on shared secret and derive symmetric key of 256 bit length
  secure_vector<uint8_t> sA = ecdhA.derive_key(32,keyB.public_value()).bits_of();
  secure_vector<uint8_t> sB = ecdhB.derive_key(32,keyA.public_value()).bits_of();

  if(sA != sB)
    return 1;

  std::cout << hex_encode(sA);
  return 0;
}
```

4.14 Public key cryptography standard

PKCS #1	PKCS #5	PKCS #8	PKCS #11	PKCS #12
YES	YES	YES	YES	-

4.15 Hash

SHA-1	SHA-2	SHA-3
YES	YES	YES

API basic reference

In the next lines both Botan and std namespaces will be omitted for simplicity. The class *HashFunction* manages the Hash. To create a HashFunction object use the static method below, specifying the type of algorithm that has to be used (e.g. "SHA-1"):

```
unique_ptr<HashFunction> HashFunction::create
        (const string& algo_spec, const string& provider = "")
```

Once the object is initialized, it is possible to update it several times, passing the new input to process (a somewhat vector) and its length; these inputs are concatenated to the previous ones and the partial result is stored in the HashFunction object:

```
void update(const uint8_t in[], size_t length)
```

At the end you can simply call the final method that returns the computed digest and resets the object for future and independent computations, i.e. clear the state of the object:

```
secure_vector<uint8_t> final()
```

Example - SHA-1

Here I put a very simple snippet of code, using the above methods, that computes the SHA-1 digest of some consecutive inputs and prints it:

```
#include <botan/hash.h>
#include <botan/hex.h>

using namespace Botan;
int main(){
  // hash object
  std::unique_ptr<HashFunction> hash1(HashFunction::create("SHA-1"));

  // process data and update hash object
  while(condition){
    value = ...
    hash1->update(value, VALUE_LEN);
  }

  // print the result and reset the object
  std::cout << hex_encode(hash1->final());
  return 0
}
```

4.16 MAC

HMAC-MD5	HMAC-SHA-1	HMAC-SHA-2	Poly1305-AES	BLAKE2-MAC
YES	YES	YES	YES	YES

4.17 Block ciphers

AES	Camellia	3DES	Blowfish	Twofish
YES	YES	YES	YES	YES

API basic reference

The class *BlockCipher* manages the block ciphers. First you need to initialize a BlockCipher object, passing the name of the algorithm:

```
unique_ptr<BlockCipher> BlockCipher::create(const string& algo_spec,
                                            const string& provider="")
```

and set the private key using:

```
void set_key(const SymmetricKey& key)
```

then it is possible to encrypt the plaintext block simply calling:

```
void encrypt(uint8_t block[])
```

and passing the current block; finally remember to clear the cipher object to handle future clean encryptions:

```
virtual void clear() = 0
```

Example - AES-256

```
#include <botan/block_cipher.h>
#include <botan/hex.h>

using namespace Botan;
int main(){
  // 256 bit key and 64 bit block to be used
  std::vector<uint8_t> key = hex_decode("0010.....ADC0");
```

```
std::vector<uint8_t> block = hex_decode("AA03..B771");

// cipher object
std::unique_ptr<BlockCipher> cipher(BlockCipher::create("AES-256"));

// set key and encrypt first block
cipher->set_key(key);
cipher->encrypt(block);
std::cout << hex_encode(block)

//reset cipher for next encryptions
cipher->clear();
...
```

4.18 Modes of operations

ECB	CBC	OFB	CFB	CTR	CCM	GCM	OCB
-	YES	YES	YES	YES	YES	YES	YES

API basic reference

The class *Cipher_Mode* manages the modes of operations. When creating a Cipher_Mode object it must be initialized with the proper mode (e.g. "CBC"):

```
static unique_ptr<Cipher_Mode> Cipher_Mode::create
    (const string& algo, Cipher_Dir direction, const string& provider="")
```

To encrypt the plaintext it is necessary to start from the Initialization Vector:

```
void start(const uint8_t nonce[], size_t nonce_len)
```

and then simply invoke on the object the finish of the process; Care about the side-effect of the below function: a copy could be necessary!

```
virtual void finish(secure_vector<uint8_t>& final_block, size_t offset=0)=0
```

Example - CBC

```
#include <botan/cipher_mode.h>
#include <botan/hex.h>

using namespace Botan;
int main(){
  // Random Number generator
  AutoSeeded_RNG rng;

  // plaintext and key
  const ::string plaintext("Hello world");
  std::vector<uint8_t> key = hex_decode("0010............ADC0");

  // create Mode object and set private key
  std::unique_ptr<Cipher_Mode> enc =
                    Cipher_Mode::create("AES-128/CBC/PKCS7", ENCRYPTION);
```

```
enc->set_key(key);

// generate fresh nonce (IV)
secure_vector<uint8_t> iv = rng.random_vec(enc->default_nonce_length());

// copy input to buffer
secure_vector<uint8_t> pt(plaintext.data(), plaintext.length());

// encrypt
enc->start(iv);
enc->finish(pt);
std::cout << hex_encode(pt)
return 0;
```

Infobox
License: Boost Software License (all individual files are public domain)
Language: C++
Company: The Crypto++ project
Last version: 7.0.0 (**2018**)
Website: https://www.cryptopp.com/
Source code: https://github.com/weidai11/cryptopp

4.19 Key generation and exchange

DH	EDH	DSA	RSA
YES	YES	YES	YES

API basic reference

First it is necessary to define a SimpleKeyAgreementDomain object, i.e. the domain for the key agreement (e.g. prime fields or binary fields). Once the domain has been created, you can easily create a key pair with the following function:

```
void GenerateKeyPair(RandomNumberGenerator &rng,
              byte *privateKey, byte *publicKey) const
```

At this point you can derive agreed value from your private key and counterpart's public key, simply calling on the domain object:

```
bool Agree(byte *agreedValue, const byte *privateKey, const
       byte *otherPublicKey, bool validateOtherPublicKey=true) const
```

If successful, the key agreement is done: the shared key is stored to agreedValue address and is ready to be used.

Example - Key agreement

This example shows a ECDH key agreement using NIST's 256 bit curve over the prime field.

```
OID CURVE = secp256r1();
AutoSeededRandomPool rng;

ECDH<ECP>::Domain dhA( CURVE ), dhB( CURVE );
SecByteBlock privA(dhA.PrivateKeyLength()), pubA(dhA.PublicKeyLength());
SecByteBlock privB(dhB.PrivateKeyLength()), pubB(dhB.PublicKeyLength());

dhA.GenerateKeyPair(rng, privA, pubA);
dhB.GenerateKeyPair(rng, privB, pubB);

if(dhA.AgreedValueLength() != dhB.AgreedValueLength())
  // error
```

```
SecByteBlock sharedA(dhA.AgreedValueLength()), sharedB(dhB.AgreedValueLength());
if(!dhA.Agree(sharedA, privA, pubB))
  //error
if(!dhB.Agree(sharedB, privB, pubA))
  //error

// use Integer for simplicity
Integer ssa, ssb;
ssa.Decode(sharedA.BytePtr(), sharedA.SizeInBytes());
ssb.Decode(sharedB.BytePtr(), sharedB.SizeInBytes());
cout << "(A): " << std::hex << ssa << endl;
cout << "(B): " << std::hex << ssb << endl;
```

4.20 Public key cryptography standard

PKCS #1	PKCS #5	PKCS #8	PKCS #11	PKCS #12
YES	YES	YES	YES	-

4.21 Hash

SHA-1	SHA-2	SHA-3
YES	YES	YES

API basic reference

The HashTransformation class manages hashes and digests in this library. To compute the digest of one single block it is possible to call:

```
void CalculateDigest(byte *digest, const byte *input, size_t length)
```

Instead, to concatenate more blocks of data it is possible to call the update function

```
void Update(const byte *input, size_t length)
```

many times. The final digest is computed simply calling:

```
void Final(byte *digest)
```

that stores the result to the address passed as parameter.

Example - SHA-1

A very simple example that shows how define a SHA-1 object and compute the digest of some data.

```
SHA hash;
byte digest[SHA::DIGESTSIZE];

hash.Update(pbData1, nData1Len);
hash.Update(pbData2, nData2Len);
hash.Update(pbData3, nData3Len);
hash.Final(digest);
```

4.22 MAC

HMAC-MD5	HMAC-SHA-1	HMAC-SHA-2	Poly1305-AES	BLAKE2-MAC
YES	YES	YES	YES	YES

4.23 Block ciphers

AES	Camellia	3DES	Blowfish	Twofish
YES	YES	YES	YES	YES

4.24 Modes of operations

ECB	CBC	OFB	CFB	CTR	CCM	GCM	OCB
YES	YES	YES	YES	YES	YES	YES	-

API basic reference

After initializing a Mode of operation object (e.g CFB_Mode) it is possible to encrypt the plaintext in a very simple way calling the below:

```
ProcessData(byte* cipherText, const byte* plainText, size_t messageLen)
```

The above function can be used both for encrypt and decrypt data for simmetric modes of operations (e.g. CFB). Remember to generate the Initialization Vector before encrypting any data with the following:

```
GenerateBlock(SecByteBlock key, size_t keylen)
```

to be called on a Random Number Generator object.

Example - CFB

```
AutoSeededRandomPool rnd;

// Generate a random key
SecByteBlock key(0x00, AES::DEFAULT_KEYLENGTH);
rnd.GenerateBlock(key, key.size());
```

```
// Generate a random IV
SecByteBlock iv(AES::BLOCKSIZE);
rnd.GenerateBlock(iv, iv.size());

byte plainText[] = "Hello! How are you.";
size_tmessageLen = std::strlen((char*)plainText) + 1;

// Encrypt
CFB_Mode<AES>::Encryption cfbEncryption(key, key.size(), iv);
cfbEncryption.ProcessData(plainText, plainText, messageLen);

// Decrypt
CFB_Mode<AES>::Decryption cfbDecryption(key, key.size(), iv);
cfbDecryption.ProcessData(plainText, plainText, messageLen);
```

References

Manuals and Docs

[1] *OpenSSL*
https://www.openssl.org/docs

[2] *Nettle*
http://www.lysator.liu.se/ nisse/nettle/nettle.html

[3] *Botan*
https://botan.randombit.net/manual

[4] *Crypto++*
https://www.cryptopp.com/docs/ref

Other

[5] *Wikipedia Cryptography Portal*
https://en.wikipedia.org/wiki/Portal:Cryptography