
# COCryL Documentation

**version 1.0**

**April 29, 2016**

# Contents

COCryL is a software library that provides various algorithms used to encrypt, sign and hash data. This library has been developed by Cyberens.

**About this manual** This manual provides a complete description of how to use the library and its various features. Each section corresponds to an algorithm used in cryptography.

The different algorithms are the following:

- AES
- SPECK
- RSA
- EdDSA
- ECIES
- SALSA
- SHA-2
- SHA-3
- PBKDF 2
- Generation of random elements

Each library function (except RandomInt) returns an integer that is an error code (if there is an error) or 0 if everything went well.

# 1 AES

AES or Advanced Encryption Standard, is a symmetric encryption algorithm. It has become a standard since 2002 in USA, described in the FIPS PUB 197. Its input is a 128-bit message and its output is a 128-bit cipher text. Depending on the version, the key length is 128 bits, 192 bits or 256 bits. To encrypt messages of different lengths, we use different encryption modes:

- ECB (Electronic Code Book)
- CBC (Cipher Block Chaining)
- OFB (Output Feedback)
- CTR (Counter)

These previous modes are described in the NIST Special Publication 800-38A.

To produce a message authentication code (MAC), we use the MAC mode. It is described in the NIST Special Publication 800-38B.

The last mode is the Galois Counter Mode, that encrypts the message using the CTR mode and products a tag using a hash function. It is described in the NIST Special Publication 800-38D.

Into the ECB, CBC, OFB and CTR mode, each message is padded, i.e. each cipher text has a length equal to $\left(\frac{length(message)}{16}+1\right)\times 16$.

## 1.1 ECB

ECB is the simplest mode. The message to encrypt is divided into blocks of 128 bits and each block is encrypted separately with the same key. We have 3 functions to:

- encrypt or decrypt a file
- encrypt or decrypt a buffer
- encrypt or decrypt a string

### 1.1.1 Encrypt or decrypt a file

To encrypt or decrypt a file with the AES-ECB algorithm, we use the following functions:

```
int AES_ECB_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char* ecbKey,
    unsigned int keySizeBits);

int AES_ECB_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char* ecbKey,
    unsigned int keySizeBits);
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ecbKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits.

Listing 1: Example of how to encrypt and decrypt a file with AES-ECB

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ecbKey = "0123456789ABCDEF0123456789ABCDEF";

err = AES_ECB_EncryptFile(message, cipher, ecbKey, 256);
if(err != 0) return err;
err = AES_ECB_DecryptFile(cipher, decipher, ecbKey, 256);
if(err != 0) return err;
```

### 1.1.2 Encrypt or decrypt a buffer

To encrypt or decrypt a buffer with the ECB mode, we use the following functions:

```
int AES_ECB_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned char* ecbKey,
    unsigned int keySizeBits)
```

```
int AES_ECB_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned char* ecbKey,
    unsigned int keySizeBits)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* ecbKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 2: Example of how to encrypt and decrypt a buffer with AES-ECB

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789ABCDEF0123456789ABCDEF";
message[10] = 't';

err = AES_ECB_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ecbKey, 256);
if(err != 0) return err;
err = AES_ECB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ecbKey, 256);
if(err != 0) return err;
```

### 1.1.3 Encrypt or decrypt a string

To encrypt or decrypt a string with the ECB mode, we use the following functions:

```
int AES_ECB_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* ecbKey, unsigned int keySizeBits)

int AES_ECB_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ecbKey, unsigned int keySizeBits)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer

- `unsigned char* outputBuffer`: the output buffer containing the resulting encryption
- `unsigned char* outputString`: the output string containing the resulting decryption
- `unsigned char* ecbKey`: the symmetric key
- `unsigned int keySizeBits`: the key length, 128, 192 or 256 bits

Listing 3: Example of how to encrypt and decrypt a string with AES-ECB

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789ABCDEF0123456789ABCDEF";

err = AES_ECB_EncryptString(message, cipher, &cipherLength, ecbKey, 256);
if(err != 0) return err;
err = AES_ECB_DecryptString(cipher, cipherLength, decipher, ecbKey, 256);
if(err != 0) return err;
```

## 1.2 CBC

In CBC mode, we XOR the 128-bit first block of clear text with a 128-bit initialisation vector. Then we encrypt the result with AES. For each new block, we use the previous cipher text as the initialisation vector.

### 1.2.1 Encrypt or decrypt a file

To encrypt or decrypt a file with the AES-CBC algorithm, we use the following functions:

```
int AES_CBC_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char* cbcKey,
    unsigned char IV[16], unsigned int keySizeBits);

int AES_CBC_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char* cbcKey,
    unsigned char IV[16], unsigned int keySizeBits);
```

The inputs are:

- `char* inputFilePath`: the path to the file to encrypt or decrypt
- `char* outputFilePath`: the path to the output file containing the resulting encryption or decryption
- `unsigned char* cbcKey`: the symmetric key
- `unsigned char IV[16]`: the initialisation vector
- `unsigned int keySizeBits`: the key length, 128, 192 or 256 bits.

Listing 4: Example of how to encrypt and decrypt a file with AES-CBC

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* cbcKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";

err = AES_CBC_EncryptFile(message, cipher, cbcKey, IV, 256);
if(err != 0) return err;
err = AES_CBC_DecryptFile(cipher, decipher, cbcKey, IV, 256);
if(err != 0) return err;
```

### 1.2.2 Encrypt or decrypt a buffer

To encrypt or decrypt a buffer with the CBC mode, we use the following functions:

```
int AES_CBC_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned char* cbcKey,
    unsigned char IV[16], unsigned int keySizeBits)

int AES_CBC_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned char* cbcKey,
    unsigned char IV[16], unsigned int keySizeBits)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* cbcKey: the symmetric key
- unsigned char IV[16]: the initialisation vector
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 5: Example of how to encrypt and decrypt a buffer with AES-CBC

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";
message[10] = 't';

err = AES_CBC_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, cbcKey, IV, 256);
if(err != 0) return err;
```

```
err = AES_CBC_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    cbcKey, IV, 256);
if(err != 0) return err;
```

### 1.2.3 Encrypt or decrypt a string

To encrypt or decrypt a string with the CBC mode, we use the following functions:

```
int AES_CBC_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int* outputSizeBytes, unsigned char* cbcKey, unsigned char IV[16], unsigned
    int keySizeBits)

int AES_CBC_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* cbcKey, unsigned char IV[16], unsigned
    int keySizeBits)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* cbcKey: the symmetric key
- unsigned char IV[16]: the initialisation vector
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 6: Example of how to encrypt and decrypt a string with AES-CBC

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";

err = AES_CBC_EncryptString(message, cipher, &cipherLength, cbcKey, IV, 256);
if(err != 0) return err;
err = AES_CBC_DecryptString(cipher, cipherLength, decipher, cbcKey, IV, 256);
if(err != 0) return err;
```

## 1.3 OFB

In OFB mode, an initialisation vector is encrypted with AES, then XORed with the first block of clear text, to obtain the first block of cipher text. Then this encrypted initialisation vector is reused as the initialisation vector for the next block.

### 1.3.1 Encrypt or decrypt a file

To encrypt or decrypt a file with the AES-OFB algorithm, we use the following functions:

```
int AES_OFB_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char* ofbKey,
    unsigned char IV[16], unsigned int keySizeBits)

int AES_OFB_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char* ofbKey,
    unsigned char IV[16], unsigned int keySizeBits)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ofbKey: the symmetric key
- unsigned char IV[16]: the initialisation vector
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits.

Listing 7: Example of how to encrypt and decrypt a file with AES-OFB

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ofbKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";

err = AES_OFB_EncryptFile(message, cipher, ofbKey, IV, 256);
if(err != 0) return err;
err = AES_OFB_DecryptFile(cipher, decipher, ofbKey, IV, 256);
if(err != 0) return err;
```

### 1.3.2 Encrypt or decrypt a buffer

To encrypt or decrypt a buffer with the OFB mode, we use the following functions:

```
int AES_OFB_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned char* ofbKey,
    unsigned char IV[16], unsigned int keySizeBits)

int AES_OFB_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes,unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* ofbKey,unsigned char IV[16], unsigned int keySizeBits)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* ofbKey: the symmetric key
- unsigned char IV[16]: the initialisation vector
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 8: Example of how to encrypt and decrypt a buffer with AES-OFB

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";
message[10] = 't';

err = AES_OFB_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ofbKey, IV, 256);
if(err != 0) return err;
err = AES_OFB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ofbKey, IV, 256);
if(err != 0) return err;
```

### 1.3.3 Encrypt or decrypt a string

To encrypt or decrypt a string with the OFB mode, we use the following functions:

```
int AES_OFB_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* ofbKey, unsigned char IV[16], unsigned
    int keySizeBits)

int AES_OFB_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ofbKey, unsigned char IV[16], unsigned
    int keySizeBits)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer

- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* ofbKey: the symmetric key
- unsigned char IV[16]: the initialisation vector
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 9: Example of how to encrypt and decrypt a string with AES-OFB

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ofbKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";

err = AES_OFB_EncryptString(message, cipher, &cipherLength, ofbKey, IV, 256);
if(err != 0) return err;
err = AES_OFB_DecryptString(cipher, cipherLength, decipher, ofbKey, IV, 256);
if(err != 0) return err;
```

## 1.4 CTR

In CTR mode, we encrypt a counter, which is incremented for each block. Then each counter is XORed with a block of clear text to obtain a block of cipher text.

### 1.4.1 Encrypt or decrypt a file

To encrypt or decrypt a file with the AES-CTR algorithm, we use the following functions:

```
int AES_CTR_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char* ctrKey,
    unsigned char startCounter[16], unsigned int keySizeBits)

int AES_CTR_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char* ctrKey,
    unsigned char startCounter[16], unsigned int keySizeBits)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ctrKey: the symmetric key
- unsigned char startCounter [16]: the first element of the counter
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits.

Listing 10: Example of how to encrypt and decrypt a file with AES-CTR

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ctrKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* startCounter = "1000000000000000";

err = AES_CTR_EncryptFile(message, cipher, ctrKey, startCounter, 256);
if(err != 0) return err;
err = AES_CTR_DecryptFile(cipher, decipher, ctrKey, startCounter, 256);
if(err != 0) return err;
```

### 1.4.2 Encrypt or decrypt a buffer

To encrypt or decrypt a buffer with the CTR mode, we use the following functions:

```
int AES_CTR_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned char* ctrKey,
    unsigned char startCounter[16], unsigned int keySizeBits)

int AES_CTR_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned char* ctrKey,
    unsigned char startCounter[16], unsigned int keySizeBits)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* ctrKey: the symmetric key
- unsigned char startCounter[16]: the first element of the counter
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 11: Example of how to encrypt and decrypt a buffer with AES-CTR

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ctrKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* startCounter = "1000000000000000";
message[10] = 't';

err = AES_CTR_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ctrKey,
    startCounter, 256);
```

```
if(err != 0) return err;
err = AES_OFB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ctrKey, startCounter, 256);
if(err != 0) return err;
```

### 1.4.3 Encrypt or decrypt a string

To encrypt or decrypt a string with the CTR mode, we use the following functions:

```
int AES_CTR_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int* outputSizeBytes, unsigned char* ctrKey, unsigned char
    startCounter[16], unsigned int keySizeBits)

int AES_CTR_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ctrKey, unsigned char startCounter[16],
    unsigned int keySizeBits)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* ctrKey: the symmetric key
- unsigned char startCounter [16]: the first element of the counter
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits

Listing 12: Example of how to encrypt and decrypt a string with AES-CTR

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ctrKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* startCounter = "1000000000000000";

err = AES_CTR_EncryptString(message, cipher, &cipherLength, ctrKey, startCounter, 256);
if(err != 0) return err;
err = AES_CTR_DecryptString(cipher, cipherLength, decipher, ctrKey, startCounter, 256);
if(err != 0) return err;
```

## 1.5 GCM

The GCM mode is a mode designed to ensure both the integrity, the authenticity and the confidentiality of the data. It is a variation of the CTR mode. It uses a hash function to ensure the authenticity of the confidential data, and can also ensure the authentication of additional data. It does not use padding.

### 1.5.1 Encrypt or decrypt a file

To encrypt or decrypt a file with the AES-GCM algorithm, we use the following functions:

```
int AES_GCM_EncryptFile(char* inputFilePath, char *additionalDataFilePath, char*
    outputFilePath, unsigned char* gcmKey, unsigned int keySizeBits, unsigned char *IV,
    unsigned int IVSizeBits, char* tagFilePath, unsigned int tagSizeBits)

int AES_GCM_DecryptFile(char* inputFilePath, char *additionalDataFilePath, char*
    outputFilePath, unsigned char* gcmKey, unsigned int keySizeBits, unsigned char *IV,
    unsigned int IVSizeBits, char* tagFilePath, unsigned int tagSizeBits)
```

The inputs are:

- `char* inputFilePath`: the path to the file to encrypt or decrypt
- `char* additionalDataFilePath`: the path to the file containing some additional data that we need to authenticate without encrypt them
- `char* outputFilePath`: the path to the output file containing the resulting encryption or decryption
- `unsigned char* gcmKey`: the symmetric key
- `unsigned int keySizeBits`: the key length, 128, 192 or 256 bits.
- `unsigned char *IV`: the initialisation vector with variable length
- `unsigned int IVSizeBits`: the number of bits of the initialisation vector (the recommended number of bits is 96)
- `char* tagFilePath`: the path to the file where we will write or read the authentication tag
- `unsigned int tagSizeBits`: the number of bits of the tag. It must be $\leqslant 128$.

Listing 13: Example of how to encrypt and decrypt a file with AES-GCM

```
int err;
char* message = "message.txt";
char* addData = "data.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
char *tag = "tag.txt";
unsigned char* gcmKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";

err = AES_GCM_EncryptFile(message, addData, cipher, gcmKey, 256, IV, 128, tag, 128);
if(err != 0) return err;
err = AES_GCM_DecryptFile(cipher, addData, decipher, gcmKey, 256, IV, 128, tag, 128);
if(err != 0) return err;
```

### 1.5.2 Encrypt or decrypt a buffer

To encrypt or decrypt a buffer with the GCM mode, we use the following functions:

```
int AES_GCM_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char *additionalDataBuffer, unsigned char* outputBuffer, unsigned char*
    gcmKey, unsigned int keySizeBits, unsigned char *IV, unsigned int IVSizeBits,
    unsigned char* tagBuffer, unsigned int tagSizeBits)

int AES_GCM_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char *additionalDataBuffer, unsigned char* outputBuffer, unsigned char*
    gcmKey, unsigned int keySizeBits, unsigned char *IV, unsigned int IVSizeBits,
    unsigned char *tagBuffer, unsigned int tagSizeBits)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char *additionalDataBuffer: the additional data to authenticate without encrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned char* gcmKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits
- unsigned char *IV: the initialisation vector with variable length
- unsigned int IVSizeBits: the number of bits of the initialisation vector (the recommended number is 96)
- unsigned char* tagBuffer: the authentication tag with variable length
- unsigned int tagSizeBits: the number of bits of the authentication tag. It must be $\leqslant 128$.

Listing 14: Example of how to encrypt and decrypt a buffer with AES-GCM

```
int err;
unsigned char message[20];
unsigned char *addData = ".additional_data.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char tag[16];
unsigned char* gcmKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";
message[10] = 't';

err = AES_GCM_EncryptBinaryBuffer(message, 20, addData, cipher, gcmKey, 256, IV, 128,
    tag, 128);
if(err != 0) return err;
err = AES_GCM_DecryptBinaryBuffer(cipher, 20, decipher, gcmKey, 256, IV, 128, tag, 128);
if(err != 0) return err;
```

### 1.5.3 Encrypt or decrypt a string

To encrypt or decrypt a string with the GCM mode, we use the following functions:

```
int AES_GCM_EncryptString(unsigned char* inputString, unsigned char *
    additionalDataBuffer, unsigned char* outputBuffer, unsigned char* gcmKey, unsigned
    int keySizeBits, unsigned char *IV, unsigned int IVSizeBits, unsigned char*
    tagBuffer, unsigned int tagSizeBits)

int AES_GCM_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char *additionalDataBuffer, unsigned char* outputString, unsigned char*
    gcmKey, unsigned int keySizeBits, unsigned char *IV, unsigned int IVSizeBits,
    unsigned char *tagBuffer, unsigned int tagSizeBits)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* additionalDataBuffer: the additional data to authenticate without encrypt them
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* gcmKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits
- unsigned char* IV: the initialisation vector with variable length
- unsigned int IVSizeBits: the number of bits of the initialisation vector
- unsigned char* tagBuffer: the authentication tag with variable length
- unsigned int tagSizeBits: the number of bits of the authentication tag. It must be $\leqslant 128$.

Listing 15: Example of how to encrypt and decrypt a string with AES-GCM

```
int err;
unsigned char *message = ".a_message_of_test.";
unsigned char *addData = ".additional_data.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char tag[16];
unsigned char* gcmKey = "0123456789ABCDEF0123456789ABCDEF";
unsigned char* IV = "88881111FFFFAAAA";

err = AES_GCM_EncryptString(message, addData, cipher, gcmKey, 256, IV, 128, tag, 128);
if(err != 0) return err;
err = AES_GCM_DecryptString(cipher, 20, addData, decipher, gcmKey, 256, IV, 128, tag,
    128);
if(err != 0) return err;
```

## 1.6 MAC

The MAC mode is a mode designed to guarantee the integrity of a message. It uses the CBC mode with a zero initialisation vector to product a cipher text which is then reduced to have a tag.

### 1.6.1 Generate or verify a tag from a file

To generate or verify a tag from a file with the AES-MAC algorithm, we use the following functions:

```
int AES_MAC_GenerationFile(char* inputFilePath, unsigned char* macKey, unsigned int
    keySizeBits, unsigned char* tagBuffer, unsigned int tagSizeBits)

int AES_MAC_VerificationFile(char* inputFilePath, unsigned char* macKey, unsigned int
    keySizeBits, unsigned char* tagBuffer, unsigned int tagSizeBits)
```

The inputs are:

- char* inputFilePath: the path to the file to authenticate
- unsigned char* macKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits.
- unsigned char *tagBuffer: the authentication tag
- unsigned int tagSizeBits: the number of bits of the desired tag. It must be $\leqslant 128$.

Listing 16: Example of how to generate and verify a tag from a file with AES-MAC

```
int err;
char* message = "message.txt";
unsigned char tag[16];
unsigned char* macKey = "0123456789ABCDEF0123456789ABCDEF";

err = AES_MAC_GenerationFile(message, macKey, 256, tag, 128);
if(err != 0) return err;
err = AES_MAC_VerificationFile(message, macKey, 256, tag, 128);
if(err != 0) return err;
```

### 1.6.2 Generate or verify a tag from a buffer

To generate or verify a tag from a buffer with the MAC mode, we use the following functions:

```
int AES_MAC_GenerationBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* macKey, unsigned int keySizeBits, unsigned char*
    tagBuffer, unsigned int tagSizeBits)

int AES_MAC_VerificationBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* macKey, unsigned int keySizeBits, unsigned char*
    tagBuffer, unsigned int tagSizeBits)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to authenticate
- unsigned int inputSizeBytes: the length in bytes of the buffer to authenticate
- unsigned char* macKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits
- unsigned char* tagBuffer: the authentication tag with variable length
- unsigned int tagSizeBits: the number of bits of the authentication tag. It must be $\leqslant 128$.

Listing 17: Example of how to generate and verify a tag from a buffer with AES-MAC

```
int err;
unsigned char message[20];
unsigned char tag[16];
unsigned char* macKey = "0123456789ABCDEF0123456789ABCDEF";
message[10] = 't';

err = AES_MAC_GenerationBinaryBuffer(message, 20, macKey, 256, tag, 128);
if(err != 0) return err;
err = AES_MAC_VerificationBinaryBuffer(message, 20, macKey, 256, tag, 128);
if(err != 0) return err;
```

### 1.6.3 Generate or verify a tag from a string

To generate or verify a tag from a string with the MAC mode, we use the following functions:

```
int AES_MAC_GenerationString(unsigned char* inputString, unsigned char* macKey, unsigned
    int keySizeBits, unsigned char* tagBuffer, unsigned int tagSizeBits)

int AES_MAC_VerificationString(unsigned char* inputString, unsigned char* macKey,
    unsigned int keySizeBits, unsigned char* tagBuffer, unsigned int tagSizeBits)
```

The inputs are:

- unsigned char* inputString: the string to authenticate
- unsigned char* macKey: the symmetric key
- unsigned int keySizeBits: the key length, 128, 192 or 256 bits
- unsigned char* tagBuffer: the authentication tag with variable length
- unsigned int tagSizeBits: the number of bits of the authentication tag. It must be $\leqslant 128$.

Listing 18: Example of how to generate and verify a tag from a string with AES-MAC

```
int err;
unsigned char *message = ".a_message_of_test.";
unsigned char* macKey = "0123456789ABCDEF0123456789ABCDEF";

err = AES_MAC_GenerationString(message, macKey, 256, tag, 128);
if(err != 0) return err;
err = AES_MAC_VerificationString(message, macKey, 256, tag, 128);
if(err != 0) return err;
```

# 2 RSA

RSA is an asymmetric encryption algorithm and a signature algorithm, described in 1977 by Ronald Rivest, Adi Shamir et Leonard Adleman.

To encrypt some data with RSA, it is necessary to use the public key of the recipient and to decrypt, it is necessary to use the recipient's private key.

To sign some data with RSA, it is necessary to use the sender's private key, and the recipient verifies the signature with the public key of the sender.

The RSA algorithm is not secure in its initial form. In order to make it secure, OAEP has been developed as a padding scheme. Similarly for the signature, the secure form is called PSS. OAEP and PSS are described in the PKCS#1 v2.2.

We will use in our algorithm, RSA keys of length 2048, 3072 or 4096 bits.

## 2.1 Generate the key pair used in the encryption or the signature

To generate the key pair used int the encryption or the signature, we use the following function:

```
int RSA_GenerateKeys(unsigned int modulusSizeBits, unsigned char* modulus, unsigned char
    publicExponent[32], unsigned char* privateExponent)
```

The inputs are:

- unsigned int modulusSizeBits: the key length, 2048, 3072 or 4096 bits
- unsigned char* modulus: the RSA public key $n$ on $\frac{modulusSizeBits}{8}$ bytes
- unsigned char publicExponent[32]: the RSA public exponent $e$ on 32 bytes
- unsigned char* privateExponent: the RSA private exponent $d$ on $\frac{modulusSizeBits}{8}$ bytes

Listing 19: Example of how to generate a key pair with RSA

```
int err;
unsigned int modulusSizeBits = 2048;
unsigned char modulus[256];
unsigned char publicExponent[32];
unsigned char privateExponent[256];

err = RSA_GenerateKeys(modulusSizeBits, modulus, publicExponent, privateExponent);
if(err != 0) return err;
```

## 2.2 Encrypt or decrypt a buffer with RSA-OAEP

To encrypt or decrypt a buffer with RSA-OAEP, we use the following functions:

```
int RSA_OAEPencryption(char* inputBuffer, unsigned int inputSizeBytes, char* labelBuffer,
    char* outputBuffer, unsigned char* modulus, unsigned char publicExponent[32],
    unsigned int modulusSizeBits)

int RSA_OAEPdecryption(char* inputBuffer, char* labelBuffer, char* outputBuffer, unsigned
    char* modulus, unsigned char* privateExponent, unsigned int modulusSizeBits)
```

The inputs are:

- char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length of the buffer to encrypt (must be $\leqslant \frac{modulusSizeBits}{8}$)
- char* labelBuffer: an optional label of length 32 bytes, equal to 0 by default
- char* outputBuffer: the output buffer, result of the encryption or decryption
- unsigned char* modulus: the RSA public key $n$ on $\frac{modulusSizeBits}{8}$ bytes
- unsigned char publicExponent[32]: the RSA public exponent $e$ on 32 bytes
- unsigned char* privateExponent: the RSA private exponent $d$ on on $\frac{modulusSizeBits}{8}$ bytes
- unsigned int modulusSizeBits: the length of the RSA key, so 2048, 3072 or 4096 bits

Listing 20: Example of how to encrypt and decrypt with RSA

```
int err;
unsigned int modulusSizeBits = 2048;
unsigned char modulus[256];
unsigned char publicExponent[32];
unsigned char privateExponent[256];
char *message = ".a_message_of_test.";
char cipher[256];
char decipher[256];
char label[32];

err = RSA_GenerateKeys(modulusSizeBits, modulus, publicExponent, privateExponent);
if(err != 0) return err;
err = RSA_OAEPencryption(message, 19, label, cipher, modulus, publicExponent,
    modulusSizeBits);
if(err != 0) return err;
err = RSA_OAEPdecryption(cipher, label, decipher, modulus, privateExponent,
    modulusSizeBits);
if(err != 0) return err;
```

## 2.3 Sign or verify a signature with RSA-PSS

To sign or verify the signature of a buffer with RSA-PSS, we use the following functions:

```
int RSA_PSSSignature(char* inputBuffer, unsigned int inputSizeBytes, unsigned char*
    signature, unsigned char* modulus, unsigned char* privateExponent, unsigned int
    modulusSizeBits)

int RSA_PSSVerification(char* inputBuffer, unsigned int inputSizeBytes, unsigned char*
    signature, unsigned char* modulus, unsigned char publicExponent[32], unsigned int
    modulusSizeBits)
```

The inputs are:

- char* inputBuffer: the buffer to sign or that we need to verify the signature

- `unsigned int` inputSizeBytes: the length of the buffer to sign or verify
- `unsigned char*` signature: the signature that we need to generate or verify (with length $\frac{modulusSizeBits}{8}$ bytes)
- `unsigned char*` modulus: the RSA public key $n$ on $\frac{modulusSizeBits}{8}$ bytes
- `unsigned char` publicExponent[32]: the RSA public exponent $e$ on 32 bytes
- `unsigned char*` privateExponent: the RSA private exponent $d$ on on $\frac{modulusSizeBits}{8}$ bytes
- `unsigned int` modulusSizeBits: the length of the RSA key, so 2048, 3072 or 4096 bits

Listing 21: Example of how to sign and verify a signature with RSA

```
int err;
unsigned int modulusSizeBits = 2048;
unsigned char modulus[256];
unsigned char publicExponent[32];
unsigned char privateExponent[256];
char *message = ".a_message_of_test.";
char signature[256];

err = RSA_GenerateKeys(modulusSizeBits, modulus, publicExponent, privateExponent);
if(err != 0) return err;
err = RSA_PSSSignature(message, 19, signature, modulus, privateExponent,
    modulusSizeBits);
if(err != 0) return err;
err = RSA_PSSVerification(message, 19, signature, modulus, publicExponent,
    modulusSizeBits);
if(err != 0) return err;
```

# 3 EdDSA

EdDSA is a digital signature algorithm using Edwards elliptic curves. It has been developed by a team directed by Daniel J. Bernstein. Two algorithms are implemented in this library, EdDSA25519, whose public key is encoded on 256 bits and EdDSA511187, whose public key is encoded on 512 bits.

## 3.1 EdDSA25519

### 3.1.1 Keys generation

To generate the key pair to sign, we use the following function:

```
int EdDSA_Curve25519_GenerateKeys(unsigned char publicKey[32], unsigned char
    privateKey[32])
```

The inputs are:

- unsigned char publicKey[32]: the public key on 32 bytes
- unsigned char privateKey[32]: the private key on 32 bytes

Listing 22: Example of how to generate a key pair with EdDSA on the Curve 25519

```
int err;
unsigned char publicKey[32];
unsigned char privateKey[32];
err = EdDSA_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
```

### 3.1.2 Signature

To sign a buffer with EdDSA25519, we use the following function:

```
int EdDSA_Curve25519_Signature(char* inputBuffer, unsigned int inputSizeBytes, unsigned
    char privateKeySender[32], unsigned char publicKeySender[32], unsigned char
    signature[64])
```

The inputs are:

- char* inputBuffer: the buffer to sign
- unsigned int inputSizeBytes: the length in bytes of the buffer to sign
- unsigned char privateKeySender[32]: the sender's private key
- unsigned char publicKeySender[32]: the sender's public key
- unsigned char signature[64]: the resulting signature

Listing 23: Example of how to sign a message with EdDSA on the Curve 25519

```
int err;
unsigned char publicKey[32];
unsigned char privateKey[32];
char *message = ".a_message_of_test.";
char signature[64];

err = EdDSA_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = EdDSA_Curve25519_Signature(message, 19, privateKey, publicKey, signature);
if(err != 0) return err;
```

### 3.1.3 Signature verification

To verify a signature with EdDSA25519, we use the following function:

```
int EdDSA_Curve25519_Verification(char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char publicKeySender[32], unsigned char signature[64]);
```

The inputs are:

- char* inputBuffer: the buffer that we need to verify the signature
- unsigned int inputSizeBytes: the length in bytes of the buffer
- unsigned char publicKeySender[32]: the sender's public key
- unsigned char signature [64]: the signature to verify

Listing 24: Example of how to verify a signature with EdDSA on the Curve 25519

```
int err;
unsigned char publicKey[32];
unsigned char privateKey[32];
char *message = ".a_message_of_test.";
char signature[64];

err = EdDSA_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = EdDSA_Curve25519_Verification(message, 19, publicKey, signature);
if(err != 0) return err;
```

## 3.2 EdDSA511187

### 3.2.1 Keys generation

To generate the key pair to sign, we use the following function:

```
int EdDSA_Curve511187_GenerateKeys(unsigned char publicKey[64], unsigned char
    privateKey[64])
```

The inputs are:

- unsigned char publicKey[64]: the public key on 64 bytes
- unsigned char privateKey[64]: the private key on 64 bytes

Listing 25: Example of how to generate a key pair with EdDSA on the Curve 511187

```
int err;
unsigned char publicKey[64];
unsigned char privateKey[64];
err = EdDSA_Curve511187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
```

### 3.2.2 Signature

To sign a buffer with EdDSA511187, we use the following function:

```
int EdDSA_Curve511187_Signature(char* inputBuffer, unsigned int inputSizeBytes, unsigned
    char privateKeySender[64], unsigned char publicKeySender[64], unsigned char
    signature[128])
```

The inputs are:

- char* inputBuffer: the buffer to sign
- unsigned int inputSizeBytes: the length in bytes of the buffer to sign
- unsigned char privateKeySender[64]: the sender's private key
- unsigned char publicKeySender[64]: the sender's public key
- unsigned char signature[128]: the resulting signature

Listing 26: Example of how to sign a message with EdDSA on the Curve 511187

```
int err;
unsigned char publicKey[64];
unsigned char privateKey[64];
char *message = ".a_message_of_test.";
char signature[128];

err = EdDSA_Curve511187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = EdDSA_Curve511187_Signature(message, 19, privateKey, publicKey, signature);
if(err != 0) return err;
```

### 3.2.3 Signature verification

To verify a signature with EdDSA511187, we use the following function:

```
int EdDSA_Curve511187_Verification(char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char publicKeySender[64], unsigned char signature[128]);
```

The inputs are:

- char* inputBuffer: the buffer that we need to verify the signature
- unsigned int inputSizeBytes: the length in bytes of the buffer
- unsigned char publicKeySender[64]: the sender's public key
- unsigned char signature[128]: the signature to verify

Listing 27: Example of how to verify a signature with EdDSA on the Curve 511187

```
int err;
unsigned char publicKey[64];
unsigned char privateKey[64];
char *message = ".a_message_of_test.";
char signature[128];

err = EdDSA_Curve51187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = EdDSA_Curve511187_Verification(message, 19, publicKey, signature);
if(err != 0) return err;
```

# 4 ECIES

Elliptic Curve Integrated Encryption Scheme (ECIES) is an asymmetric encryption scheme using elliptic curves. In this library, we have used Edwards curves, Curve25519 and Curve511187 because they ensure a good security level and allow us to reuse a part of the algorithms already implemented for EdDSA25119 and EdDSA511187.

## 4.1 ECIES on the curve Curve25519

### 4.1.1 Key generation

To generate the key pair to encrypt or decrypt, we use the following function:

```
int ECIES_Curve25519_GenerateKeys(unsigned char publicKey[32], unsigned char
    privateKey[32])
```

The inputs are:

- unsigned char publicKey[32]: the public key on 32 bytes
- unsigned char privateKey[32]: the private key on 32 bytes

Listing 28: Example of how to generate a key pair with ECIES on the Curve 25519

```
int err;
unsigned char publicKey[32];
unsigned char privateKey[32];

err = ECIES_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
```

### 4.1.2 Encrypt or decrypt a file with ECIES25519

To encrypt or decrypt a file with ECIES25519, we use the following functions:

```
int ECIES_Curve25519_EncryptionFile(char* inputFilePath, unsigned char
    publicKeyRecipient[32], char* outputFilePath)

int ECIES_Curve25519_DecryptionFile(char* inputFilePath, unsigned char
    privateKeyRecipient[32], char* outputFilePath)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- unsigned char publicKeyRecipient[32]: the recipient's public key to encrypt
- unsigned char privateKeyRecipient[32]: the recipient's private key to decrypt
- char* outputFilePath: the path to the encrypted or decrypted file

Listing 29: Example of how to encrypt and decrypt a file with ECIES on the Curve 25519

```
int err;
unsigned char publicKey[32];
unsigned char privateKey[32];
char *messagePath = "message.txt";
char *cipherPath = "cipher.txt";
char *decipherPath = "decipher.txt";

err = ECIES_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = ECIES_Curve25519_EncryptionFile(messagePath, publicKey, cipherPath);
if(err != 0) return err;
err = ECIES_Curve25519_DecryptionFile(cipherPath, privateKey, decipherPath);
if(err != 0) return err;
```

### 4.1.3 Encrypt or decrypt a binary buffer with ECIES25519

To encrypt or decrypt a binary buffer with ECIES25519, we use the following functions:

```
int ECIES_Curve25519_EncryptionBinaryBuffer(char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char publicKeyRecipient[32], char* outputBuffer, unsigned
    int* outputSizeBytes)

int ECIES_Curve25519_DecryptionBinaryBuffer(char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char privateKeyRecipient[32], char* outputBuffer, unsigned
    int *outputSizeBytes)
```

The inputs are:

- char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes
- unsigned char publicKeyRecipient[32]: the recipient's public key to encrypt
- unsigned char privateKeyRecipient[32]: the recipient's private key to decrypt
- char* outputBuffer: the encrypted or deencrypted buffer
- unsigned int *outputSizeBytes: the length in bytes of the encrypted or deencrypted buffer

Listing 30: Example of how to encrypt and decrypt a buffer with ECIES on the Curve 25519

```
int err, decipherLength, cipherLength;
unsigned char publicKey[32];
unsigned char privateKey[32];
char *message = ".a_message_of_test.";
char cipher[80];
char decipher[20];

err = ECIES_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = ECIES_Curve25519_EncryptionBinaryBuffer(message,19, publicKey, cipher,
    &cipherLength);
```

```
if(err != 0) return err;
err = ECIES_Curve25519_DecryptionBinaryBuffer(cipher, cipherLength, privateKey,
    decipher, &decipherLength);
if(err != 0) return err;
```

### 4.1.4 Encrypt or decrypt a string with ECIES25519

To encrypt or decrypt a string with ECIES25519, we use the following functions:

```
int ECIES_Curve25519_EncryptionString(char* inputString, unsigned char
    publicKeyRecipient[32], char* outputBuffer, unsigned int* outputSizeBytes)

int ECIES_Curve25519_DecryptionString(char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char privateKeyRecipient[32], char* outputString)
```

The inputs are:

- char* inputString: the string to encrypt
- char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned char publicKeyRecipient[32]: the recipient's public key to encrypt
- unsigned char privateKeyRecipient[32]: the recipient's private key to decrypt
- char* outputBuffer: the encrypted buffer
- char* outputString: the deencrypted string
- unsigned int *outputSizeBytes: the length in bytes of the encrypted buffer

Listing 31: Example of how to encrypt and decrypt a string with ECIES on the Curve 25519

```
int err, cipherLength;
unsigned char publicKey[32];
unsigned char privateKey[32];
char *message = ".a_message_of_test.";
char cipher[80];
char decipher[20];

err = ECIES_Curve25519_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = ECIES_Curve25519_EncryptionString(message, publicKey, cipher, &cipherLength);
if(err != 0) return err;
err = ECIES_Curve25519_DecryptionString(cipher, cipherLength, privateKey, decipher);
if(err != 0) return err;
```

## 4.2 ECIES on the curve Curve511187

### 4.2.1 Key generation

To generate the key pair to encrypt or decrypt, we use the following function:

```
int ECIES_Curve511187_GenerateKeys(unsigned char publicKey[64], unsigned char
    privateKey[64])
```

The inputs are:

- unsigned char publicKey[64]: the public key on 64 bytes
- unsigned char privateKey[64]: the private key on 64 bytes

Listing 32: Example of how to generate a key pair with ECIES on the Curve 511187

```
int err;
unsigned char publicKey[64];
unsigned char privateKey[64];

err = ECIES_Curve511187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
```

### 4.2.2 Encrypt or decrypt a file with ECIES511187

To encrypt or decrypt a file with ECIES511187, we use the following functions:

```
int ECIES_Curve511187_EncryptionFile(char* inputFilePath, unsigned char
    publicKeyRecipient[64], char* outputFilePath)

int ECIES_Curve511187_DecryptionFile(char* inputFilePath, unsigned char
    privateKeyRecipient[64], char* outputFilePath)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- unsigned char publicKeyRecipient[64]: the recipient's public key to encrypt
- unsigned char privateKeyRecipient[64]: the recipient's private key to decrypt
- char* outputFilePath: the path to the encrypted or deencrypted file

Listing 33: Example of how to encrypt and decrypt a file with ECIES on the Curve 511187

```
int err;
unsigned char publicKey[64];
unsigned char privateKey[64];
char *messagePath = "message.txt";
char *cipherPath = "cipher.txt";
char *decipherPath = "decipher.txt";
```

```
err = ECIES_Curve511187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = ECIES_Curve511187_EncryptionFile(messagePath, publicKey, cipherPath);
if(err != 0) return err;
err = ECIES_Curve511187_DecryptionFile(cipherPath, privateKey, decipherPath);
if(err != 0) return err;
```

### 4.2.3 Encrypt or decrypt a binary buffer with ECIES511187

To encrypt or decrypt a binary buffer with ECIES511187, we use the following functions:

```
int ECIES_Curve511187_EncryptionBinaryBuffer(char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char publicKeyRecipient[64], char* outputBuffer, unsigned
    int* outputSizeBytes)

int ECIES_Curve511187_DecryptionBinaryBuffer(char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char privateKeyRecipient[64], char* outputBuffer, unsigned
    int *outputSizeBytes)
```

The inputs are:

- char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes
- unsigned char publicKeyRecipient[64]: the recipient's public key to encrypt
- unsigned char privateKeyRecipient[64]: the recipient's private key to decrypt
- char* outputBuffer: the encrypted or deencrypted buffer
- unsigned int *outputSizeBytes: the length in bytes of the encrypted or deencrypted buffer

Listing 34: Example of how to encrypt and decrypt a buffer with ECIES on the Curve 511187

```
int err, decipherLength, cipherLength;
unsigned char publicKey[64];
unsigned char privateKey[64];
char *message = ".a_message_of_test.";
char cipher[112];
char decipher[20];

err = ECIES_Curve511187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = ECIES_Curve511187_EncryptionBinaryBuffer(message,19, publicKey, cipher,
    &cipherLength);
if(err != 0) return err;
err = ECIES_Curve511187_DecryptionBinaryBuffer(cipher, cipherLength, privateKey,
    decipher, &decipherLength);
if(err != 0) return err;
```

### 4.2.4 Encrypt or decrypt a string with ECIES511187

To encrypt or decrypt a string with ECIES511187, we use the following functions:

```
int ECIES_Curve511187_EncryptionString(char* inputString, unsigned char
    publicKeyRecipient[64], char* outputBuffer, unsigned int* outputSizeBytes)

int ECIES_Curve511187_DecryptionString(char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char privateKeyRecipient[64], char* outputString)
```

The inputs are:

- char* inputString: the string to encrypt
- char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned char publicKeyRecipient[64]: the recipient's public key to encrypt
- unsigned char privateKeyRecipient[64]: the recipient's private key to decrypt
- char* outputBuffer: the encrypted buffer
- char* outputString: the deencrypted string
- unsigned int *outputSizeBytes: the length in bytes of the encrypted buffer

Listing 35: Example of how to encrypt and decrypt a string with ECIES on the Curve 511187

```
int err, cipherLength;
unsigned char publicKey[64];
unsigned char privateKey[64];
char *message = ".a_message_of_test.";
char cipher[112];
char decipher[20];

err = ECIES_Curve511187_GenerateKeys(publicKey, privateKey);
if(err != 0) return err;
err = ECIES_Curve511187_EncryptionString(message, publicKey, cipher, &cipherLength);
if(err != 0) return err;
err = ECIES_Curve511187_DecryptionString(cipher, cipherLength, privateKey, decipher);
if(err != 0) return err;
```

# 5 SALSA

Salsa20 is a stream encryption algorithm proposed by Daniel Bernstein.

## 5.1 Encryption

To encrypt a buffer with Salsa20, we use the following function:

```
int SALSA_Encryption(char* inputBuffer, unsigned int inputSizeBytes, char* salsaKey,
    unsigned int keySizeBytes, char IV[8], char *outputBuffer)
```

The inputs are:

- char* inputBuffer: the buffer to encrypt
- unsigned int inputSizeBytes: the length of the buffer in bytes
- char* salsaKey: the key
- unsigned int keySizeBytes: the length of the key in bytes, equal to 16 or 32 bytes
- char IV[8]: the initialisation vector on 8 bytes
- char* outputBuffer: the output buffer

Listing 36: Example of how to encrypt a message with Salsa

```
int err;
char *message = ".a_message_of_test.";
char cipher[20];
char* salsaKey =  "0123456789ABCDEF";
char *IV = "11111111";

err = SALSA_Encryption(message, 19, salsaKey, 16, IV, cipher);
if(err != 0) return err;
```

## 5.2 Decryption

To decrypt a buffer with Salsa20, we use the following function:

```
int SALSA_Decryption(char* inputBuffer, unsigned int inputSizeBytes, char* salsaKey,
    unsigned int keySizeBytes, char IV[8], char *outputBuffer)
```

The inputs are:

- char* inputBuffer: the buffer to encrypt
- unsigned int inputSizeBytes: the length of the buffer in bytes
- char* salsaKey: the key

- unsigned int keySizeBytes: the length of the key in bytes, equal to 16 or 32 bytes
- char IV[8]: the initialisation vector on 8 bytes
- char* outputBuffer: the output buffer

Listing 37: Example of how to encrypt and decrypt a message with Salsa

```
int err;
char *message = ".a_message_of_test.";
char cipher[20];
char decipher[20];
char* salsaKey =  "0123456789ABCDEF";
char *IV = "11111111";
err = SALSA_Encryption(message, 19, salsaKey, 16, IV, cipher);
if(err != 0) return err;
err = SALSA_Decryption(cipher, 19, salsaKey, 16, IV, decipher);
if(err != 0) return err;
```

# 6 SHA-2

SHA-2 (Secure Hash Algorithm) is a family of hash functions, that have been designed by the National Security Agency (NSA) of the USA, on the model of the SHA-1 and SHA-0 functions. The algorithms of the SHA-2 family, SHA-256, SHA-384 and SHA-512 are described and published along with SHA-1 in the FIPS 180-2 (Secure Hash Standard). In this library, only SHA-256 and SHA-512 have been implemented. SHA-2 is described in the FIPS PUB 180-4.

## 6.1 Hash a file with SHA-2

To hash a file with SHA-2, we use the following function:

```
int SHA2_FileHash(char* inputFilePath, const unsigned int hashSizeBits, char* fileHash)
```

The inputs are:

- char* inputFilePath: the path to the file to hash
- const unsigned int hashSizeBits: the length in bits of the hash, 256 or 512 bits
- char* fileHash: the buffer containing the hash

Listing 38: Example of how to hash a file with SHA-2

```
int err;
char *messagePath = "message.txt";
char hash[32];
err = SHA2_FileHash(messagePath, 256, hash);
if(err != 0) return err;
```

## 6.2 Hash a buffer with SHA-2

To hash a buffer with SHA-2, we use the following function:

```
int SHA2_BufferHash(char *inputBuffer, unsigned int inputSizeBytes, const unsigned int
    hashSizeBits, char* bufferHash)
```

The inputs are:

- char *inputBuffer: the buffer to hash
- unsigned int inputSizeBytes: the length in bytes of the buffer to hash
- const unsigned int hashSizeBits: the length in bits of the hash, 256 or 512 bits
- char* bufferHash: the buffer containing the hash

Listing 39: Example of how to hash a buffer with SHA-2

```
int err;
char *message = ".a_message_of_test.";
char hash[32];
err = SHA2_BufferHash(message, 19, 256, hash);
if(err != 0) return err;
```

# 7 SHA-3

SHA-3 comes from the NIST hash function competition which elected the algorithm Keccak on October 2, 2012. It is not intended to replace SHA-2 but to provide an alternative following the possibilities of attacks on the standards MD5, SHA-0 and SHA-1. This library allows us to hash with the SHA-3 standard algorithm but also with the SHA-3 XOF (Extendable-Output Function) algorithm which allows us to have output of variable length. SHA-3 is described in the FIPS PUB 202.

## 7.1 Hash a file with SHA-3

To hash a file with SHA-3, we use the following function:

```
int SHA3_FileHash(char* inputFilePath, const unsigned int hashSizeBits, char* fileHash)
```

The inputs are:

- char* inputFilePath: the path to the file to hash
- const unsigned int hashSizeBits: the number of bits of the hash, 224, 256, 384 or 512 bits
- char* fileHash: the buffer containing the hash of the file

Listing 40: Example of how to hash a file with SHA-3

```
int err;
char *messagePath = "message.txt";
char hash[32];

err = SHA3_FileHash(messagePath, 256, hash);
if(err != 0) return err;
```

## 7.2 Hash a buffer with SHA-3

To hash a buffer with SHA-3, we use the following function:

```
int SHA3_BufferHash(unsigned char* inputBuffer, unsigned int inputSizeBytes, const
    unsigned int hashSizeBits, char* bufferHash)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to hash
- unsigned int inputSizeBytes: the length of the buffer in bytes
- const unsigned int hashSizeBits: the number of bits of the hash, 224, 256, 384 or 512 bits
- char* bufferHash: the buffer containing the hash

Listing 41: Example of how to hash a buffer with SHA-3

```
int err;
char *message = ".a_message_of_test.";
char hash[32];

err = SHA3_BufferHash(message, 19, 256, hash);
if(err != 0) return err;
```

## 7.3 Hash a file with SHA-3 XOF

To hash a file with SHA-3 XOF, we use the following function:

```
int SHA3_XOF_FileHash(char* inputFilePath, const unsigned int hashSizeBits, unsigned int
    version, char* fileHash)
```

The inputs are:

- char* inputFilePath: the path to the file to hash
- const unsigned int hashSizeBits: the number of bits of the hash, 224, 256, 384 or 512 bits
- unsigned int version: the capacity of the sponge function, i.e. 256 or 512
- char* fileHash: the buffer containing the hash of the file

Listing 42: Example of how to hash a file with SHA-3 XOF

```
int err;
char *messagePath = "message.txt";
char hash[128];

err = SHA3_XOF_FileHash(messagePath, 1024, 256, hash);
if(err != 0) return err;
```

## 7.4 Hash a buffer with SHA-3 XOF

To hash a buffer with SHA-3 XOF, we use the following function:

```
int SHA3_XOF_BufferHash(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned int hashSizeBits, unsigned int version, char* bufferHash)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to hash
- unsigned int inputSizeBytes: the length of the buffer in bytes
- const unsigned int hashSizeBits: the number of bits of the hash, 224, 256, 384 or 512 bits
- unsigned int version: the capacity of the sponge function, i.e. 256 or 512

- char* bufferHash: the buffer containing the hash

Listing 43: Example of how to hash a buffer with SHA-3 XOF

```
int err;
char *message = ".a_message_of_test.";
char hash[128];

err = SHA3_XOF_BufferHash(message, 19, 1024, 256, hash);
if(err != 0) return err;
```

# 8 SPECK

Speck is a family of lightweight block ciphers publicly released by the National Security Agency (NSA) in June 2013. Speck has been optimized for performance in software implementations. Speck is an add-rotate-xor (ARX) cipher.

Speck supports a variety of block and key sizes. A block is always two words, but the words may be 16, 24, 32, 48 or 64 bits in size. The corresponding key is 2, 3 or 4 words. The round function consists in two rotations, adding the right word to the left word, xoring the key into the left word, then and xoring the left word to the right word. To encrypt a message with a large number of blocks, we will use the following modes (like AES):

- ECB (Electronic Code Book)
- CBC (Cipher Block Chaining)
- OFB (Output Feedback)

To optimize the most common mode use, we have developed a function SPECK32 where wordSizeBits is equal to 32.

## 8.1 For all sizes of word

### 8.1.1 ECB

ECB is the simplest mode. The message to encrypt is divided into blocks of $keySizeWords \times wordSizeBits$ bits and each block is encrypted separately with the same key. We have 3 functions to:

- encrypt or decrypt a file
- encrypt or decrypt a buffer
- encrypt or decrypt a string

**Encrypt or decrypt a file** To encrypt or decrypt a file with the SPECK-ECB algorithm, we use the following functions:

```
int SPECK_ECB_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ecbKey, unsigned int wordSizeBits, unsigned int keySizeWords)

int SPECK_ECB_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ecbKey, unsigned int wordSizeBits, unsigned int keySizeWords)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ecbKey: the symmetric key
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits

- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 44: Example of how to encrypt or decrypt a file with SPECK-ECB

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ecbKey = "012345678";

err = SPECK_ECB_EncryptFile(message, cipher, ecbKey, 24, 3);
if(err != 0) return err;
err = SPECK_ECB_DecryptFile(cipher, decipher, ecbKey, 24, 3);
if(err != 0) return err;
```

**Encrypt or decrypt a buffer** To encrypt or decrypt a buffer with the ECB mode, we use the following functions:

```
int SPECK_ECB_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned
    char* ecbKey, unsigned int wordSizeBits, unsigned int keySizeWords)

int SPECK_ECB_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* ecbKey, unsigned int wordSizeBits, unsigned int keySizeWords)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* ecbKey: the symmetric key
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits
- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 45: Example of how to encrypt or decrypt a buffer with SPECK-ECB

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "012345678";
message[10] = 't';

err = SPECK_ECB_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ecbKey, 24, 3);
if(err != 0) return err;
```

```
err = SPECK_ECB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ecbKey, 24, 3);
if(err != 0) return err;
```

**Encrypt or decrypt a string** To encrypt or decrypt a string with the ECB mode, we use the following functions:

```
int SPECK_ECB_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* ecbKey, unsigned int wordSizeBits,
    unsigned int keySizeWords)

int SPECK_ECB_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ecbKey, unsigned int wordSizeBits,
    unsigned int keySizeWords)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* ecbKey: the symmetric key
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits
- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 46: Example of how to encrypt or decrypt a string with SPECK-ECB

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "012345678";

err = SPECK_ECB_EncryptString(message, cipher, &cipherLength, ecbKey, 24, 3);
if(err != 0) return err;
err = SPECK_ECB_DecryptString(cipher, cipherLength, decipher, ecbKey, 24, 3);
if(err != 0) return err;
```

### 8.1.2 CBC

In CBC mode, we XOR the first block of clear text with an initialisation vector. Then we encrypt the result with SPECK. For each new block, we use the previous cipher text as the initialisation vector.

**Encrypt or decrypt a file** To encrypt or decrypt a file with the SPECK-CBC algorithm, we use the following functions:

```
int SPECK_CBC_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    cbcKey, unsigned char *IV, unsigned int wordSizeBits, unsigned int keySizeWords);

int SPECK_CBC_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    cbcKey, unsigned char* IV, unsigned int wordSizeBits,  unsigned int keySizeWords);
```

The inputs are:

- `char* inputFilePath`: the path to the file to encrypt or decrypt
- `char* outputFilePath`: the path to the output file containing the resulting encryption or decryption
- `unsigned char* cbcKey`: the symmetric key
- `unsigned char *IV`: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- `unsigned int wordSizeBits`: the length of the words, 16, 24, 32, 48 or 64 bits
- `unsigned int keySizeWords`: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 47: Example of how to encrypt or decrypt a file with SPECK-CBC

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* cbcKey = "012345678";
unsigned char* IV = "012345";

err = SPECK_CBC_EncryptFile(message, cipher, cbcKey, IV, 24, 3);
if(err != 0) return err;
err = SPECK_CBC_DecryptFile(cipher, decipher, cbcKey, IV, 24, 3);
if(err != 0) return err;
```

**Encrypt or decrypt a buffer** To encrypt or decrypt a buffer with the CBC mode, we use the following functions:

```
int SPECK_CBC_EncryptBinaryBuffer(unsigned char* inputBuffer,unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned
    char* cbcKey, unsigned char* IV, unsigned int wordSizeBits, unsigned int
    keySizeWords)

int SPECK_CBC_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* cbcKey, unsigned char *IV, unsigned int wordSizeBits, unsigned int
    keySizeWords)
```

The inputs are:

- `unsigned char* inputBuffer`: the buffer to encrypt or decrypt

- `unsigned int inputSizeBytes`: the length in bytes of the buffer to encrypt or decrypt
- `unsigned char* outputBuffer`: the output buffer containing the resulting encryption or decryption
- `unsigned int *outputSizeBytes`: the length in bytes of the output buffer
- `unsigned char* cbcKey`: the symmetric key
- `unsigned char *IV`: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- `unsigned int wordSizeBits`: the length of the words, 16, 24, 32, 48 or 64 bits
- `unsigned int keySizeWords`: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 48: Example of how to encrypt or decrypt a buffer with SPECK-CBC

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "012345678";
unsigned char* IV = "888811";
message[10] = 't';

err = SPECK_CBC_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, cbcKey, IV, 24,
    3);
if(err != 0) return err;
err = SPECK_CBC_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    cbcKey, IV, 24, 3);
if(err != 0) return err;
```

**Encrypt or decrypt a string** To encrypt or decrypt a string with the CBC mode, we use the following functions:

```
int SPECK_CBC_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* cbcKey, unsigned char* IV, unsigned int
    wordSizeBits, unsigned int keySizeWords)

int SPECK_CBC_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* cbcKey, unsigned char *IV, unsigned int
    wordSizeBits, unsigned int keySizeWords)
```

The inputs are:

- `unsigned char* inputString`: the string to encrypt
- `unsigned char* inputBuffer`: the buffer to decrypt
- `unsigned int inputSizeBytes`: the length in bytes of the buffer to decrypt
- `unsigned int *outputSizeBytes`: the length in bytes of the output buffer
- `unsigned char* outputBuffer`: the output buffer containing the resulting encryption

- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* cbcKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits
- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 49: Example of how to encrypt or decrypt a string with SPECK-CBC

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "012345678";
unsigned char* IV = "888811";

err = SPECK_CBC_EncryptString(message, cipher, &cipherLength, cbcKey, IV, 24, 3);
if(err != 0) return err;
err = SPECK_CBC_DecryptString(cipher, cipherLength, decipher, cbcKey, IV, 24, 3);
if(err != 0) return err;
```

### 8.1.3 OFB

In OFB mode, an initialisation vector is encrypted with SPECK, then XORed with the first block of clear text, to obtain the first block of cipher text. Then this encrypted initialisation vector is reused as the initialisation vector for the next block.

**Encrypt or decrypt a file** To encrypt or decrypt a file with the SPECK-OFB algorithm, we use the following functions:

```
int SPECK_OFB_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ofbKey, unsigned char *IV, unsigned int wordSizeBits, unsigned int keySizeWords)

int SPECK_OFB_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ofbKey, unsigned char* IV, unsigned int wordSizeBits, unsigned int keySizeWords)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ofbKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits
- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 50: Example of how to encrypt or decrypt a file with SPECK-OFB

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ofbKey = "012345678";
unsigned char* IV = "888811";

err = SPECK_OFB_EncryptFile(message, cipher, ofbKey, IV, 24, 3);
if(err != 0) return err;
err = SPECK_OFB_DecryptFile(cipher, decipher, ofbKey, IV, 24, 3);
if(err != 0) return err;
```

**Encrypt or decrypt a buffer** To encrypt or decrypt a buffer with the OFB mode, we use the following functions:

```
int SPECK_OFB_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned
    char* ofbKey, unsigned char* IV, unsigned int wordSizeBits, unsigned int
    keySizeWords)

int SPECK_OFB_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* ofbKey, unsigned char *IV, unsigned int wordSizeBits, unsigned int
    keySizeWords)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* ofbKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits
- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 51: Example of how to encrypt or decrypt a buffer with SPECK-OFB

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "012345678";
unsigned char* IV = "888811";
```

```
message[10] = 't';

err = SPECK_OFB_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ofbKey, IV, 24,
    3);
if(err != 0) return err;
err = SPECK_OFB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ofbKey, IV, 24, 3);
if(err != 0) return err;
```

**Encrypt or decrypt a string** To encrypt or decrypt a string with the OFB mode, we use the following functions:

```
int SPECK_OFB_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* ofbKey,  unsigned char* IV, unsigned
    int wordSizeBits, unsigned int keySizeWords)

int SPECK_OFB_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ofbKey, unsigned char *IV, unsigned int
    wordSizeBits, unsigned int keySizeWords)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* ofbKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int wordSizeBits: the length of the words, 16, 24, 32, 48 or 64 bits
- unsigned int keySizeWords: the number of words of the key, 2, 3 or 4 depending on wordSizeBits

Listing 52: Example of how to encrypt or decrypt a string with SPECK-OFB

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ofbKey = "012345678";
unsigned char* IV = "888811";

err = SPECK_OFB_EncryptString(message, cipher, &cipherLength, ofbKey, IV, 24, 3);
if(err != 0) return err;
err = SPECK_OFB_DecryptString(cipher, cipherLength, decipher, ofbKey, IV, 24, 3);
if(err != 0) return err;
```

## 8.2 For wordSizeBits = 32

The most used mode is when the size of the words is 32. We have developed optimized functions for this mode.

### 8.2.1 ECB

**Encrypt or decrypt a file** To encrypt or decrypt a file with the SPECK-ECB algorithm (with wordSizeBits = 32), we use the following functions:

```
int SPECK32_ECB_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ecbKey, unsigned int keySizeBytes)

int SPECK32_ECB_DecryptFile(char* inputFilePath, char* outputFilePath,unsigned char*
    ecbKey, unsigned int keySizeBytes)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ecbKey: the symmetric key
- unsigned int keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes.

Listing 53: Example of how to encrypt or decrypt a file with SPECK32-ECB

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ecbKey = "0123456789123456";

err = SPECK32_ECB_EncryptFile(message, cipher, ecbKey, 16);
if(err != 0) return err;
err = SPECK32_ECB_DecryptFile(cipher, decipher, ecbKey, 16);
if(err != 0) return err;
```

**Encrypt or decrypt a buffer** To encrypt or decrypt a buffer with the ECB mode (with wordSizeBits = 32), we use the following functions:

```
int SPECK32_ECB_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned
    char* ecbKey, unsigned int keySizeBytes)

int SPECK32_ECB_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* ecbKey, unsigned int keySizeBytes)
```

The inputs are:

- `unsigned char* inputBuffer`: the buffer to encrypt or decrypt
- `unsigned int inputSizeBytes`: the length in bytes of the buffer to encrypt or decrypt
- `unsigned char* outputBuffer`: the output buffer containing the resulting encryption or decryption
- `unsigned int *outputSizeBytes`: the length in bytes of the output buffer
- `unsigned char* ecbKey`: the symmetric key
- `unsigned int keySizeBytes`: the number of bytes of the key, 8, 12 or 16 bytes.

Listing 54: Example of how to encrypt or decrypt a buffer with SPECK32-ECB

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789123456";
message[10] = 't';

err = SPECK32_ECB_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ecbKey, 16);
if(err != 0) return err;
err = SPECK32_ECB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ecbKey, 16);
if(err != 0) return err;
```

**Encrypt or decrypt a string** To encrypt or decrypt a string with the ECB mode (with wordSizeBits = 32), we use the following functions:

```
int SPECK32_ECB_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* ecbKey, unsigned int keySizeBytes)

int SPECK32_ECB_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ecbKey, unsigned int keySizeBytes)
```

The inputs are:

- `unsigned char* inputString`: the string to encrypt
- `unsigned char* inputBuffer`: the buffer to decrypt
- `unsigned int inputSizeBytes`: the length in bytes of the buffer to decrypt
- `unsigned int *outputSizeBytes`: the length in bytes of the output buffer
- `unsigned char* outputBuffer`: the output buffer containing the resulting encryption
- `unsigned char* outputString`: the output string containing the resulting decryption
- `unsigned char* ecbKey`: the symmetric key
- `unsigned int keySizeBytes`: the number of bytes of the key, 8, 12 or 16 bytes.

Listing 55: Example of how to encrypt or decrypt a string with SPECK32-ECB

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789123456";

err = SPECK32_ECB_EncryptString(message, cipher, &cipherLength, ecbKey, 16);
if(err != 0) return err;
err = SPECK32_ECB_DecryptString(cipher, cipherLength, decipher, ecbKey, 16);
if(err != 0) return err;
```

### 8.2.2 CBC

**Encrypt or decrypt a file** To encrypt or decrypt a file with the SPECK32-CBC algorithm, we use the following functions:

```
int SPECK32_CBC_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    cbcKey, unsigned char *IV, unsigned int keySizeBytes);

int SPECK32_CBC_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    cbcKey, unsigned char* IV, unsigned int keySizeBytes);
```

The inputs are:

- `char*` inputFilePath: the path to the file to encrypt or decrypt
- `char*` outputFilePath: the path to the output file containing the resulting encryption or decryption
- `unsigned char*` cbcKey: the symmetric key
- `unsigned char *`IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- `unsigned int` keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes.

Listing 56: Example of how to encrypt or decrypt a file with SPECK32-CBC

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* cbcKey = "0123456789123456";
unsigned char* IV = "01234567";

err = SPECK32_CBC_EncryptFile(message, cipher, cbcKey, IV, 16);
if(err != 0) return err;
err = SPECK32_CBC_DecryptFile(cipher, decipher, cbcKey, IV, 16);
if(err != 0) return err;
```

**Encrypt or decrypt a buffer** To encrypt or decrypt a buffer with SPECK32-CBC, we use the following functions:

```
int SPECK32_CBC_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned
    char* cbcKey, unsigned char* IV, unsigned int keySizeBytes)

int SPECK32_CBC_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* cbcKey, unsigned char *IV, unsigned int keySizeBytes)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* cbcKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes.

Listing 57: Example of how to encrypt or decrypt a buffer with SPECK32-CBC

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789123456";
unsigned char* IV = "88881111";
message[10] = 't';

err = SPECK32_CBC_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, cbcKey, IV,
    16);
if(err != 0) return err;
err = SPECK32_CBC_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    cbcKey, IV, 16);
if(err != 0) return err;
```

**Encrypt or decrypt a string** To encrypt or decrypt a string with SPECK32-CBC, we use the following functions:

```
int SPECK32_CBC_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* cbcKey, unsigned char* IV, unsigned int
    keySizeBytes)

int SPECK32_CBC_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* cbcKey, unsigned char *IV, unsigned int
    keySizeBytes)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* cbcKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes

Listing 58: Example of how to encrypt or decrypt a string with SPECK32-CBC

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789123456";
unsigned char* IV = "88881111";

err = SPECK32_CBC_EncryptString(message, cipher, &cipherLength, cbcKey, IV, 16);
if(err != 0) return err;
err = SPECK32_CBC_DecryptString(cipher, cipherLength, decipher, cbcKey, IV, 16);
if(err != 0) return err;
```

### 8.2.3 OFB

**Encrypt or decrypt a file** To encrypt or decrypt a file with the SPECK32-OFB algorithm, we use the following functions:

```
int SPECK32_OFB_EncryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ofbKey, unsigned char *IV, unsigned int keySizeBytes)

int SPECK32_OFB_DecryptFile(char* inputFilePath, char* outputFilePath, unsigned char*
    ofbKey, unsigned char* IV, unsigned int keySizeBytes)
```

The inputs are:

- char* inputFilePath: the path to the file to encrypt or decrypt
- char* outputFilePath: the path to the output file containing the resulting encryption or decryption
- unsigned char* ofbKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)

- unsigned int keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes

Listing 59: Example of how to encrypt or decrypt a file with SPECK32-OFB

```
int err;
char* message = "message.txt";
char* cipher = "cipher.txt";
char* decipher = "decipher.txt";
unsigned char* ofbKey = "0123456789123456";
unsigned char* IV = "88881111";

err = SPECK32_OFB_EncryptFile(message, cipher, ofbKey, IV, 16);
if(err != 0) return err;
err = SPECK32_OFB_DecryptFile(cipher, decipher, ofbKey, IV, 16);
if(err != 0) return err;
```

**Encrypt or decrypt a buffer** To encrypt or decrypt a buffer with the OFB mode (with wordSizeBits = 32), we use the following functions:

```
int SPECK32_OFB_EncryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int* outputSizeBytes, unsigned
    char* ofbKey, unsigned char* IV, unsigned int keySizeBytes)

int SPECK32_OFB_DecryptBinaryBuffer(unsigned char* inputBuffer, unsigned int
    inputSizeBytes, unsigned char* outputBuffer, unsigned int *outputSizeBytes, unsigned
    char* ofbKey, unsigned char *IV,  unsigned int keySizeBytes)
```

The inputs are:

- unsigned char* inputBuffer: the buffer to encrypt or decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to encrypt or decrypt
- unsigned char* outputBuffer: the output buffer containing the resulting encryption or decryption
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* ofbKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes

Listing 60: Example of how to encrypt or decrypt a buffer with SPECK32-OFB

```
int err;
int cipherLength, decipherLength;
unsigned char message[20];
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ecbKey = "0123456789123456";
unsigned char* IV = "88881111";
message[10] = 't';
```

```
err = SPECK32_OFB_EncryptBinaryBuffer(message, 20, cipher, &cipherLength, ofbKey, IV,
    16);
if(err != 0) return err;
err = SPECK32_OFB_DecryptBinaryBuffer(cipher, cipherLength, decipher, &decipherLength,
    ofbKey, IV, 16);
if(err != 0) return err;
```

**Encrypt or decrypt a string** To encrypt or decrypt a string with the OFB mode (with wordSizeBits = 32), we use the following functions:

```
int SPECK32_OFB_EncryptString(unsigned char* inputString, unsigned char* outputBuffer,
    unsigned int *outputSizeBytes, unsigned char* ofbKey, unsigned char* IV, unsigned int
    keySizeBytes)

int SPECK32_OFB_DecryptString(unsigned char* inputBuffer, unsigned int inputSizeBytes,
    unsigned char* outputString, unsigned char* ofbKey, unsigned char *IV, unsigned int
    keySizeBytes)
```

The inputs are:

- unsigned char* inputString: the string to encrypt
- unsigned char* inputBuffer: the buffer to decrypt
- unsigned int inputSizeBytes: the length in bytes of the buffer to decrypt
- unsigned int *outputSizeBytes: the length in bytes of the output buffer
- unsigned char* outputBuffer: the output buffer containing the resulting encryption
- unsigned char* outputString: the output string containing the resulting decryption
- unsigned char* ofbKey: the symmetric key
- unsigned char *IV: the initialisation vector (with length $\frac{wordSizeBits}{4}$ bytes)
- unsigned int keySizeBytes: the number of bytes of the key, 8, 12 or 16 bytes

Listing 61: Example of how to encrypt or decrypt a string with SPECK32-OFB

```
int err, cipherLength;
unsigned char *message = ".a_message_of_test.";
unsigned char cipher[100];
unsigned char decipher[100];
unsigned char* ofbKey = "0123456789123456";
unsigned char* IV = "88881111";

err = SPECK32_OFB_EncryptString(message, cipher, &cipherLength, ofbKey, IV, 16);
if(err != 0) return err;
err = SPECK32_OFB_DecryptString(cipher, cipherLength, decipher, ofbKey, IV, 16);
if(err != 0) return err;
```

# 9 PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) is a key derivation function that is part of RSA Laboratories' Public-Key Cryptography Standards (PKCS) series, specifically PKCS #5 v2.0, also published as Internet Engineering Task Force's RFC 2898. It replaces an earlier standard, PBKDF1, which could only produce derived keys up to 160 bits long.

PBKDF2 applies a pseudorandom function, such as a cryptographic hash, cipher, or HMAC, to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult, and is known as key stretching. It is described in the NIST Special Publication 800-132.

To generate a key from a password using PBKDF2, we use the following function:

```
int PBKDF2_KeyDerivationFunction(unsigned char* password, unsigned char salt[60],
    unsigned int counter, unsigned int outputSizeBits, unsigned char* output)
```

The inputs are:

- unsigned char* password: the password to derive
- unsigned char salt[60]: the salt on 60 bytes
- unsigned int counter: the number of iterations of the algorithm
- unsigned int outputSizeBits: the number of bits of the desired output
- unsigned char* output: the resulting key

Listing 62: Example of how to use PBKDF2 algorithm

```
int err;
unsigned char* password = "toto41";
unsigned char salt = "012345678901234567890123456789012345678901234567890123456789";
unsigned char output[16];

err = PBKDF2_KeyDerivationFunction(password, salt, 10000, 128, output);
if (err != 0) return err;
```

# 10 Generation of random elements

To use some cryptographic algorithm, we need to have a cryptographic random generator. In this library, we use the Windows CryptGenRandom function or the Linux or Apple /dev/random function depending on the operating system. We have implemented two algorithms:

```
int RandomInt(void)

int RandomBuffer(unsigned int len, unsigned char* MyBuffer)
```

The output of the first function is the desired random integer.
The inputs of the second function are:

- `unsigned int len`: the number of bytes of the buffer to fill with random characters
- `unsigned char* MyBuffer`: the buffer to fill with random characters

Listing 63: Example of how to generate a random integer and a random buffer

```
int err, i;
unsigned char buffer[20];

i = RandomInt();
RandomBuffer(20, buffer);
```

# 11 Error management

When a library function returns an error code, it is important to know the signification of this code. We have implemented a function that returns an error message associated with each error code:

```
void ErrorMessage(int error, char* message)
```

The inputs are:

- int error: the error code returned by a library function
- char* message: an output buffer containing the message code associated with the error code

Listing 64: Example of how to know the signification of an error code

```
int err = -203;
char errorMess[200];

ErrorMessage(err, errorMess);
```